

BASE DE DATOS I

Profesor: Jorge Insfran

Año: 2023



UNIDAD 01

INTRODUCCIÓN A LAS BASES DE DATOS

CONCEPTO Y ORIGEN DE LAS BASES DE DATOS Y DE LOS SGBD

Las aplicaciones informáticas de los años sesenta se realizaban totalmente por lotes (**batch**) y estaban pensadas para una tarea muy específica relacionada con muy pocas entidades tipo.

Cada aplicación (una o varias cadenas de programas) utilizaba archivos de movimientos para actualizar (creando una copia nueva) y/o para consultar uno o dos archivos maestros o, excepcionalmente, más de dos. Cada programa trataba como máximo un archivo maestro, que solía estar sobre cinta magnética y, en consecuencia, se trabajaba con **acceso secuencial**. Cada vez que se le quería añadir una aplicación que requería el uso de algunos de los datos que ya existían y de otros nuevos, se diseñaba un archivo nuevo con todos los datos necesarios (algo que provocaba redundancia) para evitar que los programas tuvieran que leer muchos archivos.

A medida que se fueron introduciendo las líneas de comunicación, los terminales y los discos, se fueron escribiendo programas que permitían a varios usuarios consultar los mismos archivos on-line y de forma simultánea. Más adelante fue surgiendo la necesidad de hacer las actualizaciones también on-line.

A medida que se integraban las aplicaciones, se tuvieron que interrelacionar sus archivos y fue necesario eliminar la redundancia. El nuevo conjunto de archivos se debía diseñar de modo que estuviesen interrelacionados; al mismo tiempo, las informaciones redundantes (como, por ejemplo, el nombre y la dirección de los clientes o el nombre y el precio de los productos), que figuraban en los archivos de más de una de las aplicaciones, debían estar ahora en un solo lugar.

El acceso on-line y la utilización eficiente de las interrelaciones exigían estructuras físicas que diesen un acceso rápido, como por ejemplo los índices, las multilistas, las técnicas de hashing, etc.

Estos conjuntos de archivos interrelacionados, con estructuras complejas y compartidos por varios procesos de forma simultánea (unos on-line y otros por lotes), recibieron al principio el nombre de **Data Banks**, y después, a inicios de los años setenta, el de **Data Bases**. Aquí los denominamos bases de datos (BD).

El software de gestión de archivos era demasiado elemental para dar satisfacción a todas estas necesidades. Por ejemplo, el tratamiento de las interrelaciones no estaba previsto, no era posible que varios usuarios actualizaran datos simultáneamente, etc. La utilización de estos conjuntos de archivos por parte de los programas de aplicación era excesivamente compleja, de modo que, especialmente durante la segunda mitad de los años setenta, fue saliendo al mercado software más sofisticado: los **Data Base Management Systems**, que aquí denominamos sistemas de gestión de Bases de Datos (SGBD).

Aplicaciones informáticas de los años sesenta

La emisión de facturas, el control de pedidos pendientes de atender, el mantenimiento del archivo de productos o la nómina del personal eran algunas de las aplicaciones informáticas habituales

Integración de aplicaciones

Por ejemplo, se integra la aplicación de facturas, la de pedidos pendientes y la gestión del archivo de productos.



Con todo lo que hemos dicho hasta ahora, podríamos definir el término Bases de Datos; una base de datos de un Sistema de Información es la representación integrada de los conjuntos de entidades instancia correspondientes a las diferentes entidades tipo del Sistema de Información y de sus interrelaciones. Esta representación informática (o conjunto estructurado de datos) debe poder ser utilizada de forma compartida por muchos usuarios de distintos tipos.

En otras palabras, una base de datos es un conjunto estructurado de datos que representa entidades y sus interrelaciones. La representación será única e integrada, a pesar de que debe permitir utilizaciones varias y simultáneas.

LOS ARCHIVOS TRADICIONALES Y LAS BASES DE DATOS

Aunque de forma muy simplificada, podríamos enumerar las principales diferencias entre los archivos tradicionales y las Bases de Datos tal y como se indica a continuación:

- 1) Entidades tipos:
 - Archivos: tienen registros de una sola entidad tipo.
 - BD: tienen datos de varias entidades tipo.
- 2) Interrelaciones:
 - Archivos: el sistema no interrelaciona archivos.
 - BD: el sistema tiene previstas herramientas para interrelacionar entidades.
- 3) Redundancia:
 - Archivos: se crean archivos a la medida de cada aplicación, con todos los datos necesarios, aunque algunos sean redundantes respecto de otros archivos.
 - BD: todas las aplicaciones trabajan con la misma Base de Datos y la integración de los datos es básica, de modo que se evita la redundancia.
- 4) Usuarios
 - Archivos: sirven para un solo usuario o una sola aplicación. Dan una sola visión del mundo real.
 - BD: es compartida por muchos usuarios de distintos tipos. Ofrece varias visiones del mundo real.

EVOLUCIÓN DE LOS SGBD

Para entender mejor qué son los SGBD, haremos un repaso de su evolución desde los años sesenta hasta nuestros días.

LOS AÑOS SESENTA Y SETENTA: SISTEMAS CENTRALIZADOS

Los SGBD de los años sesenta y setenta (IMS de IBM, IDS de Bull, DMS de UNIVAC, etc.) eran sistemas **totalmente centralizados**, como corresponde a los sistemas operativos de aquellos años, y al hardware para el que estaban hechos: una gran computadora para toda la empresa y una red de terminales sin inteligencia ni memoria. Los primeros SGBD - en los años sesenta todavía no se les denominaba así- estaban orientados a facilitar la utilización de grandes conjuntos de datos en los que las interrelaciones eran complejas. El arquetipo de aplicación era el *Bill of materials* o *Parts explosion*, típica en las industrias del automóvil, en la construcción de naves espaciales y en campos similares. Estos sistemas trabajaban exclusivamente por lotes (batch).



Al aparecer los terminales de teclado, conectados a la computadora central mediante una línea telefónica, se empiezan a construir grandes aplicaciones on-line transaccionales (OLTP). Los SGBD estaban íntimamente ligados al software de comunicaciones y de gestión de transacciones.

Aunque para escribir los programas de aplicación se utilizaban lenguajes de alto nivel como Cobol o PL/I, se disponía también de instrucciones y de subrutinas especializadas para tratar las Bases de Datos que requerían que el programador conociese muchos detalles del diseño físico, y que hacían que la programación fuese muy compleja.

Puesto que los programas estaban relacionados con el nivel físico, se debían modificar continuamente cuando se hacían cambios en el diseño y la organización de la Base de Datos. La preocupación básica era maximizar el rendimiento: el tiempo de respuesta y las transacciones por segundo

El Data Base / Data Communications

IBM denominaba Data Base/ Data Communications (DB/DC) al software de comunicaciones y de gestión de transacciones y de datos. Las aplicaciones típicas eran la reserva/compra de billetes a las compañías aéreas y de ferrocarriles y, un poco más tarde, las cuentas de clientes en el mundo bancario.

LOS AÑOS OCHENTA: SGBD RELACIONALES

Las computadoras minis, en primer lugar, y después las computadoras micros, extendieron la informática a prácticamente todas las empresas e instituciones.

Esto exigía que el desarrollo de aplicaciones fuese más sencillo. Los SGBD de los años setenta eran demasiado complejos e inflexibles, y sólo los podía utilizar un personal muy cualificado.

La aparición de los SGBD relacionales¹ supone un avance importante para facilitar la programación de aplicaciones con Bases de Datos y para conseguir que los programas sean independientes de los aspectos físicos de la Base de Datos.

Todos estos factores hacen que se extienda el uso de los SGBD. La estandarización, en el año 1986, del lenguaje SQL produjo una auténtica explosión de los SGBD relacionales.

LAS COMPUTADORAS PERSONALES

Durante los años ochenta aparecen y se extienden muy rápidamente las computadoras personales. También surge software para estos equipos monousuario (por ejemplo, dBase y sus derivados, Access), con los cuales es muy fácil crear y utilizar conjuntos de datos, y que se denominan **personal data bases**. Noten que el hecho de denominar SGBD estos primeros sistemas para PC es un poco forzado, ya que no aceptaban estructuras complejas ni interrelaciones, ni podían ser utilizados en una red que sirviese simultáneamente a muchos usuarios de diferentes tipos. Sin embargo, algunos, con el tiempo, se han ido convirtiendo en auténticos SGBD.

LOS AÑOS NOVENTA: DISTRIBUCIÓN, C/S Y 4GL

Al acabar la década de los ochenta, los SGBD relacionales ya se utilizaban prácticamente en todas las empresas. A pesar de todo, hasta la mitad de los noventa, cuando se ha necesitado un rendimiento elevado se han seguido utilizando los SGBD pre relacionales.

¹ Oracle aparece en el año 1980



A finales de los ochenta y principios de los noventa, las empresas se han encontrado con el hecho de que sus departamentos han ido comprando computadoras departamentales y personales, y han ido haciendo aplicaciones con Bases de Datos. El resultado ha sido que en el seno de la empresa hay numerosas Bases de Datos y varios SGBD de diferentes tipos o proveedores. Este fenómeno de multiplicación de las Bases de Datos y de los SGBD se ha visto incrementado por la fiebre de las fusiones de empresas.

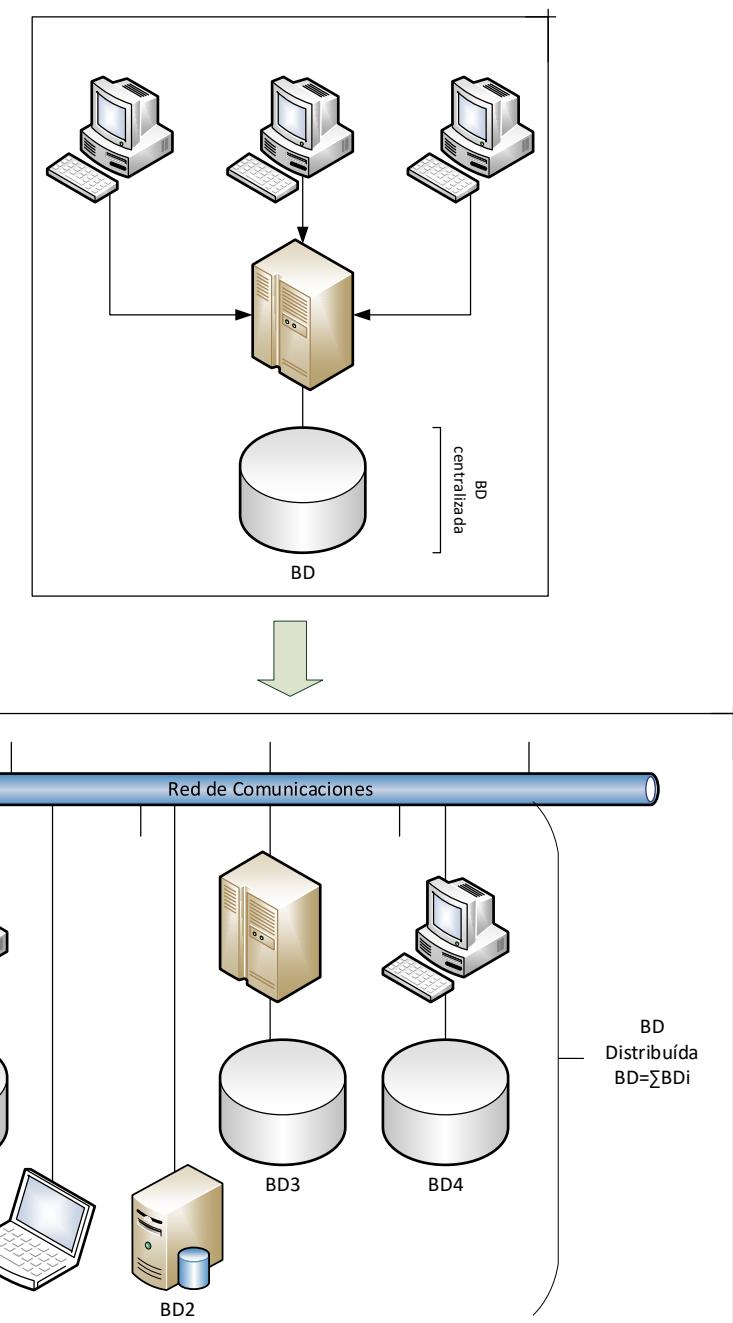
La necesidad de tener una visión global de la empresa y de interrelacionar diferentes aplicaciones que utilizan Bases de Datos diferentes, junto con la facilidad que dan las redes para la intercomunicación entre computadoras, ha conducido a los SGBD actuales, que permiten que un programa pueda trabajar con diferentes Bases de Datos como si se tratase de una sola. Es lo que se conoce como base de datos distribuida.

Esta distribución ideal se consigue cuando las diferentes Bases de Datos son soportadas por una misma marca de SGBD, es decir, cuando hay homogeneidad. Sin embargo, esto no es tan sencillo si los SGBD son heterogéneos. En la actualidad, gracias principalmente a la estandarización del lenguaje SQL, los SGBD de marcas diferentes pueden darse servicio unos a otros y colaborar para dar servicio a un programa de aplicación. No obstante, en general, en los casos de heterogeneidad no se llega a poder dar en el programa que los utiliza la apariencia de que se trata de una única Base de Datos.



Además de esta distribución “impuesta”, al querer tratar de forma integrada distintas Bases de Datos preexistentes, también se puede hacer una distribución “deseada”, diseñando una Base de Datos distribuida físicamente, y con ciertas partes replicadas en diferentes sistemas. Las razones básicas por las que interesa esta distribución son las siguientes:

- 1) Disponibilidad. La disponibilidad de un sistema con una Base de Datos distribuida puede ser más alta, porque si queda fuera de servicio uno de los sistemas, los demás seguirán funcionando. Si los datos residentes en el sistema no disponible están replicados en otro sistema, continuarán estando disponibles. En caso contrario, sólo estarán disponibles los datos de los demás sistemas.
- 2) Costo. Una Base de Datos distribuida puede reducir el costo. En el caso de un sistema centralizado, todos los equipos usuarios, que pueden estar distribuidos por distintas y lejanas áreas geográficas, están conectados al sistema central por medio de líneas de comunicación. El costo total de las comunicaciones se puede reducir haciendo que un usuario tenga más cerca los datos que utiliza con mayor frecuencia; por ejemplo, en una computadora de su propia oficina o, incluso, en su computadora personal.

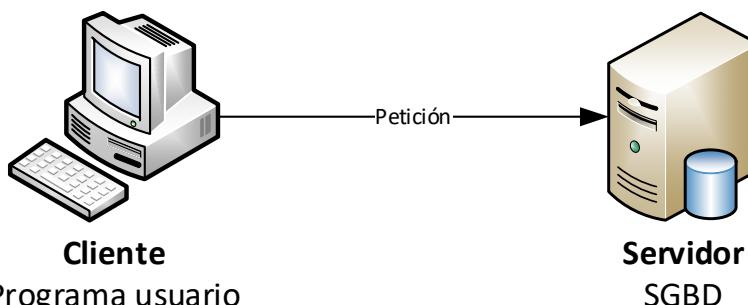


La tecnología que se utiliza habitualmente para distribuir datos es la que se conoce como entorno (o arquitectura) cliente/servidor (C/S). Todos los SGBD relacionales del mercado han sido adaptados a este entorno.

La idea del C/S es sencilla. Dos procesos diferentes, que se ejecutan en un mismo sistema o en sistemas separados, actúan de forma que uno tiene el papel de cliente o peticionario de un servicio, y el otro el de servidor o proveedor del servicio.



Por ejemplo, un programa de aplicación que un usuario ejecuta en su PC (que está conectado a una red) pide ciertos datos de una Base de Datos que reside en un equipo UNIX donde, a su vez, se ejecuta el SGBD relacional que la gestiona. El programa de aplicación es el cliente y el SGBD es el servidor.



Otros servicios

Noten que el servicio que da un servidor de un sistema C/S no tiene por qué estar relacionado con las Bases de Datos; puede ser un servicio de impresión, de envío de un fax, etc., pero aquí nos interesan los servidores que son SGBD.

Un proceso cliente puede pedir servicios a varios servidores. Un servidor puede recibir peticiones de muchos clientes. En general, un proceso A que hace de cliente, pidiendo un servicio a otro proceso B puede hacer también de servidor de un servicio que le pida otro proceso C (o incluso el B, que en esta petición sería el cliente). Incluso el cliente y el servidor pueden residir en un mismo sistema.

La facilidad para disponer de distribución de datos no es la única razón, ni siquiera la básica, del gran éxito de los entornos C/S en los años noventa. Tal vez el motivo fundamental ha sido la flexibilidad para construir y hacer crecer la configuración informática global de la empresa, así como de hacer modificaciones en ella, mediante hardware y software muy estándar y barato.

El éxito de las Bases de Datos, incluso en sistemas personales, ha llevado a la aparición de los **Fourth Generation Languages** (4GL), lenguajes muy fáciles y potentes, especializados en el desarrollo de aplicaciones fundamentadas en Bases de Datos. Proporcionan muchas facilidades en el momento de definir, generalmente de forma visual, diálogos para introducir, modificar y consultar datos en entornos C/S.

C/S, SQL y 4GL...

... son siglas de moda desde el principio de los años noventa en el mundo de los sistemas de información.

TENDENCIAS ACTUALES

Hoy día, los SGBD relativos están en plena transformación para adaptarse a tres tecnologías de éxito reciente, fuertemente relacionadas: la multimedia, la de orientación a objetos (OO) e Internet y la web.

Los tipos de datos que se pueden definir en los SGBD relativos de los años ochenta y noventa son muy limitados. La incorporación de tecnologías multimedia –imagen y sonido– en los Sistemas de Información hace necesario que los SGBD relativos acepten atributos de estos tipos.

Sin embargo, algunas aplicaciones no tienen suficiente con la incorporación de tipos especializados en multimedia. Necesitan tipos complejos que el desarrollador pueda definir a medida de la aplicación. En definitiva, se necesitan **tipos abstractos de datos**: TAD. Los SGBD más recientes ya incorporaban esta posibilidad, y abren un amplio mercado de TAD predefinidos o bibliotecas de clases.

Nos puede interesar, ...

... por ejemplo, tener en la entidad alumno un atributo foto tal que su valor sea una tira de bits muy larga, resultado de la digitalización de la fotografía del alumno.



Esto nos lleva a la **orientación a objetos** (OO). El éxito de la OO al final de los años ochenta, en el desarrollo de software básico, en las aplicaciones de ingeniería industrial y en la construcción de interfaces gráficas con los usuarios, ha hecho que durante la década de los noventa se extendiese en prácticamente todos los campos de la informática.

En los Sistemas de Información se inicia también la adopción, tímida de momento, de la OO. La utilización de lenguajes como C++, C# o Java requiere que los SGBD relacionales se adapten a ellos con interfaces adecuadas.

La rápida adopción de la web a los Sistemas de Información hace que los SGBD incorporen recursos para ser servidores de páginas web, como por ejemplo la inclusión de SQL en scripts HTML, SQL incorporado en Java, etc. Noten que en el mundo de la web son habituales los datos multimedia y la OO.

Durante estos últimos años se ha empezado a extender un tipo de aplicación de las Bases de Datos denominado **Data Warehouse**, o almacén de datos, que también produce algunos cambios en los SGBD relacionales del mercado.

A lo largo de los años que han trabajado con Bases de Datos de distintas aplicaciones, las empresas han ido acumulando gran cantidad de datos de todo tipo. Si estos datos se analizan convenientemente pueden dar información valiosa².

Por lo tanto, se trata de mantener una gran Base de Datos con información proveniente de toda clase de aplicaciones de la empresa (e, incluso, de fuera). Los datos de este gran almacén, el Data Warehouse, se obtienen por una replicación más o menos elaborada de los que hay en las Bases de Datos que se utilizan en el trabajo cotidiano de la empresa. Estos almacenes de datos se utilizan exclusivamente para hacer consultas, de forma especial para que lleven a cabo estudios³ los analistas financieros, los analistas de mercado, etc.

Actualmente, los SGBD se adaptan a este tipo de aplicación, incorporando, por ejemplo, herramientas como las siguientes:

- a) La creación y el mantenimiento de réplicas, con una cierta elaboración de los datos.
- b) La consolidación de datos de orígenes diferentes.
- c) La creación de estructuras físicas que soporten eficientemente el análisis multidimensional.

OBJETIVOS Y SERVICIOS DE LOS SGBD

Los SGBD que actualmente están en el mercado pretenden satisfacer un conjunto de objetivos directamente deducibles de lo que hemos explicado hasta ahora.

CONSULTAS NO PREDEFINIDAS Y COMPLEJAS

El objetivo fundamental de los SGBD es permitir que se hagan consultas no predefinidas (ad hoc) y complejas.

Consultas que afectan a más de una entidad tipo

- Se quiere conocer el número de alumnos de más de veinticinco años y con nota media superior a siete que están matriculados actualmente en la asignatura Bases de datos I.
- De cada alumno matriculado en menos de tres asignaturas, se quiere obtener el nombre, el número de matrícula, el nombre de las asignaturas y el nombre de profesores de estas asignaturas.

² Por ejemplo, la evolución del mercado en relación con la política de precios

³ Con frecuencia se trata de estadísticas multidimensionales



Los usuarios podrán hacer consultas de cualquier tipo y complejidad directamente al SGBD. El SGBD tendrá que responder inmediatamente sin que estas consultas estén preestablecidas; es decir, sin que se tenga que escribir, compilar y ejecutar un programa específico para cada consulta.

El usuario debe formular la consulta con un lenguaje sencillo (que se quede, obviamente, en el nivel lógico), que el sistema debe interpretar directamente.

Sin embargo, esto no significa que no se puedan escribir programas con consultas incorporadas (por ejemplo, para procesos repetitivos).

La solución estándar para alcanzar este doble objetivo (consultas no predefinidas y complejas) es el *lenguaje SQL*, que veremos más adelante.

FLEXIBILIDAD E INDEPENDENCIA

La complejidad de las Bases de Datos y la necesidad de irlas adaptando a la evolución del Sistema de Información hacen que un objetivo básico de los SGBD sea dar flexibilidad a los cambios.

Interesa obtener la máxima independencia posible entre los datos y los procesos usuarios para que se pueda llevar a cabo todo tipo de cambios tecnológicos y variaciones en la descripción de la Base de Datos, sin que se deban modificar los programas de aplicación ya escritos ni cambiar la forma de escribir las consultas (o actualizaciones) directas.

Para conseguir esta independencia, tanto los usuarios que hacen consultas (o actualizaciones) directas como los profesionales informáticos que escriben programas que las llevan incorporadas, deben poder desconocer las características físicas de la Base de Datos con que trabajan. No necesitan saber nada sobre el soporte físico, ni estar al corriente de qué SO se utiliza, qué índices hay, la compresión o no compresión de datos, etc.

De este modo, se pueden hacer cambios de tecnología y cambios físicos para mejorar el rendimiento sin afectar a nadie. Este tipo de independencia recibe el nombre de independencia física de los datos.

En el mundo de los archivos ya había independencia física en un cierto grado, pero en el mundo de las Bases de Datos acostumbra a ser mucho mayor.

Sin embargo, con la independencia física no tenemos suficiente. También queremos que los usuarios (los programadores de aplicaciones o los usuarios directos) no tengan que hacer cambios cuando se modifica la descripción lógica o el esquema de la Base de Datos (por ejemplo, cuando se añaden/suprimen entidades o interrelaciones, atributos, etc.)

Independencia lógica de los datos

Por ejemplo, el hecho de suprimir el atributo fecha de nacimiento de la entidad alumno y añadir otra entidad aula no debería afectar a ninguno de los programas existentes que no utilicen el atributo fecha de nacimiento.



Y todavía más: queremos que diferentes procesos usuarios puedan tener diferentes visiones lógicas de una misma Base de Datos, y que estas visiones se puedan mantener lo más independientes posibles de la Base de Datos, y entre ellas mismas. Este tipo de independencia se denomina independencia lógica de los datos, y da flexibilidad y elasticidad a los cambios lógicos.

EJEMPLOS DE INDEPENDENCIA LÓGICA

- 1) El personal administrativo de secretaría podría tener una visión de la entidad alumno sin que fuese necesario que existiese el atributo nota. Sin embargo, los usuarios profesores (o los programas dirigidos a ellos) podrían tener una visión en la que existiese el atributo nota, pero no el atributo fecha de pago.
- 2) Decidimos ampliar la longitud del atributo nombre y lo aumentamos de treinta a cincuenta caracteres, pero no sería necesario modificar los programas que ya tenemos escritos si no nos importa que los valores obtenidos tengan sólo los primeros treinta caracteres del nombre.

Independencia lógica

Los sistemas de gestión de archivos tradicionales no dan ninguna independencia lógica. Los SGBD sí que la dan; uno de sus objetivos es conseguir la máxima posible, pero dan menos de lo que sería deseable.

PROBLEMAS DE LA REDUNDANCIA

En el mundo de los archivos tradicionales, cada aplicación utilizaba su archivo. Sin embargo, puesto que se daba mucha coincidencia de datos entre aplicaciones, se producía también mucha redundancia entre los archivos. Ya hemos dicho que uno de los objetivos de los SGBD es facilitar la eliminación de la redundancia.

Seguramente piensan que el problema de la redundancia es el espacio perdido. Antiguamente, cuando el precio del byte de disco era muy elevado, esto era un problema grave, pero actualmente prácticamente nunca lo es. ¿Qué problema hay, entonces? Simplemente, lo que todos hemos sufrido más de una vez; si tenemos algo anotado en dos lugares diferentes no pasará demasiado tiempo hasta que las dos anotaciones dejen de ser coherentes, porque habremos modificado la anotación en uno de los lugares y nos habremos olvidado de hacerlo en el otro.

Así pues, el verdadero problema es el grave riesgo de inconsistencia o incoherencia de los datos; es decir, la pérdida de integridad que las actualizaciones pueden provocar cuando existe redundancia.

Por lo tanto, convendría evitar la redundancia. En principio, nos conviene hacer que un dato sólo figure una vez en la Base de Datos. Sin embargo, esto no siempre será cierto.

Por ejemplo, para representar una interrelación entre dos entidades, se suele repetir un mismo atributo en las dos, para que una haga referencia a la otra.

Otro ejemplo podría ser el disponer de réplicas de los datos por razones de fiabilidad, disponibilidad o costes de comunicaciones.

El SGBD debe permitir que el diseñador defina datos redundantes, pero entonces tendría que ser el mismo SGBD el que hiciese automáticamente la actualización de los datos en todos los lugares donde estuviesen repetidos.



La duplicación de datos es el tipo de redundancia más habitual, pero también tenemos redundancia cuando guardamos en la Base de Datos, datos derivados (o calculados) a partir de otros datos de la misma Base de Datos. De este modo podemos responder rápidamente a consultas globales, ya que nos ahorraremos la lectura de gran cantidad de registros.

En los casos de datos derivados, para que el resultado del cálculo se mantenga consistente con los datos elementales, es necesario rehacer el cálculo cada vez que éstos se modifican. El usuario (ya sea programador o no) puede olvidarse de hacer el nuevo cálculo; por ello convendrá que el mismo SGBD lo haga automáticamente.

Datos derivados

Es frecuente tener datos numéricos acumulados o agregados: el importe total de todas las matrículas hechas hasta hoy, el número medio de alumnos por asignatura, el saldo de la caja de la oficina, etc.

INTEGRIDAD DE LOS DATOS

Nos interesará que los SGBD aseguren el mantenimiento de la calidad de los datos en cualquier circunstancia. Acabamos de ver que la redundancia puede provocar pérdida de integridad de los datos, pero no es la única causa posible.

Se podría perder la corrección o la consistencia de los datos por muchas otras razones: errores de programas, errores de operación humana, avería de disco, transacciones incompletas por corte de alimentación eléctrica, etc.

En el subapartado anterior hemos visto que podemos decir al SGBD que nos lleve el control de las actualizaciones en el caso de las redundancias, para garantizar la integridad. Del mismo modo, podemos darle otras reglas de integridad –o restricciones– para que asegure que los programas las cumplen cuando efectúan las actualizaciones.

Cuando el SGBD detecte que un programa quiere hacer una operación que va contra las reglas establecidas al definir la Base de Datos, no se lo deberá permitir, y le tendrá que devolver un estado de error.

Al diseñar una Base de Datos para un Sistema de Información concreto y escribir su esquema, no sólo definiremos los datos, sino también las reglas de integridad que queremos que el SGBD haga cumplir.

A parte de las reglas de integridad que el diseñador de la Base de Datos puede definir y que el SGBD entenderá y hará cumplir, el mismo SGBD tiene reglas de integridad inherentes al modelo de datos que utiliza y que siempre se cumplirán. Son las denominadas reglas de integridad del modelo. Las reglas definibles por parte del usuario son las reglas de integridad del usuario. El concepto de integridad de los datos va más allá de prevenir que los programas usuarios almacenen datos incorrectos. En casos de errores o desastres, también podríamos perder la integridad de los datos. El SGBD nos debe dar las herramientas para reconstruir o restaurar los datos estropeados.

Los procesos de restauración (restore o recovery) de los que todo SGBD dispone pueden reconstruir la Base de Datos y darle el estado consistente y correcto anterior al incidente. Esto se acostumbra a hacer gracias a la obtención de copias periódicas de los datos (se denominan copias de seguridad o back-up) y mediante el mantenimiento continuo de un diario (log) donde el SGBD va anotando todas las escrituras que se hacen en la Base de Datos.



CONCURRENCIA DE USUARIOS

Un objetivo fundamental de los SGBD es permitir que varios usuarios puedan acceder concurrentemente a la misma Base de Datos.

Cuando los accesos concurrentes son todos de lectura (es decir, cuando la Base de Datos sólo se consulta), el problema que se produce es simplemente de rendimiento, causado por las limitaciones de los soportes de que se dispone: pocos mecanismos de acceso independientes, movimiento del brazo y del giro del disco demasiado lentos, buffers locales demasiado pequeños, etc.

Cuando un usuario o más de uno están actualizando los datos, se pueden producir problemas de interferencia que tengan como consecuencia la obtención de datos erróneos y la pérdida de integridad de la Base de Datos.

Para tratar los accesos concurrentes, los SGBD utilizan el concepto de transacción de Base de Datos, concepto de especial utilidad para todo aquello que hace referencia a la integridad de los datos, como veremos a continuación.

Denominamos transacción de Base de Datos o, simplemente transacción, un conjunto de operaciones simples que se ejecutan como una unidad. Los SGBD deben conseguir que el conjunto de operaciones de una transacción nunca se ejecute parcialmente. O se ejecutan todas, o no se ejecuta ninguna.

EJEMPLOS DE TRANSACCIONES

- 1) Imaginemos un programa pensado para llevar a cabo la operación de transferencia de dinero de una cuenta X a otra Y. Supongamos que la transferencia efectúa dos operaciones: en primer lugar, el cargo a X y después, el abono a Y. Este programa se debe ejecutar de forma que se hagan las dos operaciones o ninguna, ya que si por cualquier razón (por ejemplo, por interrupción del flujo eléctrico) el programa ejecutase sólo el cargo de dinero a X sin abonarlos a Y, la Base de Datos quedaría en un estado incorrecto. Queremos que la ejecución de este programa sea tratada por el SGBD como una transacción de Base de Datos.
- 2) Otro ejemplo de programa que queríramos que tuviera un comportamiento de transacción podría ser el que aumentara el 30% de la nota de todos los alumnos. Si sólo aumentara la nota a unos cuantos alumnos, la Base de Datos quedaría incorrecta.

Para indicar al SGBD que damos por acabada la ejecución de la transacción, el programa utilizará la operación de COMMIT. Si el programa no puede acabar normalmente (es decir, si el conjunto de operaciones se ha hecho sólo de forma parcial), el SGBD tendrá que deshacer todo lo que la transacción ya haya hecho. Esta operación se denomina ROLLBACK.

Acabamos de observar la utilidad del concepto de transacción para el mantenimiento de la integridad de los datos en caso de interrupción de un conjunto de operaciones lógicamente unitario. Sin embargo, entre transacciones que se ejecutan concurrentemente se pueden producir problemas de interferencia que hagan obtener resultados erróneos o que comporten la pérdida de la integridad de los datos.



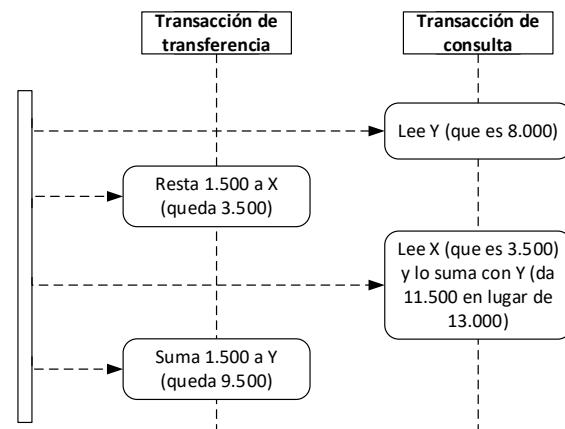
CONSECUENCIAS DE LA INTERFERENCIA ENTRE TRANSACCIONES

- 1) Imaginemos que una transacción que transfiere dinero de X a Y se ejecuta concurrentemente con una transacción que observa el saldo de las cuentas Y y X, en este orden, y nos muestra su suma. Si la ejecución de forma concurrente de las dos transacciones casualmente es tal que la transferencia se ejecuta entre la ejecución de las dos lecturas de la transacción de suma, puede producir resultados incorrectos. Además, si los decide escribir en la Base de Datos, ésta quedará inconsistente (como vemos en la figura).
- 2) Si simultáneamente con el generoso programa que aumenta la nota de los alumnos en un 30%, se ejecuta un programa que determina la nota media de todos los alumnos de una determinada asignatura, se podrá encontrar a alumnos ya gratificados y a otros no gratificados, algo que producirá resultados erróneos.

Estos son sólo dos ejemplos de las diferentes consecuencias negativas que puede tener la interferencia entre transacciones en la integridad de la Base de Datos y en la corrección del resultado de las consultas.

Transferencia de \$1.500 de la cuenta X a la cuenta Y de forma concurrente con la consulta de suma de saldos de X e Y

X	+	Y	=	Suma de Saldos
5.000		8.000		13.000



Nos interesaría que el SGBD ejecute las transacciones de forma que no se interfieran; es decir, que queden aisladas unas de otras. Para conseguir que las transacciones se ejecuten como si estuviesen aisladas, los SGBD utilizan distintas técnicas. La más conocida es el bloqueo (lock).

El bloqueo de unos datos en beneficio de una transacción consiste en poner limitaciones a los accesos que las demás transacciones podrán hacer a estos datos.

Cuando se provocan bloqueos, se producen esperas, retenciones y, en consecuencia, el sistema es más lento. Los SGBD se esfuerzan en minimizar estos efectos negativos.

SEGURIDAD

El término seguridad se ha utilizado en diferentes sentidos a lo largo de la historia de la informática.

Actualmente, en el campo de los SGBD, el término seguridad se suele utilizar para hacer referencia a los temas relativos a la confidencialidad, las autorizaciones, los derechos de acceso, etc.

Estas cuestiones siempre han sido importantes en los Sistema de Información militares, las agencias de información y en ámbitos similares, pero durante los años noventa han ido adquiriendo importancia en cualquier Sistema de Información donde se almacenen datos sobre personas.



Recuerden que en el Estado argentino tenemos una ley⁴, que exige la protección de la confidencialidad de estos datos.

Los SGBD permiten definir autorizaciones o derechos de acceso a diferentes niveles: al nivel global de toda la Base de Datos, al nivel entidad y al nivel atributo.

Estos mecanismos de seguridad requieren que el usuario se pueda identificar. Se acostumbra a utilizar códigos de usuarios (y grupos de usuarios) acompañados de contraseñas (**passwords**), pero también se utilizan tarjetas magnéticas o de presencia, identificación por reconocimiento de la voz, y otros medios biométricos.

Nos puede interesar almacenar la información con una codificación secreta; es decir, con técnicas de encriptación (como mínimo se deberían encriptar las contraseñas). Muchos de los SGBD actuales tienen prevista la encriptación.

Prácticamente todos los SGBD del mercado dan una gran variedad de herramientas para la vigilancia y la administración de la seguridad. Los hay que, incluso, tienen opciones (con precio separado) para los Sistemas de Información con unas exigencias altísimas, como por ejemplo los militares.

OTROS OBJETIVOS

Acabamos de ver los objetivos fundamentales de los SGBD actuales. Sin embargo, a medida que los SGBD evolucionan, se imponen nuevos objetivos adaptados a las nuevas necesidades y tecnologías. Como ya hemos visto, en estos momentos podríamos citar como objetivos nuevos o recientes los siguientes:

- 1) Servir eficientemente los Data Warehouse.
- 2) Adaptarse al desarrollo orientado a objetos.
- 3) Incorporar el tiempo como un elemento de caracterización de la información.
- 4) Adaptarse al mundo de Internet

ARQUITECTURA DE LOS SGBD

ESQUEMAS Y NIVELES

Para trabajar con nuestras Bases de Datos, los SGBD necesitan conocer su estructura (qué entidades tipo habrá, qué atributos tendrán, etc.).

Los SGBD necesitan que les demos una descripción o definición de la Base de Datos. Esta descripción recibe el nombre de esquema de la Base de Datos, y los SGBD la tendrán continuamente a su alcance.

El esquema de la Base de Datos es un elemento fundamental de la arquitectura de un SGBD que permite independizar el SGBD de la Base de Datos; de este modo, se puede cambiar el diseño de la Base de Datos (su esquema) sin tener que hacer ningún cambio en el SGBD.

Anteriormente, ya hemos hablado de la distinción entre dos niveles de representación informática: el nivel lógico y el físico.

⁴ Ley 25.326 PROTECCION DE LOS DATOS PERSONALES. Promulgada el 30 de octubre de 2000



El nivel lógico nos oculta los detalles de cómo se almacenan los datos, cómo se mantienen y cómo se accede físicamente a ellos. En este nivel sólo se habla de entidades, atributos y reglas de integridad.

Por cuestiones de rendimiento, nos podrá interesar describir elementos de nivel físico como, por ejemplo, qué índices tendremos y qué características presentarán, cómo y dónde (en qué espacio físico) queremos que se agrupen físicamente los registros, de qué tamaño deben ser las páginas, etc.

En el periodo 1975-1982, ANSI intentaba establecer las bases para crear estándares en el campo de las Bases de Datos. El comité conocido como ANSI/SPARC recomendó que la arquitectura de los SGBD previese tres niveles de descripción de la Base de Datos, no sólo dos⁵.

De acuerdo con la arquitectura ANSI/SPARC, debía haber tres niveles de esquemas (tres niveles de abstracción). La idea básica de ANSI/SPARC consistía en descomponer el nivel lógico en dos: el nivel externo y el nivel conceptual. Denominábamos nivel interno lo que aquí hemos denominado nivel físico.

De este modo, de acuerdo con ANSI/SPARC, habría los tres niveles de esquemas que mencionamos a continuación:

- a) En el nivel externo se sitúan las diferentes visiones lógicas que los procesos usuarios (programas de aplicación y usuarios directos) tendrán de las partes de la Base de Datos que utilizarán. Estas visiones se denominan esquemas externos.
- b) En el nivel conceptual hay una sola descripción lógica básica, única y global, que denominamos esquema conceptual, y que sirve de referencia para el resto de los esquemas.
- c) En el nivel físico hay una sola descripción física, que denominamos esquema interno.

Nota

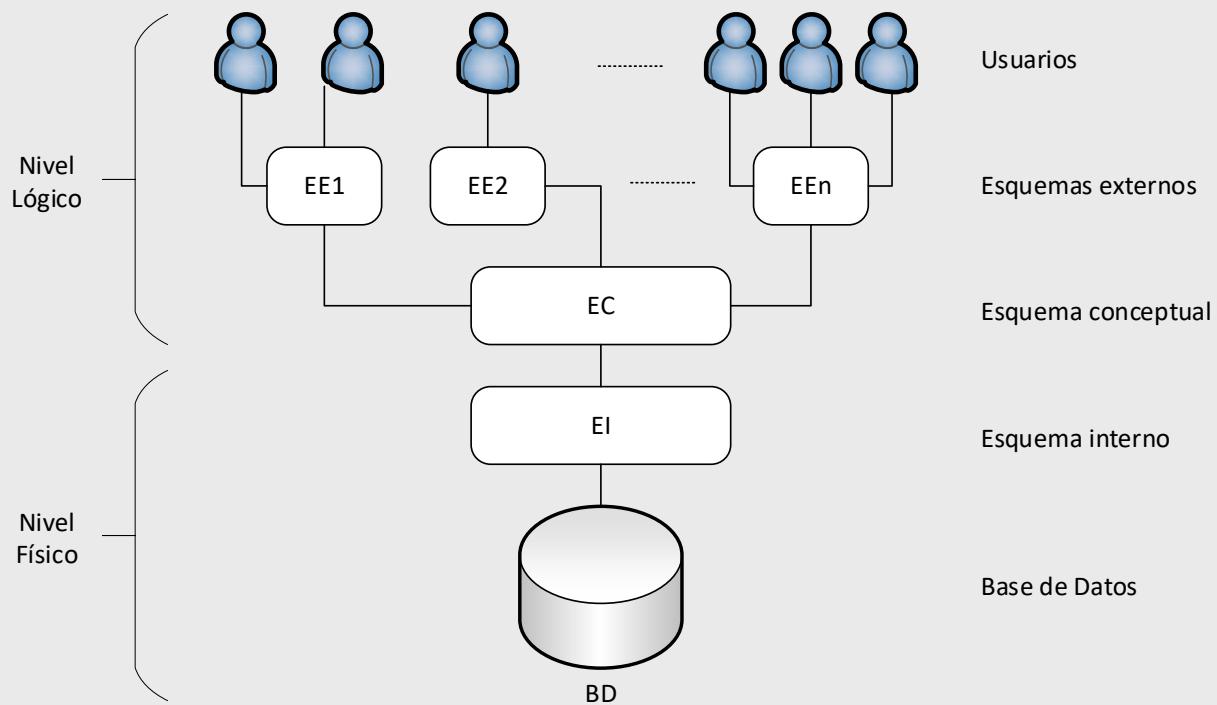
Es preciso ir con cuidado para no confundir los niveles que se describen aquí con los descritos en el caso de los archivos, aunque reciban el mismo nombre.

En el caso de las Bases de Datos, el esquema interno corresponde a la parte física, y el externo a la lógica; en el caso de los archivos, sucede lo contrario

⁵ De hecho, en el año 1971, el comité CODASYL ya había propuesto los tres niveles de esquemas



Esquemas y niveles



En el esquema conceptual se describirán las entidades tipo, sus atributos, las interrelaciones y las restricciones o reglas de integridad.

El esquema conceptual corresponde a las necesidades del conjunto de la empresa o del Sistema de Información, por lo que se escribirá de forma centralizada durante el denominado diseño lógico de la Base de Datos.

Sin embargo, cada aplicación podrá tener su visión particular, y seguramente parcial, del esquema conceptual. Los usuarios (programas o usuarios directos) verán la Base de Datos mediante esquemas externos apropiados a sus necesidades. Estos esquemas se pueden considerar redefiniciones del esquema conceptual, con las partes y los términos que convengan para las necesidades de las aplicaciones (o grupos de aplicaciones). Algunos sistemas los denominan subesquemas.

Al definir un esquema externo, se citarán sólo aquellos atributos y aquellas entidades que interesen; los podremos redenominar, podremos definir datos derivados o redefinir una entidad para que las aplicaciones que utilizan este esquema externo crean que son dos, definir combinaciones de entidades para que parezcan una sola, etc.

EJEMPLO DE ESQUEMA EXTERNO

Imaginemos una Base de Datos que en el esquema conceptual tiene definida, entre muchas otras, una entidad alumno con los siguientes atributos: numatri, nombre, apellido, numDNI, direccion, fechanac, telefono. Sin embargo, nos puede interesar que unos determinados programas o usuarios vean la Base de Datos formada de acuerdo con un esquema externo que tenga definidas dos entidades, denominadas estudiante y persona.



- a) La entidad estudiante podría tener definido el atributo numero-matricula (definido como derivable directamente de numatri), el atributo nombre-pila (de nombre), el atributo apellido y el atributo DNI (de numDNI).
- b) La entidad persona podría tener el atributo DNI (obtenido de numDNI), el atributo nombre (formado por la concatenación de nombre y apellido), el atributo dirección y el atributo edad (que deriva dinámicamente de fechanac).

El esquema interno o físico contendrá la descripción de la organización física de la Base de Datos: caminos de acceso (índices, hashing, apuntadores, etc.), codificación de los datos, gestión del espacio, tamaño de la página, etc.

El esquema de nivel interno responde a las cuestiones de rendimiento (espacio y tiempo) planteadas al hacer el diseño físico de la Base de Datos y al ajustarlo⁶ posteriormente a las necesidades cambiantes.

De acuerdo con la arquitectura ANSI/SPARC, para crear una Base de Datos hace falta definir previamente su esquema conceptual, definir como mínimo un esquema externo y, de forma eventual, definir su esquema interno. Si este último esquema no se define, el mismo SGBD tendrá que decidir los detalles de la organización física. El SGBD se encargará de hacer las correspondencias (*mappings*) entre los tres niveles de esquemas.

ESQUEMAS Y NIVELES EN LOS SGBD RELACIONALES

En los SGBD relacionales (es decir, en el mundo de SQL) se utiliza una terminología ligeramente diferente. No se separan de forma clara tres niveles de descripción. Se habla de un solo esquema **–schema–**, pero en su interior se incluyen descripciones de los tres niveles. En el **schema** se describen los elementos de aquello que en la arquitectura ANSI/SPARC se denomina esquema conceptual (entidades tipo, atributos y restricciones) y las vistas **–view–**, que corresponden aproximadamente a los esquemas externos.

El modelo relacional en que está inspirado SQL se limita al mundo lógico. Por ello, el estándar ANSI-ISO de SQL no habla en absoluto del mundo físico o interno; lo deja en manos de los SGBD relacionales del mercado. Sin embargo, estos SGBD proporcionan la posibilidad de incluir dentro del schema descripciones de estructuras y características físicas (índice, tablespace, clúster, espacios para colisiones, etc.)

INDEPENDENCIA DE LOS DATOS

En este subapartado veremos cómo la arquitectura de tres niveles que acabamos de presentar nos proporciona los dos tipos de independencia de los datos: la física y la lógica.

Hay independencia física cuando los cambios en la organización física de la Base de Datos no afectan al mundo exterior (es decir, los programas usuarios o los usuarios directos).

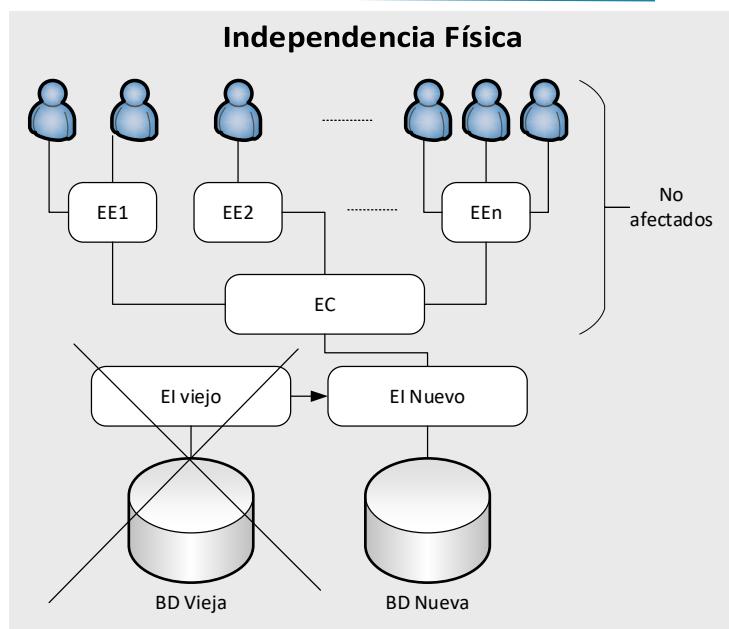
De acuerdo con la arquitectura ANSI/SPARC, habrá independencia física cuando los cambios en el esquema interno no afecten al esquema conceptual ni a los esquemas externos.

⁶ En inglés, el ajuste se conoce con el nombre de tuning.



Es obvio que cuando cambiemos unos datos de un soporte a otro, o los cambiemos de lugar dentro de un soporte, no se verán afectados ni los programas de aplicación ni los usuarios directos, ya que no se modificará el esquema conceptual ni el externo. Sin embargo, tampoco tendrían que verse afectados si cambiásemos, por ejemplo, el método de acceso a unos registros determinados⁷, el formato o la codificación, etc. Ninguno de estos casos debería afectar al mundo exterior, sino sólo a la Base de Datos física, el esquema interno, etc.

Si hay independencia física de los datos, lo único que variará al cambiar el esquema interno son las correspondencias entre el esquema conceptual y el interno. Obviamente, la mayoría de los cambios del esquema interno obligarán a rehacer la Base de Datos real (la física).



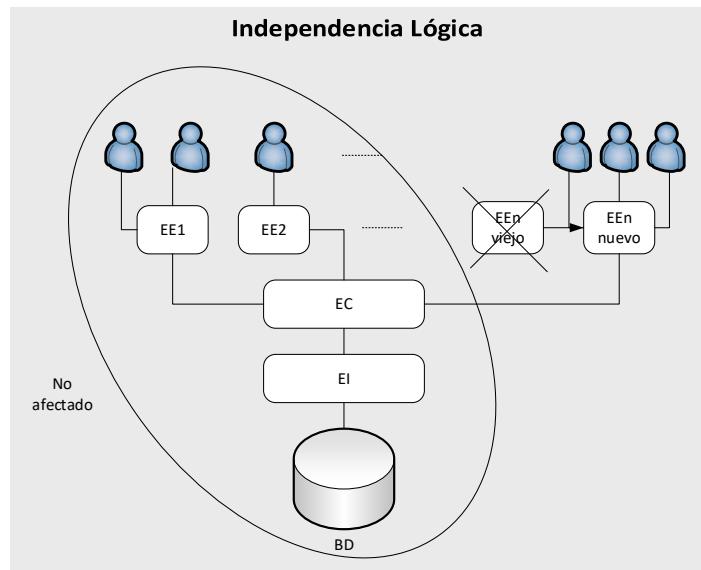
Hay independencia lógica cuando los usuarios⁸ no se ven afectados por los cambios en el nivel lógico.

Dados los dos niveles lógicos de la arquitectura ANSI/SPARC, diferenciaremos las dos situaciones siguientes:

- 1) Cambios en el esquema conceptual. Un cambio de este tipo no afectará a los esquemas externos que no hagan referencia a las entidades o a los atributos modificados.
 - 2) Cambios en los esquemas externos. Efectuar cambios en un esquema externo afectará a los usuarios que utilicen los elementos modificados. Sin embargo, no debería afectar a los demás usuarios ni al esquema conceptual, y tampoco, en consecuencia, al esquema interno y a la Base de Datos física.

USUARIOS NO AFECTADOS POR LOS CAMBIOS

Noten que no todos los cambios de elementos de un esquema externo afectarán a sus usuarios. Veamos un ejemplo de ello: antes hemos visto que cuando eliminábamos el atributo apellido del esquema conceptual, debíamos modificar el esquema externo donde definímos nombre, porque allí estaba definido como concatenación de nombre y apellido.



⁷ Por ejemplo, eliminando un índice en árbol-B o sustituyéndolo por un hashing.

⁸ Programas de aplicación o usuarios directos.



Pues bien, un programa que utilizase el atributo nombre no se vería afectado si modificásemos el esquema externo de modo que nombre fuese la concatenación de nombre y una cadena constante (por ejemplo, toda en blanco). Como resultado, habría desaparecido el apellido de nombre, sin que hubiera sido necesario modificar el programa.

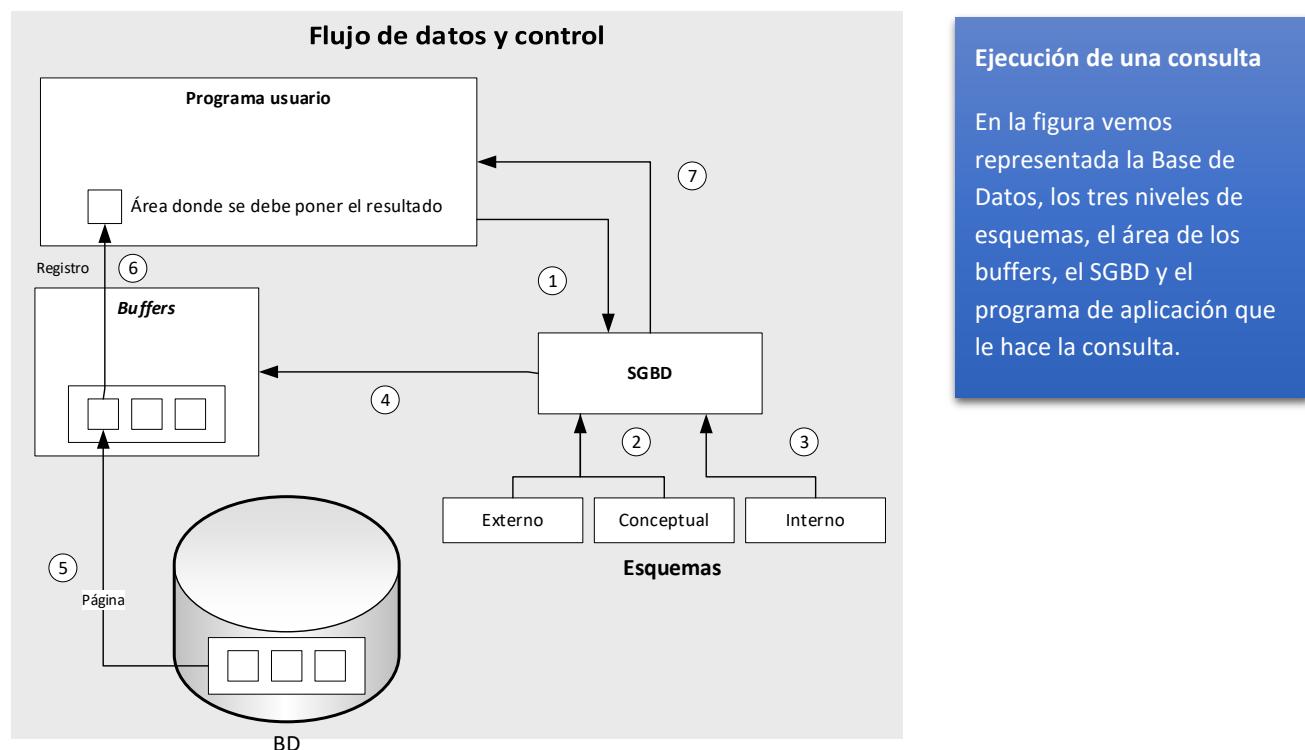
Los SGBD actuales proporcionan bastante independencia lógica, pero menos de la que haría falta, ya que las exigencias de cambios constantes en el Sistema de Información piden grados muy elevados de flexibilidad. Los sistemas de archivos tradicionales, en cambio, no ofrecen ninguna independencia lógica.

FLUJO DE DATOS Y DE CONTROL

Para entender el funcionamiento de un SGBD, a continuación, veremos los principales pasos de la ejecución de una consulta sometida al SGBD por un programa de aplicación. Explicaremos las líneas generales del flujo de datos y de control entre el SGBD, los programas de usuario y la Base de Datos.

Recuerden que el SGBD, con la ayuda del SO, lee páginas (bloques) de los soportes donde está almacenada la Base de Datos física, y las lleva a un área de buffers o memorias caché en la memoria principal. El SGBD pasa registros desde los buffers hacia el área de trabajo del mismo programa.

Supongamos que la consulta pide los datos del alumno que tiene un determinado DNI. Por lo tanto, la respuesta que el programa obtendrá será un solo registro y lo recibirá dentro de un área de trabajo propia⁹.



El proceso que se sigue es el siguiente:

- Empieza con una llamada ① del programa al SGBD, en la que se le envía la operación de consulta. El SGBD debe verificar que la sintaxis de la operación recibida sea correcta, que el usuario del programa esté autorizado

⁹ Por ejemplo, una variable con estructura de tupla.



- a hacerla, etc. Para poder llevar a cabo todo esto, el SGBD se basa ② en el esquema externo con el que trabaja el programa y en el esquema conceptual.
- b) Si la consulta es válida, el SGBD determina, consultando el esquema interno ③, qué mecanismo debe seguir para responderla. Ya sabemos que el programa usuario no dice nada respecto a cómo se debe hacer físicamente la consulta. Es el SGBD el que lo debe determinar. Casi siempre hay varias formas y diferentes caminos para responder a una consulta¹⁰. Supongamos que ha elegido aplicar un hashing al valor del DNI, que es el parámetro de la consulta, y el resultado es la dirección de la página donde se encuentra (entre muchos otros) el registro del alumno buscado.
 - c) Cuando ya se sabe cuál es la página, el SGBD comprobará ④ si por suerte esta página ya se encuentra en aquel momento en el área de los buffers (tal vez como resultado de una consulta anterior de este usuario o de otro). Si no está, el SGBD, con la ayuda del SO, la busca en disco y la carga en los buffers ⑤. Si ya está, se ahorra el acceso a disco.
 - d) Ahora, la página deseada ya está en la memoria principal. El SGBD extrae, de entre los distintos registros que la página puede contener, el registro buscado, e interpreta la codificación y el resultado según lo que diga el esquema interno.
 - e) El SGBD aplica a los datos las eventuales transformaciones lógicas que implica el esquema externo (tal vez cortando la dirección por la derecha) y las lleva al área de trabajo del programa ⑥.
 - f) A continuación, el SGBD retorna el control al programa ⑦ y da por terminada la ejecución de la consulta.

Diferencias entre SGBD

Aunque entre diferentes SGBD puede haber enormes diferencias de funcionamiento, suelen seguir el esquema general que acabamos de explicar.

MODELOS DE BASES DE DATOS

Una Base de Datos es una representación de la realidad (de la parte de la realidad que nos interesa en nuestro Sistema de Información). Dicho de otro modo, una Base de Datos se puede considerar un modelo de la realidad. El componente fundamental utilizado para modelar en un SGBD relacional son las tablas (denominadas relaciones en el mundo teórico). Sin embargo, en otros tipos de SGBD se utilizan otros componentes.

El conjunto de componentes o herramientas conceptuales que un SGBD proporciona para modelar recibe el nombre de modelo de Base de Datos. Los cuatro modelos de Base de Datos más utilizados en los Sistema de Información son el modelo relacional, el modelo jerárquico, el modelo en red y el modelo relacional con objetos.

Todo modelo de Base de Datos nos proporciona tres tipos de herramientas:

- a) Estructuras de datos con las que se puede construir la Base de Datos: tablas, árboles, etc.

Las tablas o relaciones se estudiarán en la unidad didáctica “El modelo relacional y el álgebra relacional” de este curso.

¡Cuidado con las confusiones!

Popularmente, en especial en el campo de la informática personal, se denomina Bases de Datos a lo que aquí denominamos SGBD. Tampoco se debe confundir la Base de Datos considerada como modelo de la realidad con lo que aquí denominamos modelo de Base de Datos. El modelo de Base de Datos es el conjunto de herramientas conceptuales (piezas) que se utilizan para construir el modelo de la realidad.

¹⁰ Por ejemplo, siempre tiene la posibilidad de hacer una búsqueda secuencial.



- b) Diferentes tipos de restricciones (o reglas) de integridad que el SGBD tendrá que hacer cumplir a los datos: dominios, claves, etc.
- c) Una serie de operaciones para trabajar con los datos. Un ejemplo de ello, en el modelo relacional, es la operación SELECT, que sirve para seleccionar (o leer) las filas que cumplen alguna condición. Un ejemplo de operación típica del modelo jerárquico y del modelo en red podría ser la que nos dice si un determinado registro tiene "hijos" o no.

EVOLUCIÓN DE LOS MODELOS DE BASE DE DATOS

De los cuatro modelos de Base de Datos que hemos citado, el que apareció primero, a principios de los años sesenta, fue el modelo jerárquico. Sus estructuras son registros interrelacionados en forma de árboles. El SGBD clásico de este modelo es el IMS/DL1 de IBM.

El modelo relacional se estudia con detalle en la unidad “El modelo relacional y el álgebra relacional” de esta materia.

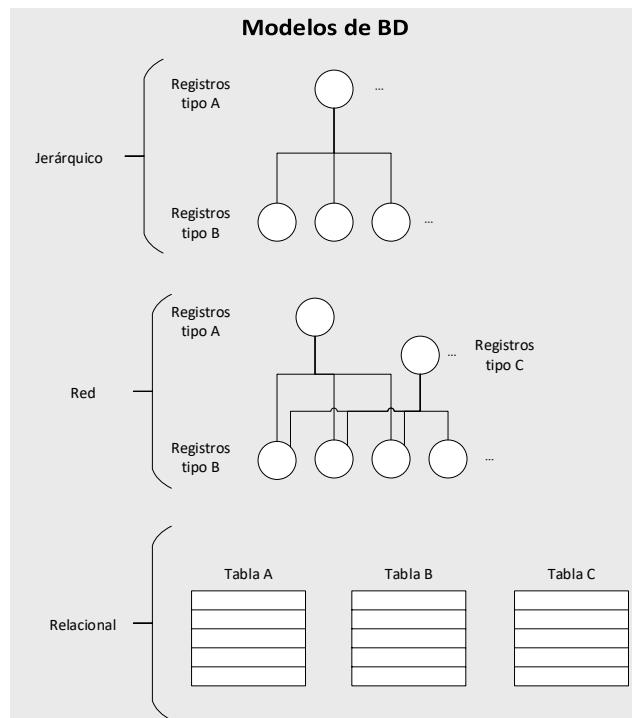
A principios de los setenta surgieron SGBD basados en un modelo en red. Como en el modelo jerárquico, hay registros e interrelaciones, pero un registro ya no está limitado a ser “hijo” de un solo registro tipo. El comité CODASYL- DBTG propuso un estándar basado en este modelo, que fue adoptado por muchos constructores de SGBD¹¹. Sin embargo, encontró la oposición de IBM, la empresa entonces dominante. La propuesta de CODASYL-DBTG ya definía tres niveles de esquemas.

Durante los años ochenta apareció una gran cantidad de SGBD basados en el modelo relacional propuesto en 1969 por E.F. Codd, de IBM, y prácticamente todos utilizaban como lenguaje nativo el SQL¹². El modelo relacional se basa en el concepto matemático de relación, que aquí podemos considerar de momento equivalente al término tabla (formada por filas y columnas). La mayor parte de los Sistema de Información que actualmente están en funcionamiento utilizan SGBD relacionales, pero algunos siguen utilizando los jerárquicos o en red (especialmente en Sistema de Información antiguos muy grandes).

Así como en los modelos pre-relacionales (jerárquico y en red), las estructuras de datos constan de dos elementos básicos (los registros y las interrelaciones), en el modelo relacional constan de un solo elemento: la tabla, formada por filas y columnas. Las interrelaciones se deben modelizar utilizando las tablas.

Otra diferencia importante entre los modelos pre-relacionales y el modelo relacional es que el modelo relacional se limita al nivel lógico (no hace absolutamente ninguna consideración sobre las representaciones físicas). Es decir, nos da una independencia física de datos total. Esto es así si hablamos del modelo teórico, pero los SGBD del mercado nos proporcionan una independencia limitada.

Estos últimos años se está extendiendo el modelo de Base de Datos relacional con objetos. Se trata de ampliar el modelo relacional, añadiéndole la posibilidad de que los tipos de



¹¹ Por ejemplo, IDS de Bull, DMS de Univac y DBMS de Digital.

¹² Por ejemplo, Oracle, DB2 de IBM, Informix, Ingres, Allbase de HP y SQL-Server de Sybase.



datos sean tipos abstractos de datos, TAD. Esto acerca los sistemas relacionales al paradigma de la OO. Los primeros SGBD relacionales que dieron esta posibilidad fueron Oracle (versión 8), Informix (versión 9) e IBM/DB2/UDB (versión 5).

Hablamos de modelos de Base de Datos, pero de hecho se acostumbran a denominar modelos de datos, ya que permiten modelarlos. Sin embargo, hay modelos de datos que no son utilizados por los SGBD del mercado: sólo se usan durante el proceso de análisis y diseño, pero no en las realizaciones.

Los más conocidos de estos tipos de modelos son los modelos semánticos y los funcionales. Éstos nos proporcionan herramientas muy potentes para describir las estructuras de la información del mundo real, la semántica y las interrelaciones, pero normalmente no disponen de operaciones para tratarlas. Se limitan a ser herramientas de descripción lógica. Son muy utilizados en la etapa del diseño de Base de Datos y en herramientas CASE. El más extendido de estos modelos es el conocido como modelo ER (entity-relationship), que estudiaremos más adelante.

La evolución de los modelos...

... a lo largo de los años los ha ido alejando del mundo físico y los ha acercado al mundo lógico; es decir, se han alejado de las máquinas y se han acercado a las personas.

Actualmente, la práctica más extendida en el mundo profesional de los desarrolladores de Sistema de Información es la utilización del modelo ER durante el análisis y las primeras etapas del diseño de los datos, y la utilización del modelo relacional para acabar el diseño y construir la Base de Datos con un SGBD.

En esta asignatura hablamos sólo de Base de Datos con modelos de datos estructurados, que son los que normalmente se utilizan en los Sistema de Información empresariales. Sin embargo, hay SGBD especializados en tipos de aplicaciones concretas que no siguen ninguno de estos modelos. Por ejemplo, los SGBD documentales o los de Base de Datos geográficas.

LENGUAJES Y USUARIOS

Para comunicarse con el SGBD, el usuario, ya sea un programa de aplicación o un usuario directo, se vale de un lenguaje. Hay muchos lenguajes diferentes, según el tipo de usuarios para los que están pensados y el tipo de cosas que los usuarios deben poder expresar con ellos:

- a) Habrá usuarios informáticos muy expertos que querrán escribir procesos complejos y que necesitarán lenguajes complejos.
- b) Sin embargo, habrá usuarios finales no informáticos, ocasionales (esporádicos), que sólo harán consultas. Estos usuarios necesitarán un lenguaje muy sencillo, aunque dé un rendimiento bajo en tiempo de respuesta.
- c) También podrá haber usuarios finales no informáticos, dedicados o especializados. Son usuarios cotidianos o, incluso, dedicados exclusivamente a trabajar con la Base de Datos¹³. Estos usuarios necesitarán lenguajes muy eficientes y compactos, aunque no sea fácil aprenderlos. Tal vez serán lenguajes especializados en tipos concretos de tareas.

¿Qué debería poder decir el usuario al SGBD?

Por un lado, la persona que hace el diseño debe tener la posibilidad de describir al SGBD la Base de Datos que ha diseñado. Por otro lado, debe ser posible pedirle al sistema que rellene y actualice la base de datos con los datos que se le den. Además, y obviamente, el usuario debe disponer de medios para hacerle consultas

¹³ Por ejemplo, personas dedicadas a introducir datos masivamente.



Hay lenguajes especializados en la escritura de esquemas; es decir, en la descripción de la Base de Datos. Se conocen genéricamente como DDL o data definition language. Incluso hay lenguajes específicos para esquemas internos, lenguajes para esquemas conceptuales y lenguajes para esquemas externos.

Otros lenguajes están especializados en la utilización de la Base de Datos (consultas y mantenimiento). Se conocen como DML o data management language. Sin embargo, lo más frecuente es que el mismo lenguaje disponga de construcciones para las dos funciones, DDL y DML.

El lenguaje SQL, que es el más utilizado en las Bases de Datos relacionales, tiene verbos –instrucciones– de tres tipos diferentes:

- 1) Verbos del tipo DML; por ejemplo, SELECT para hacer consultas, e INSERT, UPDATE y DELETE para hacer el mantenimiento de los datos.
- 2) Verbos del tipo DDL; por ejemplo, CREATE TABLE para definir las tablas, sus columnas y las restricciones.
- 3) Además, SQL tiene verbos de control del entorno, como por ejemplo COMMIT y ROLLBACK para delimitar transacciones.

El lenguaje SQL se explicará en la unidad didáctica “El lenguaje SQL” de esta materia.

En cuanto a los aspectos DML, podemos diferenciar dos tipos de lenguajes:

- a) Lenguajes muy declarativos (o implícitos), con los que se especifica qué se quiere hacer sin explicar cómo se debe hacer.
- b) Lenguajes más explícitos o procedimentales, que nos exigen conocer más cuestiones del funcionamiento del SGBD para detallar paso a paso cómo se deben realizar las operaciones (lo que se denomina navegar por la Base de Datos).

Lenguajes declarativos y procedimentales

El aprendizaje y la utilización de los lenguajes procedimentales acostumbran a ser más difíciles que los declarativos, y por ello sólo los utilizan usuarios informáticos. Con los procedimentales se pueden escribir procesos más eficientes que con los declarativos.

Como es obvio, los aspectos DDL (las descripciones de los datos) son siempre declarativos por su propia naturaleza.

Los lenguajes utilizados en los SGBD pre relacionales eran procedimentales. SQL es básicamente declarativo, pero tiene posibilidades procedimentales.

Aunque casi todos los SGBD del mercado tienen SQL como lenguaje nativo, ofrecen otras posibilidades, como por ejemplo 4GL y herramientas visuales:

Lenguajes 4GL (4th Generation Languages)¹⁴ de muy alto nivel, que suelen combinar elementos procedimentales con elementos declarativos. Pretenden facilitar no sólo el tratamiento de la Base de Datos, sino también la definición de menús, pantallas y diálogos.

¹⁴ Empezaron a aparecer al final de los años ochenta.



Herramientas o interfaces visuales¹⁵ muy fáciles de utilizar, que permiten usar las Bases de Datos siguiendo el estilo de diálogos con ventanas, iconos y ratón, puesto de moda por las aplicaciones Windows. No sólo son útiles a los usuarios no informáticos, sino que facilitan mucho el trabajo a los usuarios informáticos: permiten consultar y actualizar la Base de Datos, así como definirla y actualizar su definición con mucha facilidad y claridad.

Tanto los 4GL como las herramientas visuales (con frecuencia unidas en una sola herramienta) traducen lo que hace el usuario a instrucciones SQL por distintas vías:

- En el caso de los 4GL, la traducción se suele hacer mediante la compilación.
- En el caso de las herramientas visuales, se efectúa por medio del intérprete de SQL integrado en el SGBD.

Si queremos escribir un programa de aplicación que trabaje con Bases de Datos, seguramente querremos utilizar nuestro lenguaje habitual de programación¹⁶. Sin embargo, generalmente estos lenguajes no tienen instrucciones para realizar el acceso a las Bases de Datos. Entonces tenemos las dos opciones siguientes:

- 1) Las llamadas a funciones: en el mercado hay librerías de funciones especializadas en Bases de Datos (por ejemplo, las librerías ODBC). Sólo es preciso incluir llamadas a las funciones deseadas dentro del programa escrito con el lenguaje habitual. Las funciones serán las que se encargarán de enviar las instrucciones (generalmente en SQL) en tiempo de ejecución al SGBD.
- 2) El lenguaje hospedado: otra posibilidad consiste en incluir directamente las instrucciones del lenguaje de Bases de Datos en nuestro programa. Sin embargo, esto exige utilizar un precompilador especializado que acepte en nuestro lenguaje de programación habitual las instrucciones del lenguaje de Bases de Datos. Entonces se dice que este lenguaje (casi siempre SQL) es el lenguaje hospedado o incorporado (embedded), y nuestro lenguaje de programación (Pascal, C, Cobol, etc.) es el lenguaje anfitrión (host).

ADMINISTRACIÓN DE BASES DE DATOS

Hay un tipo de usuario especial: el que realiza tareas de administración y control de la Base de Datos. Una empresa o institución que tenga Sistema de Información construidos en torno a Bases de Datos necesita que alguien lleve a cabo una serie de funciones centralizadas de gestión y administración, para asegurar que la explotación de la Base de Datos es la correcta. Este conjunto de funciones se conoce con el nombre de administración de Bases de Datos (ABD), y los usuarios que hacen este tipo especial de trabajo se denominan administradores de Bases de Datos.

Los administradores de Bases de Datos son los responsables del correcto funcionamiento de la Base de Datos y velan por que siempre se mantenga útil. Intervienen en situaciones problemáticas o de emergencia, pero su responsabilidad fundamental es velar por que no se produzcan incidentes.

A continuación, damos una lista de tareas típicas del ABD:

- 1) Mantenimiento, administración y control de los esquemas. Comunicación de los cambios a los usuarios.

¹⁵ Han proliferado en los años noventa.

¹⁶ Pascal, C, Cobol, PL/I, Basic, MUMPS, Fortran, Java, etc.



-
- 2) Asegurar la máxima disponibilidad de los datos; por ejemplo, haciendo copias (back-ups), administrando diarios (journals o logs), reconstruyendo la Base de Datos, etc.
 - 3) Resolución de emergencias.
 - 4) Vigilancia de la integridad y de la calidad de los datos.
 - 5) Diseño físico, estrategia de caminos de acceso y reestructuraciones.
 - 6) Control del rendimiento y decisiones relativas a las modificaciones en los esquemas y/o en los parámetros del SGBD y del SO, para mejorarlo.
 - 7) Normativa y asesoramiento a los programadores y a los usuarios finales sobre la utilización de la Base de Datos.
 - 8) Control y administración de la seguridad: autorizaciones, restricciones, etc.
-

LA TAREA DEL ABD NO ES SENCILLA.

Los SGBD del mercado procuran reducir al mínimo el volumen de estas tareas, pero en sistemas muy grandes y críticos se llega a tener grupos de ABD de más de diez personas. Buena parte del software que acompaña el SGBD está orientado a facilitar la gran diversidad de tareas controladas por el ABD: monitores del rendimiento, monitores de la seguridad, verificadores de la consistencia entre índices y datos, reorganizadores, gestores de las copias de seguridad, etc. La mayoría de estas herramientas tienen interfaces visuales para facilitar la tarea del ABD.



RESUMEN

En esta unidad hemos hecho una introducción a los conceptos fundamentales del mundo de las Bases de Datos y de los SGBD. Hemos explicado la evolución de los SGBD, que ha conducido de una estructura centralizada y poco flexible a una distribuida y flexible, y de una utilización procedimental que requería muchos conocimientos a un uso declarativo y sencillo.

Hemos revisado los objetivos de los SGBD actuales y algunos de los servicios que nos dan para conseguirlos. Es especialmente importante el concepto de transacción y la forma en que se utiliza para velar por la integridad de los datos.

La arquitectura de tres niveles aporta una gran flexibilidad a los cambios, tanto a los físicos como a los lógicos. Hemos visto cómo un SGBD puede funcionar utilizando los tres esquemas propios de esta arquitectura.

Hemos explicado que los componentes de un modelo de Bases de Datos son las estructuras, las restricciones y las operaciones. Los diferentes modelos de Bases de Datos se diferencian básicamente por sus estructuras. Hemos hablado de los modelos más conocidos, especialmente del modelo relacional, que está basado en tablas y que estudiaremos más adelante.

El modelo relacional se estudiará en la unidad “El modelo relacional y el álgebra relacional” de este curso.

Cada tipo de usuario del SGBD puede utilizar un lenguaje apropiado para su trabajo. Unos usuarios con una tarea importante y difícil son los administradores de las Bases de Datos.



UNIDAD 02

MODELO ENTIDAD RELACIÓN

El modelo de datos entidad-relación (E-R) está basado en una percepción del mundo real consistente en objetos básicos llamados entidades y de relaciones entre estos objetos.

Se desarrolló para facilitar el diseño de bases de datos permitiendo la especificación de un esquema de la empresa que representa la estructura lógica completa de una base de datos.

El modelo de datos E-R es uno de los diferentes modelos de datos semánticos; el aspecto semántico del modelo yace en la representación del significado de los datos.

El modelo E-R es extremadamente útil para hacer corresponder los significados e interacciones de las empresas del mundo real con un esquema conceptual. El modelo de datos entidad-relación (E-R) está basado en una percepción del mundo real consistente en objetos básicos llamados entidades y de relaciones entre estos objetos.

Se desarrolló para facilitar el diseño de bases de datos permitiendo la especificación de un esquema de la empresa que representa la estructura lógica completa de una base de datos.

El modelo de datos E-R es uno de los diferentes modelos de datos semánticos; el aspecto semántico del modelo yace en la representación del significado de los datos.

El modelo E-R es extremadamente útil para hacer corresponder los significados e interacciones de las empresas del mundo real con un esquema conceptual.

OBJETOS BÁSICOS DEL MODELO E-R

Los conceptos básicos previstos por el modelo ER son entidades, relaciones y atributos.

ENTIDADES Y CONJUNTO DE ENTIDADES

Una **entidad** es un objeto que existe y puede distinguirse de otros objetos. La entidad puede ser concreta, por ejemplo: una persona o un libro; o abstracta, por ejemplo, un día festivo o un concepto.

Un **conjunto de entidades** es un grupo de entidades del mismo tipo. El conjunto de todas las personas que tienen una cuenta en el banco, por ejemplo, puede definirse como el conjunto de entidades clientes.

Una entidad está representada por un conjunto de **atributos**. Los posibles atributos del conjunto de entidades clientes son nombre, documento, calle y ciudad. Para cada atributo existe un rango de valores permitidos, llamado **dominio** del atributo. El dominio del atributo nombre podría ser el conjunto de todos los nombres de personas de cierta longitud.

RELACIONES Y CONJUNTO DE RELACIONES

Una **relación** es una asociación entre varias entidades. Por ejemplo, es posible definir una relación que asocia al cliente Gutiérrez con la cuenta 401.



Un **conjunto de relaciones** es un grupo de relaciones del mismo tipo. Se definirá el conjunto de relaciones *clientecuenta* para denotar la asociación entre los clientes y las cuentas bancarias que tienen.

La relación *clientecuenta* es un ejemplo de una **relación binaria**, es decir, una que implica a dos conjuntos de entidades.

Existen conjuntos de relaciones que incluyen a n-conjuntos de entidades, **relaciones n-arias**, por ejemplo, la relación ternaria *cliecuentasuc* que especifica que el cliente Gutiérrez tienen la cuenta 401 en la sucursal Córdoba.

Las **relaciones recursivas (Reflexivas)** son relaciones binarias que conectan una entidad consigo misma.

Una relación también puede tener **atributos descriptivos o rótulos**. Por ejemplo, fecha podría ser un atributo del conjunto de relaciones *clientecuenta*. Esto especifica la última fecha en que el cliente tuvo acceso a su cuenta.

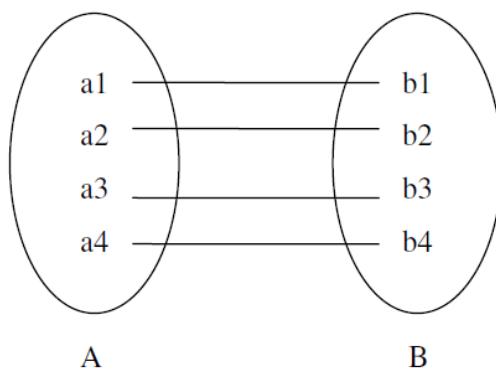
CARDINALIDADES DE MAPEO

Un esquema ER empresarial puede definir ciertas limitantes con las que deben cumplir los datos contenidos en la base de datos. Una limitante importante es la de las **cardinalidades de mapeo** que expresan el número de entidades con las que puede asociarse otra entidad mediante una relación.

Las cardinalidades de mapeo son más útiles al describir conjuntos binarios de relaciones, aunque también son aplicables a conjuntos n-arios de relaciones.

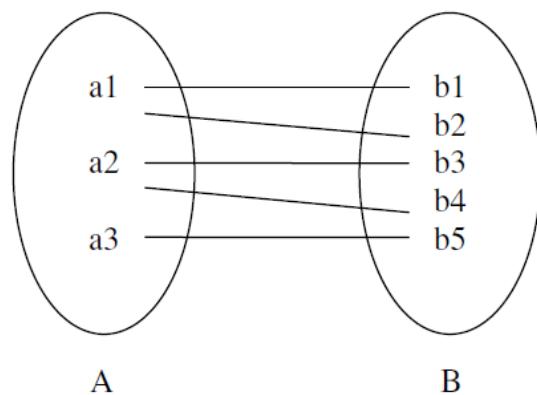
Para un conjunto binario de relaciones R entre los conjuntos de entidades A y B, la cardinalidad de mapeo puede ser:

Una a una: una entidad de A está asociada únicamente con una entidad de B y una entidad de B está asociada solo con una entidad de A.

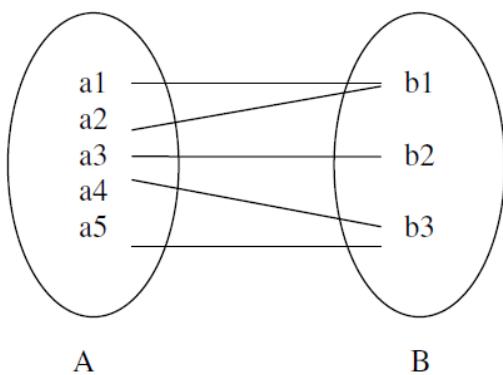


Una a muchas: una entidad en A está asociada con varias entidades de B, pero una entidad de B puede asociarse únicamente con una entidad de A.

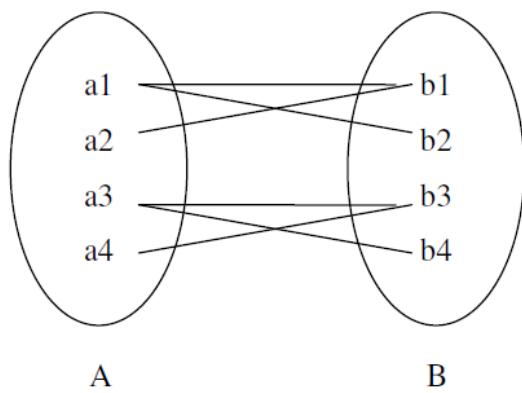




Muchas a una: una entidad de A está asociada únicamente con una entidad en B, pero una entidad de B está relacionada con varias entidades de A.



Muchas a muchas: una entidad en A está asociada con varias entidades de B y una entidad en B está vinculada con varias entidades de A.



Para ilustrar lo anterior, considérese el conjunto de relaciones *clientecuenta*. Si en un banco dado una cuenta puede pertenecer únicamente a un cliente y un cliente puede tener varias cuentas, entonces el conjunto de relaciones *clientecuenta* es una a muchas, de cliente a cuenta. Si una cuenta puede pertenecer a varios clientes, entonces el conjunto de relaciones *clientecuenta* es una a muchas, de cuenta a cliente, entonces en definitiva el conjunto de relaciones *clientecuenta* es muchas a muchas.

Las **dependencias de existencia** constituyen otra clase importante de limitantes. Si la existencia de la entidad x depende de la existencia de la entidad y, entonces se dice que x es dependiente por existencia de y.



Funcionalmente esto quiere decir que, si se elimina y, también se eliminará x. Se dice que la entidad y en una entidad dominante y que x es una entidad subordinada. Por ejemplo, supongamos que tenemos los conjuntos de entidades cuenta y transacción. Se forma la relación *cuentatransac* entre estos dos conjuntos es decir que para una cuenta determinada pueden existir varias transacciones. Esta relación es una a muchas de cuenta a transacción. Cada entidad transacción debe estar relacionada con una entidad cuenta. Si se elimina una entidad cuenta, entonces deben eliminarse también todas las entidades transacción vinculadas con esa cuenta. Por lo contrario, pueden eliminarse entidades transacción de la base de datos sin afectar ninguna cuenta. Por lo tanto, el conjunto de entidades cuenta es dominante y transacción es subordinada en la relación *cuentatransac*.

CLAVES PRIMARIAS

Una tarea muy importante dentro de la modelación de bases de datos consiste en especificar cómo se van a distinguir las entidades y las relaciones. Conceptualmente, las entidades individuales y las relaciones son distintas entre sí, pero desde el punto de vista de una base de datos la diferencia entre ellas debe expresarse en términos de sus atributos. Para hacer estas distinciones, se asigna una **clave primaria** a cada conjunto de entidades, ésta es un conjunto de uno o más atributos que, juntos, permiten identificar de forma única a una entidad dentro del conjunto de entidades. Por ejemplo: el atributo documento del conjunto entidades cliente es suficiente para distinguir a una entidad cliente de otra, por lo tanto, puede ser la clave primaria de ese conjunto de entidades.

Es posible que un conjunto de entidades no tenga suficientes atributos para formar una clave primaria. Por ejemplo: el conjunto entidades transacción tiene tres atributos: numtransac, fecha e importe. Aunque cada entidad transacción es distinta, dos transacciones hechas en cuentas diferentes pueden tener el mismo número de transacción, entonces el conjunto entidades transacción no tienen una clave primaria. Una entidad de un conjunto de este tipo se denomina **entidad débil** y una entidad que puede tener una clave primaria recibe el nombre de **entidad fuerte**. El concepto de entidades fuertes y débiles está relacionado con el de "dependencia por existencia".

Un conjunto de entidades débiles no tiene una clave primaria sin embargo es preciso tener una forma de distinguir entre todas las entidades del conjunto, aquella que depende de una entidad fuerte de otro conjunto relacionado. El **discriminador** de un conjunto de entidades débiles es un conjunto de atributos que permite hacer esta distinción. Por lo tanto, para nuestro ejemplo el discriminador es numtransac ya que para cada cuenta estos números identifican de forma única cada una de las transacciones.

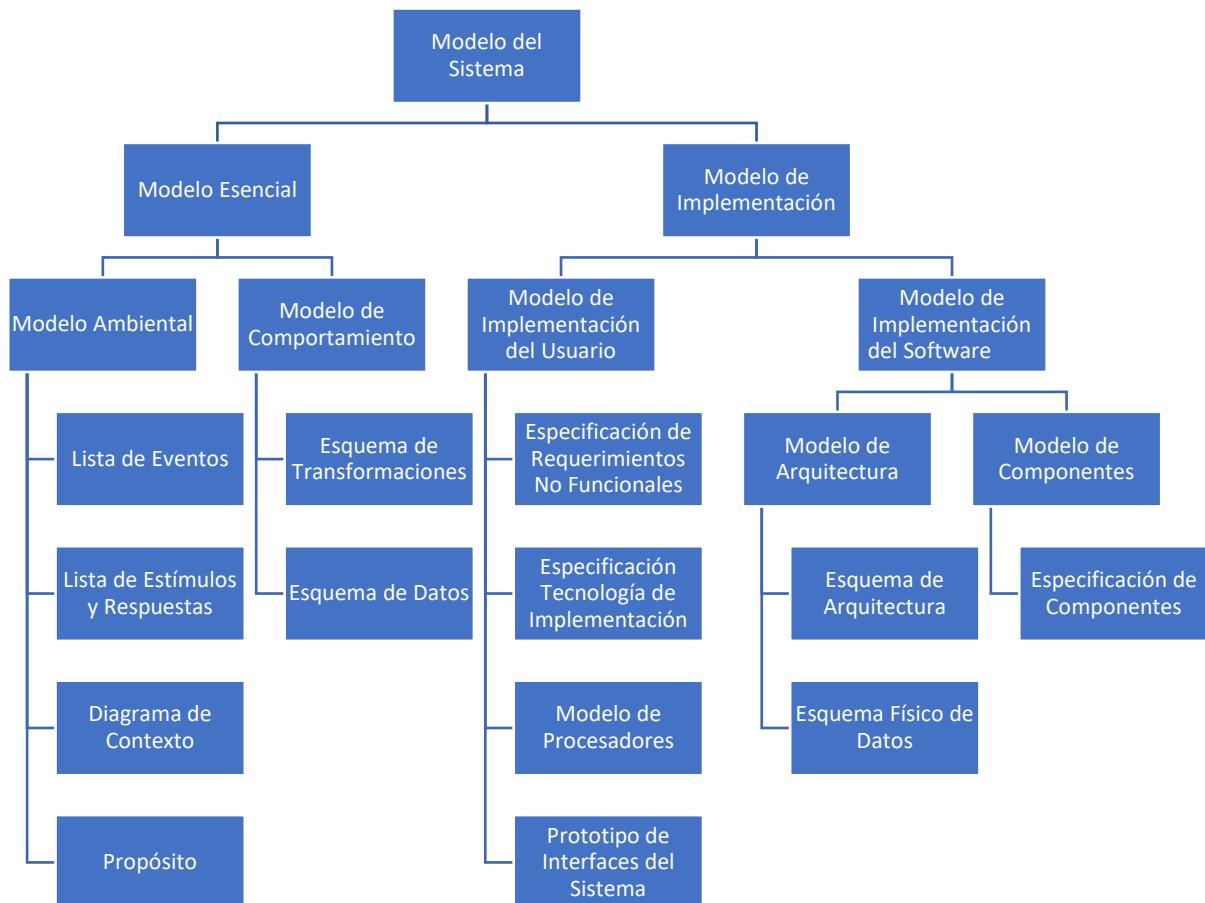
La clave primaria de un conjunto de entidades débiles está formada por la clave primaria de la entidad fuerte de la que depende su existencia y su discriminador. En el caso del conjunto de entidades transacción, su clave primaria es (numcuenta, numtransac), donde numcuenta identifica a la entidad dominante de una transacción y numtransac distingue a las entidades transacción dentro de la misma cuenta.

Los conjuntos de relaciones también tienen claves primarias. Sus claves primarias se forman tomando todos los atributos que constituyen las claves primarias de los conjuntos de entidades que definen el conjunto de relaciones. Por ejemplo: documento es la clave primaria de cliente y numcuenta es la clave primaria de cuenta. Por lo tanto, la clave primaria del conjunto de relaciones clientecuenta es (documento, numcuenta).



METODOLOGÍA ESTRUCTURADA

Dentro de la metodología estructurada, el Modelo Entidad Relación se encuadra en el Esquema de Datos, y el Esquema de Datos es parte del Modelo de Comportamiento, que a su vez es parte del Modelo Esencial, como podemos ver en el siguiente cuadro.



ESQUEMA DE DATOS

OBJETIVO:

Descripción de los datos que el sistema debe conocer (recordar) para poder responder a los estímulos. Es una visión pasiva del sistema y necesita mostrar:

- Las relaciones entre los datos, que no pueden mostrarse en los almacenamientos del DFD, pues generan requerimientos falsos. Las especificaciones de procesos del DFD muestran todas las relaciones entre los objetos, mediante los accesos esenciales; esos accesos **pueden eliminarse** de la especificación, ya que serán expresados en este esquema.
- Detalle de los datos: Identificadores y atributos descriptivos.



HERRAMIENTAS:

Las herramientas usadas son:

- Diagrama de Entidad Relación (conocido como DER), con extensiones de súper/subtipo y tipos de objetos asociativos. A lo largo del texto hablaremos indistintamente de objetos y tipos de objetos, que serán equivalentes a las entidades del DER; de la misma manera, asumiremos que atributo y elemento de dato tienen el mismo significado.
- Diccionario de datos, con la siguiente composición:

$$\text{DD} = \{\text{Objeto}\} + \{\text{Objeto_Asociativo}\} + \{\text{Objeto_Débil}\} + \{\text{Super_Tipo}\} + \{\text{Relación}\} + \{\text{Estructura_de_Datos}\} + \{\text{Atributo}\}$$

TÉCNICAS:

CONSTRUCCIÓN DEL ESQUEMA DE DATOS

Esta técnica es el resultado de aplicar criterios vertidos por variados autores. Los resultados que se obtienen son mucho más "naturales" que los obtenidos con otras técnicas.

El aspecto datos es más estable que el aspecto funcional, en la mayoría de los sistemas; también es mucho más difícil *pensar* el esquema de datos. La mayor dificultad se presenta en establecer la estructura de los objetos, y las relaciones entre los mismos; muchas veces decimos que es posible afirmar que un DER está mal, pero no puede asegurarse que un DER esté completamente bien.

- 1) Identificar Objetos: Un objeto es la representación de una cosa de existencia real o artificial, que interesa al sistema. Puede ser algo tangible, un rol desempeñado por una persona u organización, un incidente o una interacción. Es importante destacar que no siempre habrá una correspondencia uno a uno entre los objetos del Esquema de Datos y los objetos del mundo real (a los que hemos anteriormente llamado "cosas"). Muchos objetos reales son complejos -es decir, poseen una estructura- y están formados por otros objetos. Pensemos, por ejemplo, en un auto con sus diferentes partes. Si nos interesa registrar información de las distintas componentes del automóvil, no quedará otra alternativa que representarlos como distintos objetos en el DER.
- 2) Individualizar identificadores únicos de objetos: Un identificador es un atributo que confirma la existencia de un objeto dado, y la identidad de las distintas instancias del mismo. Si no puede encontrarse un identificador, o el objeto posee una sola instancia, este elemento **NO ES UN OBJETO**. Un objeto será **débil** si no es identificable por sus propios atributos y que para poder serlo requiere de uno o más atributos externos.
- 3) Identificar relaciones entre objetos: Una relación es una asociación esencial de la memoria para permitir los accesos esenciales y mejorar la descripción de los objetos. Generalmente las relaciones se implementan a través de atributos referenciales (Si, por ejemplo, fuésemos a implementar este esquema en una base de datos relacional, las relaciones se implementarían a través de claves foráneas). Ya que aquí estamos en la esencia, NO DEBEN utilizarse tales atributos para representar asociaciones entre objetos (por ejemplo, no debería incluirse el atributo Código de Cliente en el objeto Factura - siempre y cuando exista el objeto Cliente-). Las asociaciones se representan únicamente a través de relaciones.



-
- 4) Clasificar relaciones: Según los conceptos de:
 - a. **Grado:** es la cantidad de objetos que intervienen en la relación (unaria, binaria, etc.);
 - b. **Conectividad:** mapa de la asociación entre objetos (1:1, 1:N, M:N).
 - c. **Condicionabilidad:** indica si la participación de cada uno de los objetos en la relación es obligatoria u opcional, es decir, si existen o no instancias de los objetos intervenientes en la relación.
 - 5) Identificar atributos (elementos de datos que el sistema maneja).
 - 6) Asignar atributos a objetos y relaciones, materializando así la relación intra-objeto, según ese atributo describa una característica del objeto, y/o dependa funcional y no transitivamente del identificador del objeto/relación.
 - 7) Identificar objetos asociativos. Un objeto asociativo es un elemento que se comporta como relación y como objeto al mismo tiempo. Para que exista una instancia del mismo, **deben existir instancias de todos los objetos que relaciona.**
 - 8) Identificar súper/sub-tipos, agrupando objetos que posean atributos comunes y alguna condición de diferenciación.
 - 9) Dibujar el DER, según la notación de la herramienta.
 - 10) Eliminar elementos redundantes o fuera del alcance del sistema.
 - 11) Generar una entrada en el Diccionario de Datos por cada elemento del DER.
 - 12) Validar aplicando:
 - a. **Normalización**, con las observaciones indicadas más adelante.
 - b. Técnica de preguntas y respuestas.
 - 13) Revisar el esquema con el resto del modelo.



DIAGRAMA DE ENTIDAD RELACIÓN (DER)

INTRODUCCIÓN

Un sistema puede ser descripto de distintas formas. El sistema no cambia; lo que estamos haciendo es verlo desde distintos puntos de vista, priorizando un aspecto diferente del sistema cada vez. Mediante el DER priorizamos el **aspecto datos**.

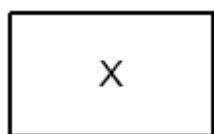
En el DFD hay caminos; éstos muestran el fluir de los datos, es el sistema en funcionamiento. Podemos ver que es lo que está pasando. En el DER también hay caminos, pero éstos representan las **relaciones** entre los distintos tipos de información que maneja el sistema; aquí no fluyen datos como en un DFD.

El DER representa los **datos almacenados** en un sistema presentado como una red de almacenamientos conectados por relaciones; es una vista estática, se ve al sistema como una entidad pasiva.

El DER de un sistema es más resistente al cambio que un DFD. Por ejemplo, en una organización, las políticas de crédito a un cliente pueden cambiar (funciones), pero la entidad *Cliente* sigue existiendo como tal. El DER depende del ambiente, y, por lo tanto, es menos factible que cambie.

CONVENCIONES

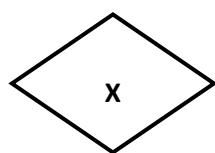
TIPO DE OBJETO (ENTIDAD):



Rectángulo

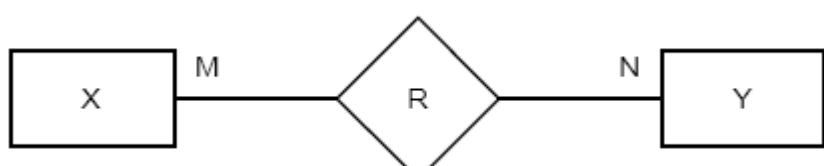
Es representado mediante un rectángulo con nombre. Agrupa bajo un mismo nombre un conjunto de cosas del mismo tipo pertenecientes al mundo real. Este objeto juega un rol significativo en el sistema a ser descripto, por ejemplo, *Clientes*, *Artículos*, etc.

RELACIÓN:



Rombo

Es representada mediante un rombo con nombre. Es el resultado de algún proceso del mundo real que vincula tipos de objetos que participan en ese proceso. Una relación es una abstracción porque ésta no describe el proceso, únicamente describe la forma en que las entidades se combinan. Sólo se define una relación entre dos o más tipos de objetos si dicha relación está basada en políticas, reglas o leyes que interesen al sistema que se modela.

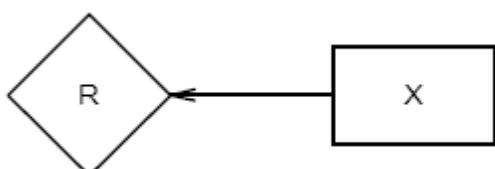


Las relaciones son típicamente multidireccionales, es decir, X está relacionado con Y y viceversa.

Se dice que X está relacionado con Y cuando desde una ocurrencia (**instancia**) de X puedo obtener la/s ocurrencia/s de Y que le corresponden.

El **orden** de la relación es **M:N**. Por ejemplo, si X es *Departamento*, Y es *Empleado*, y sabemos que un empleado trabaja en un solo departamento, y que en un departamento trabajan muchos empleados, el orden de la relación R (*Trabaja*) es **1:N**.

TIPO DE OBJETO ASOCIATIVO:

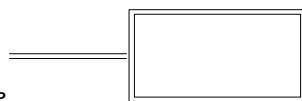


Rectángulo unido a un Rombo

Es representado mediante un rectángulo unido a un rombo. Surge cuando una relación contiene además información (Agregación / Asociación).

Este tipo de objeto es una relación que además posee atributos propios. Este tipo de objeto sólo existe mientras existan los objetos que relaciona.

OBJETOS O ENTIDADES DÉBILES:

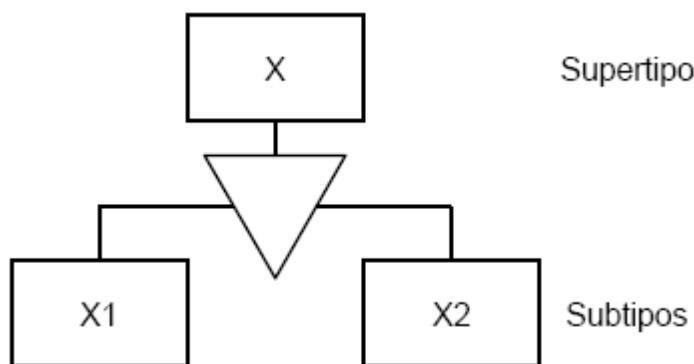


Rectángulo con línea doble

Es representado mediante un rectángulo con línea doble. La línea de relación hacia el objeto fuerte también se dibuja doble.

Un objeto que posee identificadores internos que determinan únicamente a cada una de sus instancias es un **OBJETO FUERTE**. Un objeto que deriva su existencia a partir del conjunto de atributos identificadores de otro u otros objetos es un **OBJETO DÉBIL**. Dichos atributos también se conocen como atributos externos.

SUPERTIPO Y SUBTIPOS



El **Supertipo** es representado como un rectángulo vinculado a sus subtipos por medio de un triángulo. Es el resultado de tratar una clase de objetos similares como un nuevo tipo de objeto (Generalización / Clasificación). (Ej.: *Vehículo* es un supertipo compuesto de *Auto* y *Camión*).

El **Subtipo** es una entidad. Es el resultado de tratar un subconjunto de algún tipo de objeto como un nuevo tipo de objeto (Diferenciación funcional).

Consideraciones Prácticas:

Para determinar si el modelo es completo, una buena forma de hacerlo es escribir todas las preguntas a las que debe responder el sistema y verificar si las relaciones del modelo permiten obtener las respuestas. Así como un DFD es erróneo cuando no produce una salida a partir de una entrada, un DER es erróneo si no puede responder a una pregunta de interés del sistema.

Hay que tener mucho cuidado con el nivel de detalle bajo el cual se agrupan los datos. No debe ser ni muy detallado ni muy general.

Lo importante es representar las interconexiones entre las lógicas principales dentro del área de interés del sistema.

- De ser muy detallado, nos damos cuenta cuando dos tipos de objetos están descriptos por los mismos atributos y tienen las mismas relaciones (Ej.: *Vendedores_Internos* y *Vendedores_Externos* que son tratados de la misma forma).
- De ser muy general nos damos cuenta cuando debemos identificar bajo qué condiciones son válidos determinados subconjuntos de atributos de un tipo de objeto. (Ej.: tener definido como entidad *Cajas_de_Ahorro* que incluye como subtipos *Caja_Ahorro_Común* y *Caja_Ahorro_Especial*, y cada una de éstas debe tener atributos específicos distintos).

Reglas de Conexión:

- Un tipo de objeto puede o no estar conectado y si lo está, puede ser a una o más relaciones.
- Una relación debe conectarse a uno o más Objetos.
- Un objeto no puede estar conectado directamente a otro.
- Una relación no puede estar conectada directamente a otra.

Reglas de Consistencia Interna:

- No puede haber tipos de objetos con el mismo nombre.
- Se debe tener un identificador único para cada tipo de objeto, relación, instancia de objeto e instancia de relación.
- No incluir relaciones irrelevantes para el sistema.
- Eliminar relaciones que no puedan existir en el mundo real.
- Eliminar relaciones que son redundantes.
- Un DER es consistente si puede proveer todos los datos requeridos en la aplicación de la técnica pregunta-respuesta.

Reglas de Claridad Semántica:

- No incluir atributos irrelevantes para el sistema.
- Todos los atributos de una entidad que son relevantes para el sistema deben ser incluidos.



-
- Si un objeto sólo tiene su identificación como atributo, quizás sea conveniente eliminarlo e incluir la información en otra Entidad.
 - Agrupar en súper / subtipos los objetos que dependen de relaciones idénticas.



MISCELÁNEAS

Es necesario incluir en el DD la información de todos los elementos del DER del sistema. A continuación, detallamos las convenciones a utilizar (que es un grupo de los tantos posibles).

Agregaremos un “@” a los atributos que sirvan como identificación al tipo de objeto (*Clave*), y “*ref*” a dichos atributos cuando figuran en una relación.

TIPO_DE_OBJETO	= Significado + @Identificador + {Atributos}
TIPO_DE_OBJETO_ASOCIATIVO	= Significado + @Identificador-Ref-N + @Identificador-Ref-M + {Atributos}
SUPERTIPO	= Significado + Atributos_Comunes + {Subtipos}
SUBTIPO	= * Idem Tipo de Objeto *
RELACIÓN	= Significado + [@Identificador-Ref-1 + @Identificador-Ref-1 * Caso 1:1 * @Identificador-Ref-1 + @Identificador-Ref-N * Caso 1:N * @Identificador-Ref-M + @Identificador-Ref-N * Caso M:N *]

CONVENCIONES

Notación para describir la composición de los elementos del DD:

=	... está compuesto por ...
+	... y ...
[]	o exclusivo.
{ }	repetición (en la llave izquierda desde; en la derecha, hasta).
()	opcional.
* *	comentario.

NORMALIZACIÓN EN LA CONSTRUCCIÓN DEL ESQUEMA DE DATOS.

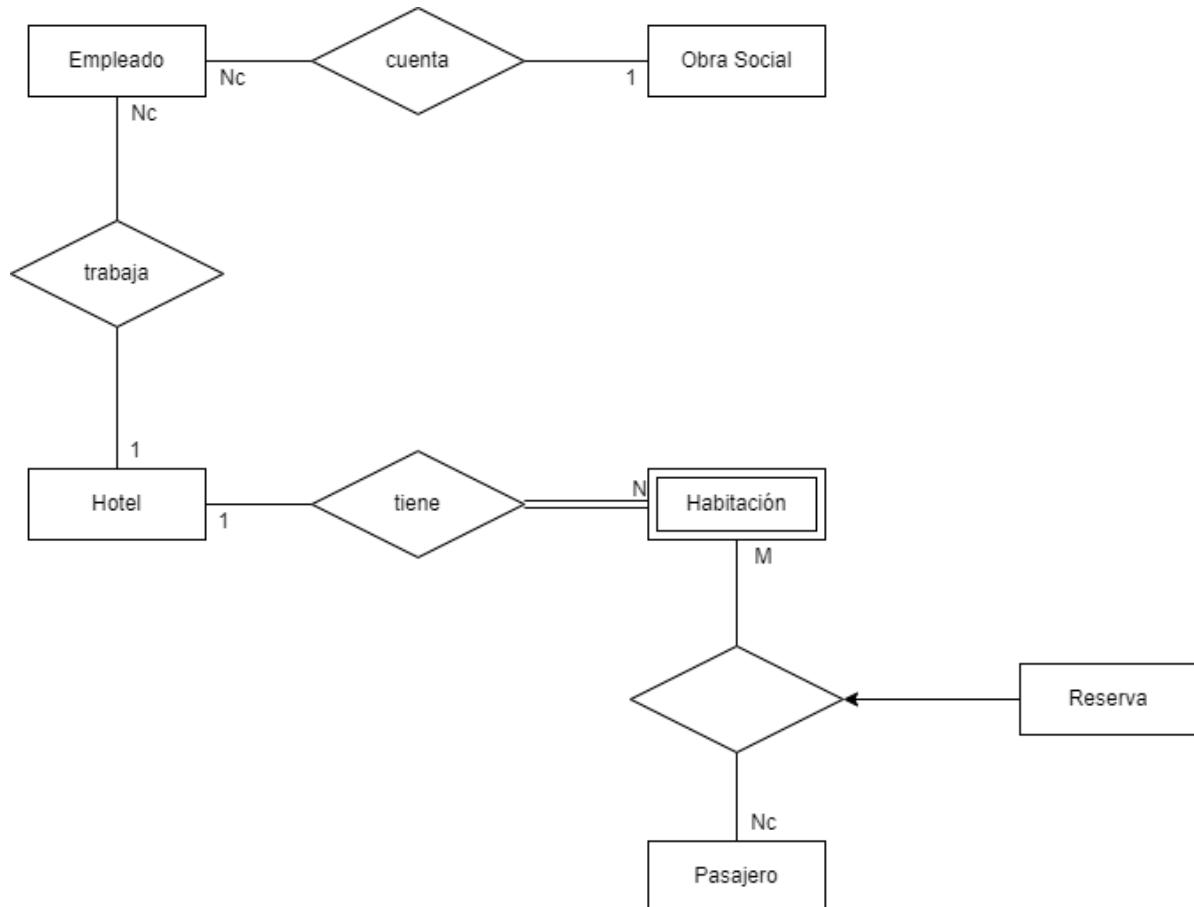
Para detectar objetos y para eliminar redundancias, podemos valernos de alguna de las técnicas de **normalización**. El problema consiste en que la mayoría de dichas técnicas tienen como objetivo construir un **esquema procesable**, introduciendo factores tecnológicos que, a este nivel, **están prohibidos**. Uno de estos factores es el generado por la primera forma normal, que impide la existencia de grupos repetitivos.

Para poder solucionar estos problemas, se sugiere lo siguiente:

- Utilizar una técnica de normalización que permita trabajar con relaciones anidadas (o sea, elementos repetitivos).
- Utilizar una técnica de normalización tradicional y luego volver a desnormalizar las relaciones anidadas.



En ambos casos se deberá testear con el usuario si los nuevos objetos o relaciones son válidos para el dominio de información del sistema. Además, se aplique o no una técnica de normalización, el concepto de **dependencia funcional** es muy útil para la construcción del Esquema.

EJEMPLO

Ilustración 1 Diagrama Entidad Relación
Tabla 1 Ejemplo de Diccionario de Datos

Empleado	=	@Legajo + Nombre + Sueldo
Habitación	=	@Identificador + Número + Ubicación + Capacidad + Categoría
Hotel	=	@Nombre + @Ciudad
Obra_Social	=	@Nombre + ImporteDesde + ImporteHasta
Pasajero	=	@ID + TipoDoc + NroDoc + Nombre + NroTarjetaCredito
Reserva	=	@FechaDesde + FechaHasta + NroPersonas + @Pasajero_ref_Nc + @Habitación_ref_M

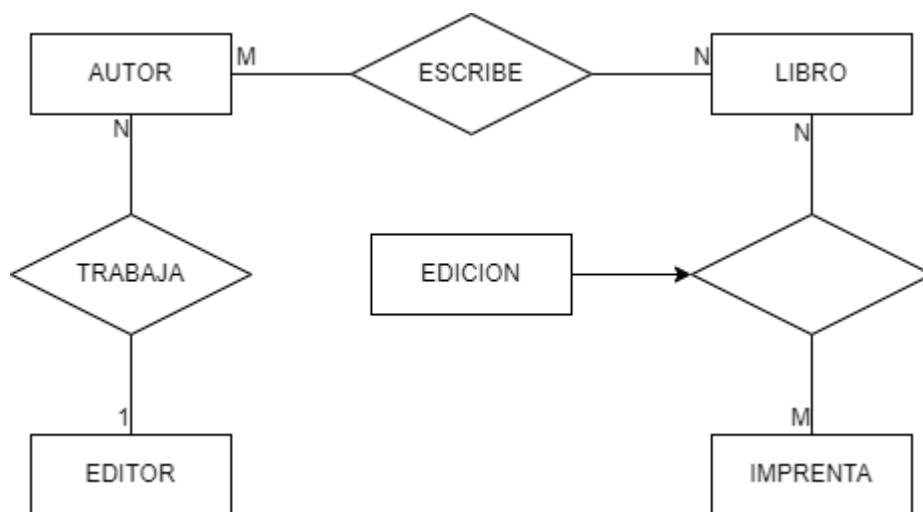


ANEXO A: TÉCNICA DE PREGUNTAS Y RESPUESTAS

La técnica de preguntas y respuestas consiste en verificar si las respuestas a una lista de preguntas propuestas por el usuario se encuentran contempladas en el esquema de datos construido.

EJEMPLO:

La editorial ABC trabaja con varios y diferentes autores quienes escriben los libros que serán publicados. Algunos autores escriben solo un libro, mientras los otros han escrito varios. La editorial también trabaja con varias imprentas. Distintas ediciones de un libro pueden ser realizadas por distintas imprentas. Se desea guardar la fecha de cada edición y la cantidad de ejemplares editados en la misma. Un editor de esta compañía trabaja con varios autores a la vez, editando y produciendo los libros; es el trabajo del editor preparar la última copia revisada pasada a máquina y lista para ser impresa.



DICCIONARIO DE DATOS

AUTOR	=	@id_autor + nombre + dirección + tel. + ...
EDICION	=	LIBRO-ref-N + IMPRENTA-ref-M + fecha edición + cant.ejemplares
EDITOR	=	@id_editor+ nombre + ...
IMPRENTA	=	@id_imprenta + dirección + tel + ...
LIBRO	=	@ISBN + título + género

Ahora se aplica la técnica de preguntas y respuestas. El usuario desea saber:

- 1) Qué libros editó cada editor en un determinado momento.
- 2)
 - a. Que libros escribió cierto autor,
 - b. en un determinado periodo
- 3)
 - a. Cuál es el autor de un determinado libro,

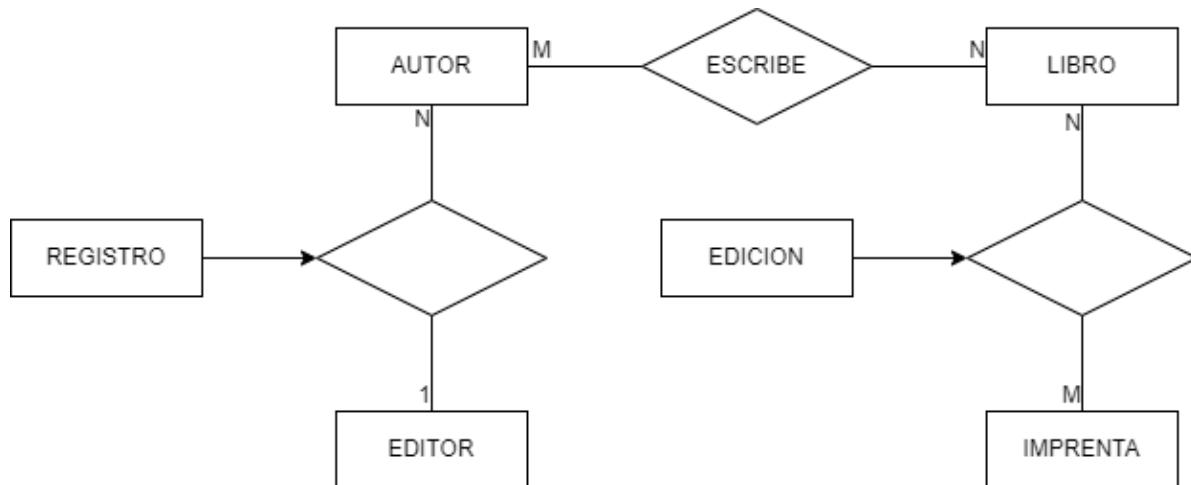


- b. Dónde y cuándo nació dicho autor.
- 4) En qué imprenta se imprimió cierto libro.
- 5)
- Cuántos ejemplares de un cierto libro se imprimieron en una imprenta.
 - en una cierta fecha o período.
- 6) Con qué editor trabaja cada autor en este momento.
- 7)
- Con qué editor trabajó cierto autor en una determinada fecha.
 - Cuántas veces trabajó con dicho autor y por cuánto tiempo.
- 8) Con qué imprentas se relaciona un editor, por los libros con los que está trabajando.
- 9) Cuál es el género en el que se especializa ahora cada editor.

En la forma en que está construido el esquema de datos, hay preguntas que no se pueden responder, por ejemplo:

- Para contestar la pregunta 2b, debería agregarse en el diccionario de datos de LIBRO, la fecha en la cual el mismo se realizó.
- Para contestar la pregunta 3b, debería agregarse en el diccionario de datos de AUTOR, lugar y fecha de nacimiento.
- Para contestar las preguntas 6, 7a, 7b, y 8, debería agregarse en un objeto asociativo entre EDITOR y AUTOR que contenga el identificador de cada uno de ellos más un conjunto repetitivo que incluye la fecha de comienzo del trabajo más la duración.
- Para contestar la pregunta 9, debería agregarse en el diccionario de datos de EDITOR, el o los géneros en que se especializa.

Luego de estas modificaciones se obtendría:



DICCIONARIO DE DATOS

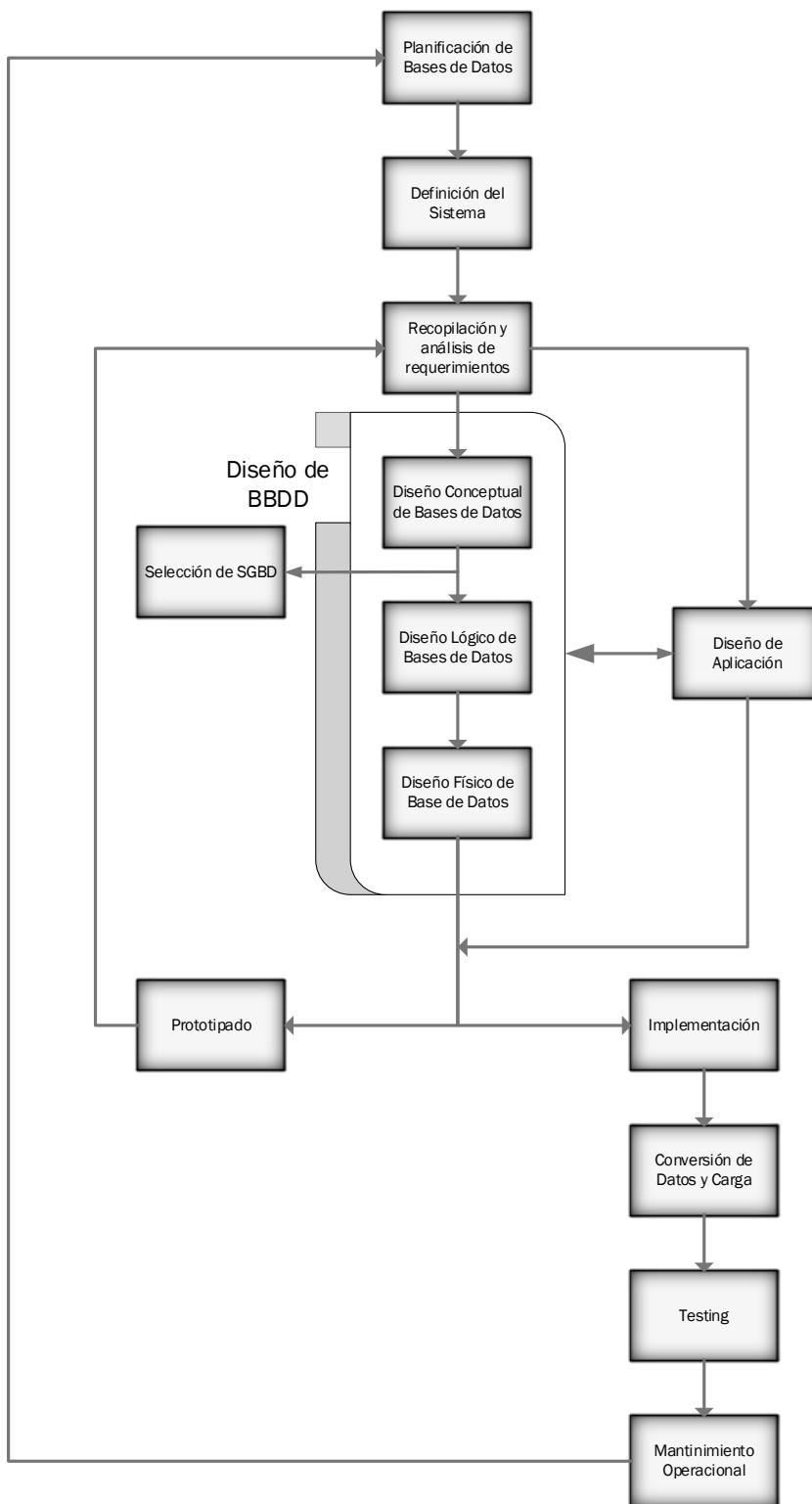
AUTOR	=	@id_autor + nombre + dirección + tel. + fecha_nac +lugar_nac
EDICION	=	LIBRO-ref-N + IMPRENTA-ref-M + fecha edición +cant.ejemplares
EDITOR	=	@id_editor+ nombre + {género}
IMPRENTA	=	@id_imprenta + dirección + tel + ...



LIBRO = @ISBN + título + género + fecha_escritura

REGISTRO = AUTOR-ref-M + EDITOR-ref-1 + fecha_comienzo_trabajo + duración



ANEXO B: CICLO DE VIDA CENTRADO EN LOS DATOS¹⁷


¹⁷ Database Systems - A Practical Approach to Design, Implementation, and Management – Chapter 10



UNIDAD 03

EL MODELO RELACIONAL

INTRODUCCIÓN

Esta unidad didáctica está dedicada al estudio del modelo de datos relacional.

El concepto de modelo de datos se ha presentado en otra unidad didáctica. En ésta se profundiza en un modelo de datos concreto: el modelo relacional, que actualmente tiene una gran relevancia. Sus conceptos fundamentales están bien asentados y, además, los sistemas de gestión de bases de datos relacionales son los más extendidos en su utilización práctica. Por estos motivos pensamos que es importante conocerlo.

El estudio del modelo relacional sirve, además, de base para los contenidos de otra unidad, dedicada al lenguaje SQL. Este lenguaje permite definir y manipular bases de datos relacionales. Los fundamentos del modelo relacional resultan imprescindibles para conseguir un buen dominio del SQL.

OBJETIVOS

En los materiales didácticos de esta unidad encontraremos las herramientas indispensables para alcanzar los siguientes objetivos:

1. Conocer los fundamentos del modelo de datos relacional.
2. Saber distinguir las características que debe tener un sistema de gestión de bases de datos relacional para que sea coherente con los fundamentos del modelo relacional.
3. Comprender las ventajas del modelo relacional que derivan del alto grado de independencia de los datos que proporciona, y de la simplicidad y la uniformidad del modelo.

INTRODUCCIÓN AL MODELO RELACIONAL

El modelo relacional es un modelo de datos y, como tal, tiene en cuenta los tres aspectos siguientes de los datos:

La estructura, que debe permitir representar la información que nos interesa del mundo real.

La manipulación, a la que da apoyo mediante las operaciones de actualización y consulta de los datos.

La integridad, que es facilitada mediante el establecimiento de reglas de integridad; es decir, condiciones que los datos deben cumplir.

Un sistema de gestión de bases de datos relacional (SGBDR) da apoyo a la definición de datos mediante la estructura de los datos del modelo relacional, así como a la manipulación de estos datos con las operaciones del modelo; además, asegura que se satisfacen las reglas de integridad que el modelo relacional establece.

Los principios del modelo de datos relacional fueron establecidos por E.F. Codd en los años 1969 y 1970. De todos modos, hasta la década de los ochenta no se empezaron a comercializar los primeros SGBD relacionales con



rendimientos aceptables. Cabe señalar que los SGBD relacionales que se comercializan actualmente todavía no soportan todo lo que establece la teoría relacional hasta el último detalle.

El principal objetivo del modelo de datos relacional es facilitar que la base de datos sea percibida o vista por el usuario como una estructura lógica que consiste en un conjunto de relaciones y no como una estructura física de implementación. Esto ayuda a conseguir un alto grado de independencia de los datos.

Un objetivo adicional del modelo es conseguir que esta estructura lógica con la que se percibe la base de datos sea simple y uniforme. Con el fin de proporcionar simplicidad y uniformidad, toda la información se representa de una única manera: mediante valores explícitos que contienen las relaciones (no se utilizan conceptos como por ejemplo apuntadores entre las relaciones). Con el mismo propósito, todos los valores de datos se consideran atómicos; es decir, no es posible descomponerlos.

Hay que precisar que un SGBD relacional, en el nivel físico, puede emplear cualquier estructura de datos para implementar la estructura lógica formada por las relaciones. En particular, a nivel físico, el sistema puede utilizar apuntadores, índices, etc. Sin embargo, esta implementación física queda oculta al usuario.

En los siguientes apartados estudiaremos la estructura de los datos, las operaciones y las reglas de integridad del modelo relacional. Hay dos formas posibles de enfocar el estudio de los contenidos de este módulo. La primera consiste en seguirlos en orden de exposición. De este modo, se van tratando todos los elementos de la teoría del modelo relacional de forma muy precisa y en un orden lógico. Otra posibilidad, sin embargo, es empezar con la lectura del resumen final del módulo y leer después todo el resto de los contenidos en el orden normal. El resumen describe los aspectos más relevantes de la teoría relacional que se explican y, de este modo, proporciona una visión global de los contenidos del módulo que, para algunos estudiantes, puede ser útil comprender antes de iniciar un estudio detallado.

ESTRUCTURA DE LOS DATOS

El modelo relacional proporciona una estructura de los datos que consiste en un conjunto de relaciones con objeto de representar la información que nos interesa del mundo real.

La estructura de los datos del modelo relacional se basa, pues, en el concepto de relación.

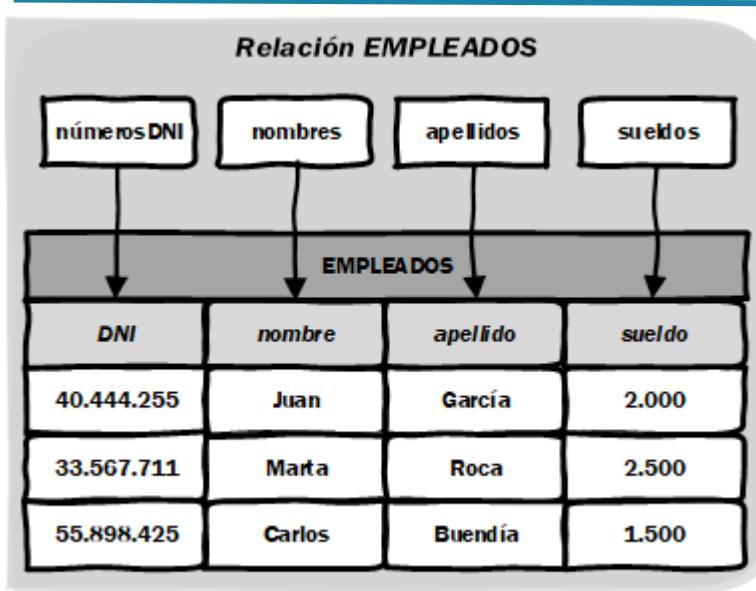
VISIÓN INFORMAL DE UNA RELACIÓN

En primer lugar, presentaremos el concepto de relación de manera informal. Se puede obtener una buena idea intuitiva de lo que es una relación si la visualizamos como una tabla o un fichero. En la figura 1 se muestra la visualización tabular de una relación que contiene datos de empleados. Cada fila de la tabla contiene una colección de valores de datos relacionados entre sí; en nuestro ejemplo, son los datos correspondientes a un mismo empleado.

La tabla tiene un nombre (EMPLEADOS) y también tiene un nombre cada una de sus columnas (DNI, nombre, apellido y sueldo). El nombre de la tabla y los de las columnas ayudan a entender el significado de los valores que contiene la tabla. Cada columna contiene valores de un cierto dominio; por ejemplo, la columna DNI contiene valores del dominio númerosDNI.

Si definimos las relaciones de forma más precisa, nos daremos cuenta de que presentan algunas características importantes que, en la visión superficial que hemos presentado, quedan ocultas. Estas características son las que motivan que el concepto de relación sea totalmente diferente del de fichero, a pesar de que, a primera vista, relaciones y ficheros puedan parecer similares.





Conjunto de relaciones

Una base de datos relacional consta de un conjunto de relaciones, cada una de las cuales se puede visualizar de este modo tan sencillo.

La estructura de los datos del modelo relacional resulta fácil de entender para el usuario.

VISIÓN FORMAL DE UNA RELACIÓN

A continuación, definimos formalmente las relaciones y otros conceptos que están vinculados a ellas, como por ejemplo dominio, esquema de relación, etc.

Un dominio D es un conjunto de valores atómicos. Por lo que respecta al modelo relacional, atómico significa indivisible; es decir, que por muy complejo o largo que sea un valor atómico, no tiene una estructuración interna para un SGBD relacional.

Los dominios pueden ser de dos tipos:

- 1) Dominios predefinidos, que corresponde a los tipos de datos que normalmente proporcionan los lenguajes de bases de datos, como por ejemplo los enteros, las cadenas de caracteres, los reales, etc.
- 2) Dominios definidos por el usuario, que pueden ser más específicos. Toda definición de un dominio debe constar, como mínimo, del nombre del dominio y de la descripción de los valores que forman parte de éste.

Dominio definido por el usuario

Por ejemplo, el usuario puede definir un dominio para las edades de los empleados que se denominé dom_edad y que contenga los valores enteros que están entre 16 y 65.

Una relación se compone del esquema (o intención de la relación) y de la extensión.

Si consideramos la representación tabular anterior (figura 1), el esquema correspondería a la cabecera de la tabla y la extensión correspondería al cuerpo:



EMPLEADOS			
DNI	nombre	apellido	sueldo
40.444.255	Juan	García	2.000
33.567.711	Marta	Roca	2.500
55.898.425	Carlos	Buendía	1.500

Esquema

Extensión

El esquema de la relación consiste en un nombre de relación R y un conjunto de atributos $\{A_1, A_2, \dots, A_n\}$.

NOMBRE Y CONJUNTO DE ATRIBUTOS DE LA RELACIÓN EMPLEADOS

Si tomamos como ejemplo la figura 1, el nombre de la relación es *EMPLEADOS* y el conjunto de atributos es $\{DNI, nombre, apellido, sueldo\}$.

Tomaremos la convención de denotar el esquema de la relación de la forma siguiente: $R(A_1, A_2, \dots, A_n)$, donde R es el nombre la relación y A_1, A_2, \dots, A_n es una ordenación cualquiera de los atributos que pertenecen al conjunto $\{A_1, A_2, \dots, A_n\}$.

DENOTACIÓN DEL ESKUEMA DE LA RELACIÓN EMPLEADOS

El esquema de la relación de la figura 1 se podría denotar, por ejemplo, como *EMPLEADOS(DNI, nombre, apellido, sueldo)*, o también, *EMPLEADOS(nombre, apellido, DNI, sueldo)*, porque cualquier ordenación de sus atributos se considera válida para denotar el esquema de una relación.

Un atributo A_i es el nombre del papel que ejerce un dominio D en un esquema de relación. D es el dominio de A_i y se denota como dominio (A_i).

DOMINIO DEL ATRIBUTO DNI

Según la figura 1, el atributo *DNI* corresponde al papel que ejerce el dominio *númerosDNI* en el esquema de la relación *EMPLEADOS* y, entonces, $\text{dominio}(DNI) = \text{númerosDNI}$.

Conviene observar que cada atributo es único en un esquema de relación, porque no tiene sentido que un mismo dominio ejerza dos veces el mismo papel en un mismo esquema. Por consiguiente, no puede ocurrir que en un esquema de relación haya dos atributos con el mismo nombre. En cambio, sí que se puede repetir un nombre de atributo en relaciones diferentes. Los dominios de los atributos, por el contrario, no deben ser necesariamente todos diferentes en una relación.

EJEMPLO DE ATRIBUTOS DIFERENTES CON EL MISMO DOMINIO

Si tomamos como ejemplo el esquema de relación *PERSONAS(DNI, nombre, apellido, telcasa, teltrabajo)*, los atributos *telcasa* y *teltrabajo* pueden tener el mismo dominio: $\text{dominio}(\text{telcasa}) = \text{teléfono}$ y $\text{dominio}(\text{teltrabajo}) = \text{teléfono}$.

En este caso, el dominio *teléfono* ejerce dos papeles diferentes en el esquema de relación: el de indicar el teléfono particular de una persona y el de indicar el del trabajo.



La extensión de la relación de esquema $R(A_1, A_2, \dots, A_n)$ es un conjunto de tuplas t_i ($i = 1, 2, \dots, m$), donde cada tupla t_i es, a su vez un conjunto de pares $t_i = \{<A_1:v_{i1}>, <A_2:v_{i2}> \dots <A_n:v_{in}>\}$ y, para cada par $<A_j:v_{ij}>$, se cumple que v_{ij} es un valor de dominio(A_j), o bien un valor especial que denominaremos nulo.

Para simplificar, tomaremos la convención de referirnos a una tupla $t_i = \{<A_1:v_{i1}>, <A_2:v_{i2}>, \dots, <A_n:v_{in}>\}$ que pertenece a la extensión del esquema denotado como $R(A_1, A_2, \dots, A_n)$, de la forma siguiente: $t_i = <v_{i1}, v_{i2}, \dots, v_{in}>$.

Si denotamos el esquema de la relación representada en la figura 1 como $\text{EMPLEADOS}(DNI, nombre, apellido, sueldo}$, el conjunto de tuplas de su extensión será el de la figura siguiente:

Algunos autores...

... denominan tablas, columnas y filas a las relaciones, los atributos y las tuplas, respectivamente.

Extensión de la relación de esquema $\text{EMPLEADOS}(DNI, nombre, apellido, sueldo)$

- <40.444.255, Juan, García, 2.000>**
- <33.567.711, Marta, Roca, 2.500>**
- <55.898.425, Carlos, Buendía, 1.500>**

Esta figura...

... nos muestra la extensión de EMPLEADOS en forma de conjunto, mientras que las figuras anteriores nos la mostraban en forma de filas de una tabla. La representación tabular es más cómoda, pero no refleja la definición de extensión con tanta exactitud

Si en una tupla $t_i = <v_{i1}, v_{i2}, \dots, v_{in}>$, el valor v_{ij} es un **valor nulo**, entonces el valor del atributo A_j es desconocido para la tupla t_i de la relación, o bien no es aplicable a esta tupla.

EJEMPLO DE VALOR NULO

Podríamos tener un atributo *telcasa* en la relación EMPLEADOS y se podría dar el caso de que un empleado no tuviese teléfono en su casa, o bien que lo tuviese, pero no se conociese su número. En las dos situaciones, el valor del atributo *telcasa* para la tupla correspondiente al empleado sería el valor nulo.

El grado de una relación es el número de atributos que pertenecen a su esquema.

GRADO DE LA RELACIÓN EMPLEADOS

El grado de la relación de esquema $\text{EMPLEADOS}(DNI, nombre, apellido, sueldo)$, es 4.

La cardinalidad de una relación es el número de tuplas que pertenecen a su extensión.

CARDINALIDAD DE LA RELACIÓN EMPLEADOS

Observando la figura 3 se deduce que la cardinalidad de la relación EMPLEADOS es 3.



DIFERENCIAS ENTRE RELACIONES Y FICHEROS

A primera vista, relaciones y ficheros resultan similares. Los registros y los campos que forman los ficheros se parecen a las tuplas y a los atributos de las relaciones, respectivamente.

A pesar de esta similitud superficial, la visión formal de relación que hemos presentado establece algunas características de las relaciones que las hacen diferentes de los ficheros clásicos. A continuación, describimos estas características:

- 1) **Atomicidad de los valores de los atributos:** los valores de los atributos de una relación deben ser atómicos; es decir, no deben tener estructura interna. Esta característica proviene del hecho de que los atributos siempre deben tomar un valor de su dominio o bien un valor nulo, y de que se ha establecido que los valores de los dominios deben ser atómicos en el modelo relacional.

El objetivo de la atomicidad de los valores es dar simplicidad y uniformidad al modelo relacional.

- 2) **No-repetición de las tuplas:** en un fichero clásico puede ocurrir que dos de los registros sean exactamente iguales; es decir, que contengan los mismos datos. En el caso del modelo relacional, en cambio, no es posible que una relación contenga tuplas repetidas. Esta característica se deduce de la misma definición de la extensión de una relación. La extensión es un conjunto de tuplas y, en un conjunto, no puede haber elementos repetidos.

- 3) **No-ordenación de las tuplas:** de la definición de la extensión de una relación como un conjunto de tuplas se deduce también que estas tuplas no estarán ordenadas, teniendo en cuenta que no es posible que haya una ordenación entre los elementos de un conjunto.

La finalidad de esta característica es conseguir que, mediante el modelo relacional, se puedan representar los hechos en un nivel abstracto que sea independiente de su estructura física de implementación. Más concretamente, aunque los SGBD relacionales deban proporcionar una implementación física que almacenará las tuplas de las relaciones en un orden concreto, esta ordenación no es visible si nos situamos en el nivel conceptual.

Ejemplo de no-ordenación de las tuplas

En una base de datos relacional, por ejemplo, no tiene sentido consultar la “primera tupla” de la relación **EMPLEADOS**.

- 4) **No-ordenación de los atributos:** el esquema de una relación consta de un nombre de relación R y un conjunto de atributos $\{A_1, A_2, \dots, A_n\}$. Así pues, no hay un orden entre los atributos de un esquema de relación, teniendo en cuenta que estos atributos forman un conjunto.

Como en el caso anterior, el objetivo de esta característica es representar los hechos en un nivel abstracto, independientemente de su implementación física.

Ejemplo de no-ordenación de los atributos

El esquema de relación **EMPLEADOS(DNI, nombre, apellido, sueldo)** denota el mismo esquema de relación que **EMPLEADOS(nombre, apellido, DNI, sueldo)**.



CLAVE CANDIDATA, CLAVE PRIMARIA Y CLAVE ALTERNATIVA DE LAS RELACIONES

Toda la información que contiene una base de datos debe poderse identificar de alguna forma. En el caso particular de las bases de datos que siguen el modelo relacional, para identificar los datos que la base de datos contiene, se pueden utilizar las claves candidatas de las relaciones. A continuación, definimos qué se entiende por *clave candidata*, *clave primaria* y *clave alternativa* de una relación. Para hacerlo, será necesario definir el concepto de *superclave*.

Una superclave de una relación de esquema $R(A_1, A_2, \dots, A_n)$ es un subconjunto de los atributos del esquema tal que no puede haber dos tuplas en la extensión de la relación que tengan la misma combinación de valores para los atributos del subconjunto.

Una superclave, por lo tanto, nos permite identificar todas las tuplas que contiene la relación.

Por ejemplo, ...

... si se almacena información sobre los empleados de una empresa, es preciso tener la posibilidad de distinguir qué datos corresponden a cada uno de los diferentes empleados.

Observemos que...

... toda relación tiene, por lo menos, una superclave, que es la formada por todos los atributos de su esquema. Esto se debe a la propiedad que cumple toda relación de no tener tuplas repetidas. En el ejemplo de *EMPLEADOS(DNI, CUIL, nombre, apellido, teléfono)* esta superclave sería: {DNI, CUIL, nombre, apellido, teléfono}.

ALGUNAS SUPERCLAVES DE LA RELACIÓN EMPLEADOS

En la relación de esquema *EMPLEADOS(DNI, CUIL, nombre, apellido, teléfono)*, algunas de las superclaves de la relación serían los siguientes subconjuntos de atributos: {DNI, CUIL, nombre, apellido, teléfono}, {DNI, apellido}, {DNI} y {CUIL}.

Una clave candidata de una relación es una superclave C de la relación que cumple que ningún subconjunto propio de C es superclave

Es decir, C cumple que la eliminación de cualquiera de sus atributos da un conjunto de atributos que no es superclave de la relación. Intuitivamente, una clave candidata permite identificar cualquier tupla de una relación, de manera que no sobre ningón atributo para hacer la identificación.

Notemos que...

... puesto que toda relación tiene por lo menos una superclave, podemos garantizar que toda relación tiene como mínimo una clave candidata.



CLAVES CANDIDATAS DE EMPLEADOS

En la relación de esquema **EMPLEADOS(*DNI, CUIL, nombre, apellido, teléfono*)**, sólo hay dos claves candidatas: **{DNI}** y **{CUIL}**.

Habitualmente, una de las claves candidatas de una relación se designa clave primaria de la relación. La clave primaria es la clave candidata cuyos valores se utilizarán para identificar las tuplas de la relación.

El diseñador de la base de datos es quien elige la clave primaria de entre las claves candidatas.

Relación con una clave candidata

Si una relación sólo tiene una clave candidata, entonces esta clave candidata debe ser también su clave primaria.

Ya que todas las relaciones tienen como mínimo una clave candidata, podemos garantizar que, para toda relación, será posible designar una clave primaria.

Las claves candidatas no elegidas como clave primaria se denominan claves alternativas.

Utilizaremos la convención de subrayar los atributos que forman parte de la clave primaria en el esquema de la relación. Así pues, **R(A₁, A₂, ..., A_i, ..., A_n)** indica que los atributos A₁, A₂, ..., A_i forman la clave primaria de R.

ELECCIÓN DE LA CLAVE PRIMARIA DE EMPLEADOS

En la relación de esquema **EMPLEADOS(*DNI, CUIL, nombre, apellido, teléfono*)**, donde hay dos claves candidatas, **{DNI}** y **{CUIL}**, se puede elegir como clave primaria **{DNI}**. Lo indicaremos subrayando el atributo **DNI** en el esquema de la relación **EMPLEADOS(DNI, CUIL, nombre, apellido, teléfono)**. En este caso, la clave **{CUIL}** será una clave alternativa de **EMPLEADOS**.

Es posible que una clave candidata o una clave primaria conste de más de un atributo.

CLAVE PRIMARIA DE LA RELACIÓN DESPACHOS

En la relación de esquema **DESPACHOS(*edificio, número, superficie*)**, la clave primaria está formada por los atributos **edificio** y **número**. En este caso, podrá ocurrir que dos despachos diferentes estén en el mismo edificio, o bien que tengan el mismo número, pero nunca pasará que tengan la misma combinación de valores para **edificio** y **número**.

CLAVES FORÁNEAS DE LAS RELACIONES

Hasta ahora hemos estudiado las relaciones de forma individual, pero debemos tener en cuenta que una base de datos relacional normalmente contiene más de una relación, para poder representar distintos tipos de hechos que suceden en el mundo real. Por ejemplo, podríamos tener una pequeña base de datos que contuviese dos relaciones: una denominada **EMPLEADOS**, que almacenaría datos de los empleados de una empresa, y otra con el nombre **DESPACHOS**, que almacenaría los datos de los despachos que tiene la empresa.

Debemos considerar también que entre los distintos hechos que se dan en el mundo real pueden existir lazos o vínculos. Por ejemplo, los empleados que trabajan para una empresa pueden estar vinculados con los despachos de la empresa, porque a cada empleado se le asigna un despacho concreto para trabajar.

En el modelo relacional, para reflejar este tipo de vínculos, tenemos la posibilidad de expresar conexiones entre las distintas tuplas de las relaciones. Por ejemplo, en la base de datos anterior, que tiene las relaciones **EMPLEADOS** y



DESPACHOS, puede ser necesario conectar tuplas de *EMPLEADOS* con tuplas de *DESPACHOS* para indicar qué despacho tiene asignado cada empleado.

En ocasiones, incluso puede ser necesario reflejar lazos entre tuplas que pertenecen a una misma relación. Por ejemplo, en la misma base de datos anterior puede ser necesario conectar determinadas tuplas de *EMPLEADOS* con otras tuplas de *EMPLEADOS* para indicar, para cada empleado, quién actúa como su jefe.

El mecanismo que proporcionan las bases de datos relacionales para conectar tuplas son las claves foráneas de las relaciones. Las **claves foráneas** permiten establecer conexiones entre las tuplas de las relaciones. Para hacer la conexión, una clave foránea tiene el conjunto de atributos de una relación que referencian la clave primaria de otra relación (o incluso de la misma relación).

CLAVES FORÁNEAS DE LA RELACIÓN EMPLEADOS

En la figura siguiente, la relación *EMPLEADOS(DNI, nombre, apellido, teléfono, DNIjefe, edificiodesp, númerodesp)*, tiene una clave foránea formada por los atributos *edificiodesp* y *númerodesp* que se refiere a la clave primaria de la relación *DESPACHOS(edificio, número, superficie)*. Esta clave foránea indica, para cada empleado, el despacho donde trabaja. Además, el atributo *DNIjefe* es otra clave foránea que referencia la clave primaria de la misma relación *EMPLEADOS*, e indica, para cada empleado, quien es su jefe.



Las claves foráneas tienen por objetivo establecer una conexión con la clave primaria que referencian. Por lo tanto, los valores de una clave foránea deben estar presentes en la clave primaria correspondiente, o bien deben ser valores nulos. En caso contrario, la clave foránea representaría una referencia o conexión incorrecta.

EJEMPLO

En la relación de esquema *EMPLEADOS(DNI, nombre, apellido, DNIjefe, edificiodesp, númerodesp)*, la clave foránea *{edificiodesp, númerodesp}* referencia la relación *DESPACHOS(edificio, número, superficie)*. De este modo, se cumple que todos los valores que no son nulos de los atributos *edificiodesp* y *númerodesp* son valores que existen para los atributos *edificio* y *número* de *DESPACHOS*, tal y como se puede ver a continuación:



- Relación *DESPACHOS*:

DESPACHOS		
<u>edificio</u>	<u>número</u>	<u>superficie</u>
Marina	120	10
Marina	122	15
Marina	230	20
Diagonal	120	10

- Relación *EMPLEADOS*:

EMPLEADOS					
<u>DNI</u>	<u>nombre</u>	<u>apellido</u>	<u>DNIjefe</u>	<u>edificiodesp</u>	<u>númerodesp</u>
40.444.255	Juan	García	NULO	Marina	120
33.567.711	Marta	Roca	40.444.255	Marina	120
55.898.425	Carlos	Buendía	40.444.255	Diagonal	120
77.232.144	Elena	Pla	40.444.255	NULO	NULO

Supongamos que hubiese un empleado con los valores <55.555.555, María, Casagran, NULO, París, 400>. Puesto que no hay ningún despacho con los valores París y 400 para *edificio* y *número*, la tupla de este empleado hace una referencia incorrecta; es decir, indica un despacho para el empleado que, de hecho, no existe.

Es preciso señalar que en la relación *EMPLEADOS* hay otra clave foránea, {*DNIjefe*}, que referencia la misma relación *EMPLEADOS*, y entonces se cumple que todos los valores que no son nulos del atributo *DNIjefe* son valores que existen para el atributo *DNI* de la misma relación *EMPLEADOS*.

A continuación, estableceremos de forma más precisa qué se entiende por *clave foránea*.

Una clave foránea de una relación R es un subconjunto de atributos del esquema de la relación, que denominamos CF y que cumple las siguientes condiciones:

1) Existe una relación S (S no debe ser necesariamente diferente de R) que tiene por clave primaria CP.

2) Se cumple que, para toda tupla t de la extensión de R, los valores para CF de t son valores nulos o bien valores que coinciden con los valores para CP de alguna tupla s de S.

Y entonces, se dice que la clave foránea CF referencia la clave primaria CP de la relación S, y también que la clave foránea CF referencia la relación S.

Conviene subrayar que, ...

... tal y como ya hemos mencionado, el modelo relacional permite representar toda la información mediante valores explícitos que contienen las relaciones, y no le hace falta nada más. De este modo, las conexiones entre tuplas de las relaciones se expresan con los valores explícitos de las claves foráneas de las relaciones, y no son necesarios conceptos adicionales (por ejemplo, apuntadores entre tuplas), para establecer estas conexiones. Esta característica da simplicidad y uniformidad al modelo.



De la noción que hemos dado de clave foránea se pueden extraer varias consecuencias:

- 1) Si una clave foránea *CF* referencia una clave primaria *CP*, el número de atributos de *CF* y de *CP* debe coincidir.

Ejemplo de coincidencia del número de atributos de *CF* y *CP*

En el ejemplo anterior, tanto la clave foránea *{edificiodesp, númerodesp}* como la clave primaria que referencia *{edificio, número}* tienen dos atributos. Si no sucediese así, no sería posible que los valores de *CF* existieran en *CP*.

- 2) Por el mismo motivo, se puede establecer una correspondencia (en concreto, una biyección) entre los atributos de la clave foránea y los atributos de la clave primaria que referencia.

Ejemplo de correspondencia entre los atributos de *CF* y los de *CP*

En el ejemplo anterior, a *edificiodesp* le corresponde el atributo *edificio*, y a *númerodesp* le corresponde el atributo *número*.

- 3) También se deduce de la noción de *clave foránea* que los dominios de sus atributos deben coincidir con los dominios de los atributos correspondientes a la clave primaria que referencia. Esta coincidencia de dominios hace que sea posible que los valores de la clave foránea coincidan con valores de la clave primaria referenciada.

Ejemplo de coincidencia de los dominios

En el ejemplo anterior, se debe cumplir que $\text{dominio}(\text{edificiodesp}) = \text{dominio}(\text{edificio})$ y también que $\text{dominio}(\text{númerodesp}) = \text{dominio}(\text{número})$.

Observemos que, de hecho, esta condición se podría relajar, y se podría permitir que los dominios no fuesen exactamente iguales, sino que sólo fuesen, y de alguna forma que convendría precisar, dominios “compatibles”.

Para simplificarlo, nosotros supondremos que los dominios deben ser iguales en todos los casos en que, según Date (2001), se aceptarían dominios “compatibles”.

Ejemplo de atributo que forma parte de la clave primaria y de una clave foránea

Puede suceder que algún atributo de una relación forme parte tanto de la clave primaria como de una clave foránea de la relación. Esto se da en las relaciones siguientes: EDIFICIOS(*nombreedificio, dirección*), y DESPACHOS(*edificio, número, superficie*), donde *{edificio}* es una clave foránea que referencia EDIFICIOS.

En este ejemplo, el atributo *edificio* forma parte tanto de la clave primaria como de la clave foránea de la relación DESPACHOS.

CREACIÓN DE LAS RELACIONES DE UNA BASE DE DATOS

Hemos visto que una base de datos relacional consta de varias relaciones. Cada relación tiene varios atributos que toman valores de unos ciertos dominios; también tiene una clave primaria y puede tener una o más claves foráneas. Los **lenguajes de los SGBD relacionales** deben proporcionar la forma de definir todos estos elementos para crear una base de datos.

Más adelante se verá con detalle la sintaxis y el significado de las sentencias de definición de la base de datos para el caso concreto del lenguaje SQL.

OPERACIONES DEL MODELO RELACIONAL

Las operaciones del modelo relacional deben permitir manipular datos almacenados en una base de datos relacional y, por lo tanto, estructurados en forma de relaciones. La manipulación de datos incluye básicamente dos aspectos: la actualización y la consulta.

La actualización de los datos consiste en hacer que los cambios que se producen en la realidad queden reflejados en las relaciones de la base de datos.



EJEMPLO DE ACTUALIZACIÓN

Si una base de datos contiene, por ejemplo, información de los empleados de una empresa, y la empresa contrata a un empleado, será necesario reflejar este cambio añadiendo los datos del nuevo empleado a la base de datos.

Existen tres operaciones básicas de actualización:

- a) **Inserción**, que sirve para añadir una o más tuplas a una relación.
- b) **Borrado**, que sirve para eliminar una o más tuplas de una relación.
- c) **Modificación**, que sirve para alterar los valores que tienen una o más tuplas de una relación para uno o más de sus atributos.

La consulta de los datos consiste en la obtención de datos deducibles a partir de las relaciones que contiene la base de datos

EJEMPLO DE CONSULTA

Si una base de datos contiene, por ejemplo, información de los empleados de una empresa, puede interesar consultar el nombre y apellido de todos los empleados que trabajan en un despacho situado en un edificio que tiene por nombre *Marina*.

La obtención de los datos que responden a una consulta puede requerir el análisis y la extracción de datos de una o más de las relaciones que mantiene la base de datos.

Según la forma como se especifican las consultas, podemos clasificar los lenguajes relationales en dos tipos:

- 1) **Lenguajes basados en el álgebra relacional**. El álgebra relacional se inspira en la teoría de conjuntos. Si queremos especificar una consulta, es necesario seguir uno o más pasos que sirven para ir construyendo, mediante operaciones del álgebra relacional, una nueva relación que contenga los datos que responden a la consulta a partir de las relaciones almacenadas. Los lenguajes basados en el álgebra relacional son **lenguajes procedimentales**, ya que los pasos que forman la consulta describen un procedimiento.
- 2) **Lenguajes basados en el cálculo relacional**. El cálculo relacional tiene su fundamento teórico en el cálculo de predicados de la lógica matemática. Proporciona una notación que permite formular la definición de la relación donde están los datos que responden la consulta en términos de las relaciones almacenadas. Esta definición no describe un procedimiento; por lo tanto, se dice que los lenguajes basados en el cálculo relacional son **lenguajes declarativos** (no procedimentales).

El **lenguaje SQL**, en las sentencias de consulta, combina construcciones del álgebra relacional y del cálculo relacional con un predominio de las construcciones del cálculo. Este predominio determina que SQL sea un lenguaje declarativo.

El **estudio del álgebra relacional** presenta un interés especial, pues ayuda a entender qué servicios de consulta debe proporcionar un lenguaje relacional, facilita la comprensión de algunas de las construcciones del lenguaje SQL y también sirve de base para el tratamiento de las consultas que efectúan los SGBD internamente. Este último tema queda fuera del ámbito del presente curso, pero es necesario para estudios más avanzados sobre bases de datos.

REGLAS DE INTEGRIDAD

Una base de datos contiene unos datos que, en cada momento, deben reflejar la realidad o, más concretamente, la situación de una porción del mundo real. En el caso de las bases de datos relationales, esto significa que la extensión de las relaciones (es decir, las tuplas que contienen las relaciones) deben tener valores que reflejen la realidad correctamente.



Suele ser bastante frecuente que determinadas configuraciones de valores para las tuplas de las relaciones no tengan sentido, porque no representan ninguna situación posible del mundo real.

UN SUELDO NEGATIVO

En la relación de esquema *EMPLEADOS(DNI, nombre, apellido, sueldo)*, una tupla que tiene un valor de -1.000 para el sueldo probablemente no tiene sentido, porque los sueldos no pueden ser negativos.

Denominamos integridad la propiedad de los datos de corresponder a representaciones plausibles del mundo real.

Como es evidente, para que los datos sean íntegros, es preciso que cumplan varias condiciones.

El hecho de que los sueldos no puedan ser negativos es una condición que se debería cumplir en la relación *EMPLEADOS*.

En general, las condiciones que garantizan la integridad de los datos pueden ser de dos tipos:

- 1) Las restricciones de integridad de usuario** son condiciones específicas de una base de datos concreta; es decir, son las que se deben cumplir en una base de datos particular con unos usuarios concretos, pero que no son necesariamente relevantes en otra base de datos.

Restricción de integridad de usuario en *EMPLEADOS*

Éste sería el caso de la condición anterior, según la cual los sueldos no podían ser negativos. Observemos que esta condición era necesaria en la base de datos concreta de este ejemplo porque aparecía el atributo *sueldo*, al que se quería dar un significado; sin embargo, podría no ser necesaria en otra base de datos diferente donde, por ejemplo, no hubiese sueldos.

- 2) Las reglas de integridad de modelo**, en cambio, son condiciones más generales, propias de un modelo de datos, y se deben cumplir en toda base de datos que siga dicho modelo.

Ejemplo de regla de integridad del modelo de datos relacional

En el caso del modelo de datos relacional, habrá una regla de integridad para garantizar que los valores de una clave primaria de una relación no se repitan en tuplas diferentes de la relación. Toda base de datos relacional debe cumplir esta regla que, por lo tanto, es una regla de integridad del modelo.

Los SGBD deben proporcionar la forma de definir las restricciones de integridad de usuario de una base de datos; una vez definidas, deben velar por su cumplimiento.

Las reglas de integridad del modelo, en cambio, no se deben definir para cada base de datos concreta, porque se consideran preestablecidas para todas las bases de datos de un modelo. Un SGBD de un modelo determinado debe velar por el cumplimiento de las reglas de integridad preestablecidas por su modelo.

A continuación, estudiaremos con detalle las **reglas de integridad del modelo relacional**, reglas que todo SGBD relacional debe obligar a cumplir.

REGLA DE INTEGRIDAD DE UNICIDAD DE LA CLAVE PRIMARIA

La regla de integridad de unicidad está relacionada con la definición de clave primaria. Concretamente, establece que toda clave primaria que se elija para una relación no debe tener valores repetidos.

EJEMPLO

Tenemos la siguiente relación:



DESPACHOS		
<u>edificio</u>	<u>número</u>	superficie
Marina	120	10
Marina	122	15
Marina	230	20
Diagonal	120	10

En esta relación, dado que la clave primaria está formada por *edificio* y *número*, no hay ningún despacho que repita tanto *edificio* como *número* de otro despacho. Sin embargo, sí se repiten valores de *edificio* (por ejemplo, Marina); y también se repiten valores de *número* (120). A pesar de ello, el *edificio* y el *número* no se repiten nunca al mismo tiempo.

A continuación, explicamos esta regla de forma más precisa.

La regla de integridad de unicidad de la clave primaria establece que, si el conjunto de atributos CP es la clave primaria de una relación R, entonces la extensión de R no puede tener en ningún momento dos tuplas con la misma combinación de valores para los atributos de CP.

Un SGBD relacional deberá garantizar el cumplimiento de esta regla de integridad en todas las inserciones, así como en todas las modificaciones que afecten a atributos que pertenecen a la clave primaria de la relación.

EJEMPLO

Tenemos la siguiente relación:

DESPACHOS		
<u>edificio</u>	<u>número</u>	superficie
Marina	120	10
Marina	122	15
Marina	230	20
Diagonal	120	10

En esta relación no se debería poder insertar la tupla <Diagonal, 120, 30>, ni modificar la tupla <Marina, 122, 15>, de modo que pasara a ser <Marina, 120, 15>.

REGLA DE INTEGRIDAD DE ENTIDAD DE LA CLAVE PRIMARIA

La regla de integridad de entidad de la clave primaria dispone que los atributos de la clave primaria de una relación no pueden tener valores nulos.

EJEMPLO

Tenemos la siguiente relación:



DESPACHOS		
<u>edificio</u>	<u>número</u>	<u>superficie</u>
Marina	120	10
Marina	122	15
Marina	230	20
Diagonal	120	10

En esta relación, puesto que la clave primaria está formada por *edificio* y *número*, no hay ningún despacho que tenga un valor nulo para *edificio*, ni tampoco para *número*.

Esta regla es necesaria para que los valores de las claves primarias puedan identificar las tuplas individuales de las relaciones. Si las claves primarias tuviesen valores nulos, es posible que algunas tuplas no se pudieran distinguir.

EJEMPLO DE CLAVE PRIMARIA INCORRECTA CON VALORES NULOS

En el ejemplo anterior, si un despacho tuviese un valor nulo para *edificio* porque en un momento dado el nombre de este edificio no se conoce, por ejemplo <NULL, 120, 30>, la clave primaria no nos permitiría distinguirlo del despacho <Marina, 120, 10> ni del despacho <Diagonal, 120, 10>. No podríamos estar seguros de que el valor desconocido de *edificio* no es ni Marina ni Diagonal.

A continuación, definimos esta regla de forma más precisa.

La regla de integridad de entidad de la clave primaria establece que, si el conjunto de atributos CP es la clave primaria de una relación R, la extensión de R no puede tener ninguna tupla con algún valor nulo para alguno de los atributos de CP.

Un SGBD relacional tendrá que garantizar el cumplimiento de esta regla de integridad en todas las inserciones y, también, en todas las modificaciones que afecten a atributos que pertenecen a la clave primaria de la relación.

EJEMPLO

En la relación *DESPACHOS* anterior, no se debería insertar la tupla <Diagonal, NULL, 15>. Tampoco debería ser posible modificar la tupla <Marina, 120, 10> de modo que pasara a ser <NULL, 120, 10>.

REGLA DE INTEGRIDAD REFERENCIAL

La regla de integridad referencial está relacionada con el concepto de *clave foránea*. Concretamente, determina que todos los valores que toma una clave foránea deben ser valores nulos o valores que existen en la clave primaria que referencia.

EJEMPLO

Si tenemos las siguientes relaciones:



- Relación *DESPACHOS*:

DESPACHOS		
<u>edificio</u>	<u>número</u>	<u>superficie</u>
Marina	120	10
Marina	122	15
Marina	230	20
Diagonal	120	10

- Relación *EMPLEADOS*:

EMPLEADOS				
<u>DNI</u>	<u>nombre</u>	<u>apellido</u>	<u>edificiodesp</u>	<u>númerodesp</u>
40.444.255	Juan	García	Marina	120
33.567.711	Marta	Roca	Marina	120
55.898.425	Carlos	Buendía	Diagonal	120
77.232.144	Elena	Pla	NULO	NULO

donde *edificiodesp* y *númerodesp* de la relación *EMPLEADOS* forman una clave foránea que referencia la relación *DESPACHOS*. Debe ocurrir que los valores no nulos de *edificiodesp* y *númerodesp* de la relación *EMPLEADOS* estén en la relación *DESPACHOS* como valores de *edificio* y *número*. Por ejemplo, el empleado <40.444.255, Juan García, Marina, 120> tiene el valor Marina para *edificiodesp*, y el valor 120 para *númerodesp*, de modo que en la relación *DESPACHOS* hay un despacho con valor Marina para *edificio* y con valor 120 para *número*.

La necesidad de la regla de integridad relacional proviene del hecho de que las claves foráneas tienen por objetivo establecer una conexión con la clave primaria que referencian. Si un valor de una clave foránea no estuviese presente en la clave primaria correspondiente, representaría una referencia o una conexión incorrecta.

REFERENCIA INCORRECTA

Supongamos que en el ejemplo anterior hubiese un empleado con los valores <56.666.789, Pedro, López, Valencia, 325>. Ya que no hay un despacho con los valores Valencia y 325 para *edificio* y *número*, la tupla de este empleado hace una referencia incorrecta; es decir, indica un despacho para el empleado que, de hecho, no existe.

A continuación, explicamos la regla de modo más preciso.



La regla de integridad referencial establece que si el conjunto de atributos CF es una clave foránea de una relación R que referencia una relación S (no necesariamente diferente de R), que tiene por clave primaria CP, entonces, para toda tupla t de la extensión de R, los valores para el conjunto de atributos CF de t son valores nulos, o bien valores que coinciden con los valores para CP de alguna tupla s de S.

En el caso de que una tupla t de la extensión de R tenga valores para CF que coincidan con los valores para CP de una tupla s de S, decimos que t es una tupla que referencia s y que s es una tupla que tiene una clave primaria referenciada por t.

Un SGBD relacional tendrá que hacer cumplir esta regla de integridad. Deberá efectuar comprobaciones cuando se produzcan las siguientes operaciones:

- a) Inserciones en una relación que tenga una clave foránea.
- b) Modificaciones que afecten a atributos que pertenecen a la clave foránea de una relación.
- c) Borrados en relaciones referenciadas por otras relaciones.
- d) Modificaciones que afecten a atributos que pertenecen a la clave primaria de una relación referenciada por otra relación.

EJEMPLO

Retomamos el ejemplo anterior, donde *edificiodesp* y *númerodesp* de la relación *EMPLEADOS* forman una clave foránea que referencia la relación *DESPACHOS*:

- Relación DESPACHOS:

DESPACHOS		
<u>edificio</u>	<u>número</u>	<u>superficie</u>
Marina	120	10
Marina	122	15
Marina	230	20
Diagonal	120	10

- Relación *EMPLEADOS*:

EMPLEADOS				
<u>DNI</u>	<u>nombre</u>	<u>apellido</u>	<u>edificiodesp</u>	<u>númerodesp</u>
40.444.255	Juan	García	Marina	120
33.567.711	Marta	Roca	Marina	120
55.898.425	Carlos	Buendía	Diagonal	120
77.232.144	Elena	Pla	NULO	NULO

Las siguientes operaciones provocarían el incumplimiento de la regla de integridad referencial:



- Inserción de <12.764.411, Jorge, Puig, Diagonal, 220> en *EMPLEADOS*.
- Modificación de <40.444.255, Juan, García, Marina, 120> de *EMPLEADOS* por <40.444.255, Juan, García, Marina, 400>.
- Borrado de <Marina, 120, 10> de *DESPACHOS*.
- Modificación de <Diagonal, 120, 10> de *DESPACHOS* por <París, 120, 10>.

Un SGBD relacional debe procurar que se cumplan las reglas de integridad del modelo. Una forma habitual de mantener estas reglas consiste en rechazar toda operación de actualización que deje la base de datos en un estado en el que alguna regla no se cumpla. En algunos casos, sin embargo, el SGBD tiene la posibilidad de aceptar la operación y efectuar acciones adicionales compensatorias, de modo que el estado que se obtenga satisfaga las reglas de integridad, a pesar de haber ejecutado la operación.

Esta última política se puede aplicar en las siguientes operaciones de actualización que violarían la regla de integridad:

- a) Borrado de una tupla que tiene una clave primaria referenciada.
- b) Modificación de los valores de los atributos de la clave primaria de una tupla que tiene una clave primaria referenciada.

En los casos anteriores, algunas de las políticas que se podrán aplicar serán las siguientes: **restricción, actualización en cascada y anulación**. A continuación, explicamos el significado de las tres posibilidades mencionadas.

RESTRICCIÓN

La política de restricción consiste en no aceptar la operación de actualización.

Más concretamente, la restricción en caso de borrado, consiste en no permitir borrar una tupla si tiene una clave primaria referenciada por alguna clave foránea.

De forma similar, la restricción en caso de modificación consiste en no permitir modificar ningún atributo de la clave primaria de una tupla si tiene una clave primaria referenciada por alguna clave foránea.

EJEMPLO DE APLICACIÓN DE LA RESTRICCIÓN

Supongamos que tenemos las siguientes relaciones:

- Relación *CLIENTES*:

CLIENTES	
	<u>numcliente</u>
	10
	15
	18
	...
	-
	-
	-



- Relación *PEDIDOS_PENDIENTES*:

PEDIDOS_PENDIENTES		
<u>numped</u>	...	<u>numcliente*</u>
1.234	-	10
1.235	-	10
1.236	-	15

* {numcliente} referencia *CLIENTES*.

- Si aplicamos la restricción en caso de borrado y, por ejemplo, queremos borrar al cliente número 10, no podremos hacerlo porque tiene pedidos pendientes que lo referencian.
- Si aplicamos la restricción en caso de modificación y queremos modificar el número del cliente 15, no será posible hacerlo porque también tiene pedidos pendientes que lo referencian.

ACTUALIZACIÓN EN CASCADA

La política de actualización en cascada consiste en permitir la operación de actualización de la tupla, y en efectuar operaciones compensatorias que propaguen en cascada la actualización a las tuplas que la referenciaban; se actúa de este modo para mantener la integridad referencial.

Más concretamente, la actualización en cascada en caso de borrado consiste en permitir el borrado de una tupla t que tiene una clave primaria referenciada, y borrar también todas las tuplas que referencian t.

De forma similar, la actualización en cascada en caso de modificación consiste en permitir la modificación de atributos de la clave primaria de una tupla t que tiene una clave primaria referenciada, y modificar del mismo modo todas las tuplas que referencian t.

EJEMPLO DE APLICACIÓN DE LA ACTUALIZACIÓN EN CASCADA

Supongamos que tenemos las siguientes relaciones:

- Relación EDIFICIOS:

EDIFICIOS	
<u>nombreedificio</u>	...
Marina	-
Diagonal	-



- Relación *DESPACHOS*:

DESPACHOS		
<u>edificio*</u>	<u>número</u>	<u>superficie</u>
Marina	120	10
Marina	122	15
Marina	230	20
Diagonal	120	10

* {edificio} referencia *EDIFICIOS*

- a) Si aplicamos la actualización en cascada en caso de borrado y, por ejemplo, queremos borrar el edificio Diagonal, se borrará también el despacho Diagonal 120 que hay en el edificio, y nos quedará:

- Relación *EDIFICIOS*:

EDIFICIOS		
<u>nombreedificio</u>	...	
Marina	-	

- Relación *DESPACHOS*:

DESPACHOS		
<u>edificio*</u>	<u>número</u>	<u>superficie</u>
Marina	120	10
Marina	122	15
Marina	230	20

* {edificio} referencia *EDIFICIOS*

- b) Si aplicamos la actualización en cascada en caso de modificación, y queremos modificar el nombre del edificio Marina por Mar, también se cambiará Marina por Mar en los despachos Marina 120, Marina 122 y Marina 230, y nos quedará:

- Relación *EDIFICIOS*:

EDIFICIOS		
<u>nombreedificio</u>	...	
Mar	-	

- Relación *DESPACHOS*:

DESPACHOS		
<u>edificio*</u>	<u>número</u>	<u>superficie</u>
Mar	120	10
Mar	122	15
Mar	230	20

* {edificio} referencia *EDIFICIOS*



ANULACIÓN

Esta política consiste en permitir la operación de actualización de la tupla y en efectuar operaciones compensatorias que pongan valores nulos a los atributos de la clave foránea de las tuplas que la referencian; esta acción se lleva a cabo para mantener la integridad referencial.

Puesto que generalmente los SGBD relacionales permiten establecer que un determinado atributo de una relación no admite valores nulos, sólo se puede aplicar la política de anulación si los atributos de la clave foránea sí los admiten.

Más concretamente, la anulación en caso de borrado consiste en permitir el borrado de una tupla *t* que tiene una clave referenciada y, además, modificar todas las tuplas que referencian *t*, de modo que los atributos de la clave foránea correspondiente tomen valores nulos.

De forma similar, la anulación en caso de modificación consiste en permitir la modificación de atributos de la clave primaria de una tupla *t* que tiene una clave referenciada y, además, modificar todas las tuplas que referencian *t*, de modo que los atributos de la clave foránea correspondiente tomen valores nulos.

EJEMPLO DE APLICACIÓN DE LA ANULACIÓN

El mejor modo de entender en qué consiste la anulación es mediante un ejemplo. Tenemos las siguientes relaciones:

- Relación *VENDEDORES*:

VENDEDORES		
<u>numvendedor</u>	...	
1	-	
2	-	
3	-	

- Relación *CLIENTES*:

CLIENTES		
<u>numcliente</u>	...	vendedorasig*
23	-	1
35	-	1
38	-	2
42	-	2
50	-	3

* {vendedorasig} referencia *VENDEDORES*.

- 1) Si aplicamos la anulación en caso de borrado y, por ejemplo, queremos borrar al vendedor número 1, se modificarán todos los clientes que lo tenían asignado, y pasarán a tener un valor nulo en vendedorasig. Nos quedará:



- Relación *VENDEDORES*:

VENDEDORES		
<u>numvendedor</u>
2	-	-
3	-	-

- Relación *CLIENTES*:

CLIENTES		
<u>numcliente</u>	...	vendedorasig*
23	-	NULO
35	-	NULO
38	-	2
42	-	2
50	-	3

* {vendedorasig} referencia *VENDEDORES*.

- 2) Si aplicamos la anulación en caso de modificación, y ahora queremos cambiar el número del vendedor 2 por 5, se modificarán todos los clientes que lo tenían asignado y pasarán a tener un valor nulo en vendedorasig. Nos quedará:

- Relación *VENDEDORES*:

VENDEDORES		
<u>numvendedor</u>
5	-	-
3	-	-

- Relación *CLIENTES*:

CLIENTES		
<u>numcliente</u>	...	vendedorasig*
23	-	NULO
35	-	NULO
38	-	NULO
42	-	NULO
50	-	3

* {vendedorasig} referencia *VENDEDORES*.



SELECCIÓN DE LA POLÍTICA DE MANTENIMIENTO DE LA INTEGRIDAD REFERENCIAL

Hemos visto que en caso de borrado o modificación de una clave primaria referenciada por alguna clave foránea hay varias políticas de mantenimiento de la regla de integridad referencial.

El diseñador puede elegir para cada clave foránea qué política se aplicará en caso de borrado de la clave primaria referenciada, y cuál en caso de modificación de ésta. El diseñador deberá tener en cuenta el significado de cada clave foránea concreta para poder elegir adecuadamente.

Aplicación de políticas diferentes

Puede ocurrir que, para una determinada clave foránea, la política adecuada en caso de borrado sea diferente de la adecuada en caso de modificación. Por ejemplo, puede ser necesario aplicar la restricción en caso de borrado y la actualización en cascada en caso de modificación.

REGLA DE INTEGRIDAD DE DOMINIO

La regla de integridad de dominio está relacionada, como su nombre indica, con la noción de *dominio*. Esta regla establece dos condiciones.

La primera condición consiste en que un valor no nulo de un atributo A_i debe pertenecer al dominio del atributo A_i ; es decir, debe pertenecer a $\text{dominio}(A_i)$.

Esta condición implica que todos los valores no nulos que contiene la base de datos para un determinado atributo deben ser del dominio declarado para dicho atributo.

EJEMPLO

Si en la relación *EMPLEADOS(DNI, nombre, apellido, edademp)* hemos declarado que *dominio(DNI)* es el dominio predefinido de los enteros, entonces no podremos insertar, por ejemplo, ningún empleado que tenga por *DNI* el valor "Luis", que no es un entero.

Recordemos que los dominios pueden ser de dos tipos: predefinidos o definidos por el usuario. Observad que los dominios definidos por el usuario resultan muy útiles, porque nos permiten determinar de forma más específica cuáles serán los valores admitidos por los atributos.

EJEMPLO

Supongamos ahora que en la relación *EMPLEADOS(DNI, nombre, apellido, edademp)* hemos declarado que *dominio(edademp)* es el dominio definido por el usuario *edad*. Supongamos también que el dominio *edad* se ha definido como el conjunto de los enteros que están entre 16 y 65. En este caso, por ejemplo, no será posible insertar un empleado con un valor de 90 para *edademp*.

La segunda condición de la regla de integridad de dominio es más compleja, especialmente en el caso de dominios definidos por el usuario; los SGBD actuales no la soportan para estos últimos dominios. Por estos motivos sólo la presentaremos superficialmente.

Esta segunda condición sirve para establecer que los operadores que pueden aplicarse sobre los valores dependen de los dominios de estos valores; es decir, un operador determinado sólo se puede aplicar sobre valores que tengan dominios que le sean adecuados



EJEMPLO

Analizaremos esta segunda condición de la regla de integridad de dominio con un ejemplo concreto. Si en la relación *EMPLEADOS(DNI, nombre, apellido, edademp)* se ha declarado que *dominio(DNI)* es el dominio predefinido de los enteros, entonces no se permitirá consultar todos aquellos empleados cuyo DNI sea igual a 'Elena' (*DNI* = 'Elena'). El motivo es que no tiene sentido que el operador de comparación = se aplique entre un *DNI* que tiene por dominio los enteros, y el valor 'Elena', que es una serie de caracteres.

De este modo, el hecho de que los operadores que se pueden aplicar sobre los valores dependan del dominio de estos valores permite detectar errores que se podrían cometer cuando se consulta o se actualiza la base de datos. Los dominios definidos por el usuario son muy útiles, porque nos permitirán determinar de forma más específica cuáles serán los operadores que se podrán aplicar sobre los valores.

EJEMPLO

Veamos otro ejemplo con dominios definidos por el usuario. Supongamos que en la conocida relación *EMPLEADOS(DNI, nombre, apellido, edademp)* se ha declarado que *dominio(DNI)* es el dominio definido por el usuario *númerosDNI* y que *dominio(edademp)* es el dominio definido por el usuario *edad*. Supongamos que *númerosDNI* corresponde a los enteros positivos y que *edad* corresponde a los enteros que están entre 16 y 65. En este caso, será incorrecto, por ejemplo, consultar los empleados que tienen el valor de *DNI* igual al valor de *edademp*. El motivo es que, aunque tanto los valores de *DNI* como los de *edademp* sean enteros, sus dominios son diferentes; por ello, según el significado que el usuario les da, no tiene sentido compararlos.

Sin embargo, los actuales SGBD relacionales no dan apoyo a la segunda condición de la regla de integridad de dominio para dominios definidos por el usuario. Si se quisiera hacer, sería necesario que el diseñador tuviese alguna forma de especificar, para cada operador que se desease utilizar, para qué combinaciones de dominios definidos por el usuario tiene sentido que se aplique. El lenguaje estándar SQL no incluye actualmente esta posibilidad.



ESQUEMA LÓGICO DE BASE DE DATOS

OBJETIVO:

Describir la visión que el usuario tendrá de los datos una vez que el sistema esté implementado.

Es necesario describir los datos mediante un modelo conceptual estándar y refinarlo por volúmenes de proceso; para esta explicación, de los modelos posibles, elegimos el modelo relacional. También será necesario definir el o los lenguajes de consulta que permitirán al usuario definir sus propias consultas.

HERRAMIENTAS

- Diagrama de Tablas
- Diccionario de Datos

TÉCNICAS

Se propone una técnica basada en la normalización y una adaptación al modelo relacional del procedimiento indicado por Teorey & Fry para el modelo en red (Design of Database Structures -1982).

El Esquema de Datos plantea un modelo de tipo conceptual; aquí el objetivo es mostrar la visión que el usuario tendrá una vez que los datos estén implementados. Dicha implementación dependerá de la tecnología adoptada (base de datos, procesadores, comunicaciones, etc.). Las alternativas que pueden aparecer son varias:

- Una base de datos en un procesador: El Esquema de Datos se implementará a través de una única base de datos (Construiremos un sólo Esquema Lógico de Base de Datos).
- Varias bases de datos en varios procesadores: El Esquema de Datos se implementará a través de varias bases de datos (Construiremos un Esquema Lógico de Base de Datos por cada base de datos).
- Una base de datos distribuida y varios procesadores: Debe construirse un único Esquema Lógico de Base de Datos.

A los efectos explicativos supondremos:

- Sistema de Base de Datos Centralizado: Para resolver los conflictos de distribución será necesario tomar en cuenta (e incluir) los costos de comunicaciones, y los de actualización de copias.
- El costo del sistema de recuperación de la Base de Datos es proporcional al de utilización de la misma.
- La resolución de conflictos de concurrencia, seguridad, privacidad, etc., son postergados.
- Se refina únicamente en base a las respuestas planeadas.



CONSTRUCCIÓN DEL ESQUEMA LÓGICO DE BASE DE DATOS

- A) Formular la estructura para un RDBMS¹⁸ por cada base de datos identificada
- Obtener las tablas del sistema, implementando los objetos y las relaciones del Esquema de Datos en tablas
 - Definir una tabla por cada objeto

En el modelo relacional se maneja la siguiente terminología:

Clave Primaria: Atributo que permite identificar únicamente una fila (registro) de una tabla.

Clave Foránea: Atributo cuyo valor referencia a la clave primaria de otra tabla

Columna: Sinónimo de atributo

En todos los casos, los atributos del objeto serán ahora las columnas de la tabla, y el o los identificadores constituirán su clave primaria.

IMPLEMENTACIÓN DE SUPERTIPOS Y SUBTIPOS

Deberán evaluarse las siguientes alternativas:

- Crear tablas separadas para el supertipo y sus subtipos: Esta es la mejor alternativa por su facilidad de adaptación ante futuros requerimientos; por otra parte, es la que generalmente hace mejor uso del espacio de almacenamiento. Puede ser la mejor opción si hay una gran cantidad de ocurrencias del supertipo y los subtipos, pero generalmente hace más complejo el diseño de los programas, ya que requiere una mayor navegación entre las tablas para seleccionar la ocurrencia deseada de un subtipo; para mejorar esta situación, puede agregarse una columna en la tabla supertipo para indicar que tabla subtipo tiene asociada (indicador de subtipo).
- Combinar el supertipo y todos sus subtipos en una única tabla: La tabla resultante posee todos los elementos de datos del supertipo y de cada uno de los subtipos. Es la opción más sencilla desde el punto de vista del acceso a disco y de la navegación, pero tiene un alto grado de redundancia ya que habrá un gran número de columnas con valores nulos. Puede no ser necesario agregar un indicador de subtipo, ya que los programas podrían detectar el subtipo simplemente controlando que determinadas columnas tengan o no valores nulos.
- Implementar todos los subtipos como tablas separadas y no implementar el supertipo: En este caso, las tablas que implementan los subtipos tienen sus propios elementos de datos y los del supertipo; es decir, se repiten los elementos del supertipo en cada una de las tablas que representan los subtipos. Obviamente, esto genera cierta redundancia, pero la alternativa puede resultar adecuada cuando el supertipo posee pocos elementos. Para facilitar el acceso es conveniente que la clave primaria de la tabla esté formada por el identificador del supertipo y un código de subtipo.
- Implementar el supertipo como una tabla y combinar todos los subtipos en una única tabla: La tabla que implementa los subtipos contiene los elementos de datos de todos los subtipos. Esta alternativa puede ser conveniente cuando la mayoría de los procesos acceden al supertipo.

¹⁸ RDBMS (Relational Data Base Management System): Sistema Administrador de Bases de Datos, por ejemplo: Oracle, SQL Server, Sybase, MySQL, PostgreSQL, DB2



IMPLEMENTACIÓN DE OBJETOS ASOCIATIVOS

Las columnas que formen parte de la clave primaria referenciarán a las claves primarias de las tablas que participen de la relación.

IMPLEMENTACIÓN DE OBJETOS DÉBILES

La clave primaria de la tabla resultante estará formada por la clave primaria de la tabla de la que dependa y un atributo propio que implementará el atributo calificador del objeto débil.

ii. Implementar las relaciones de acuerdo a las siguientes reglas de transformación

- Dos objetos y una relación (relación binaria)

Conectividad 1 a 1: Incluir como atributo de la tabla que represente al objeto más importante para el sistema, el identificador del otro objeto (clave foránea). (Alternativamente, cada tabla podría tener como clave foránea la clave primaria de la otra).

Conectividad 1 a 1c: En la tabla de conectividad 1c debe incluirse la clave primaria de la de conectividad 1. (Si se ha optado por incluir en cada tabla la clave de la otra, en la de conectividad 1 la clave foránea deberá permitir valores nulos).

Conectividad 1c a 1c: En este caso, ambas tablas podrían contener la clave de la otra, con valores nulos permitidos. En el caso que se decidiera incluir la clave foránea en una sola de las tablas, también deberá considerarse que tomará valores nulos.

Conectividades 1 a N, 1 a Nc y 1c a Nc: En todos los casos se debe incluir en la tabla que implemente el objeto de conectividad N (o Nc) la clave primaria de la tabla de conectividad 1; si la relación es 1c a Nc, la clave foránea podrá tomar valores nulos.

Conectividad M a N: Definir una tabla (llamada de correlación) por cada relación de este tipo. Sus columnas serán las claves primarias de las tablas a asociar. (En el caso de presentarse condicionalidad, se deberá hacer lo mismo, pero considerando que no necesariamente habrá filas en la tabla de correlación para todas las combinaciones posibles entre las instancias de cada una de las tablas relacionadas).

- Un objeto y una relación (relación unaria)

Conectividades 1 a 1, 1 a 1c y 1c a 1c: En todos los casos se requiere incluir una clave foránea, que podrá tomar valores nulos cuando la conectividad sea 1:1c o 1c:1c.

· N objetos y una relación de grado “n” (relación n-aria): Generar una tabla a partir de la relación, cuyas columnas serán los atributos de la misma. Para determinar que atributos serán parte de la clave de la nueva tabla deben seguirse las siguientes reglas:

-Si todos los objetos que participan en la relación tienen conectividad N, todas las columnas de la tabla serán parte de la clave primaria;

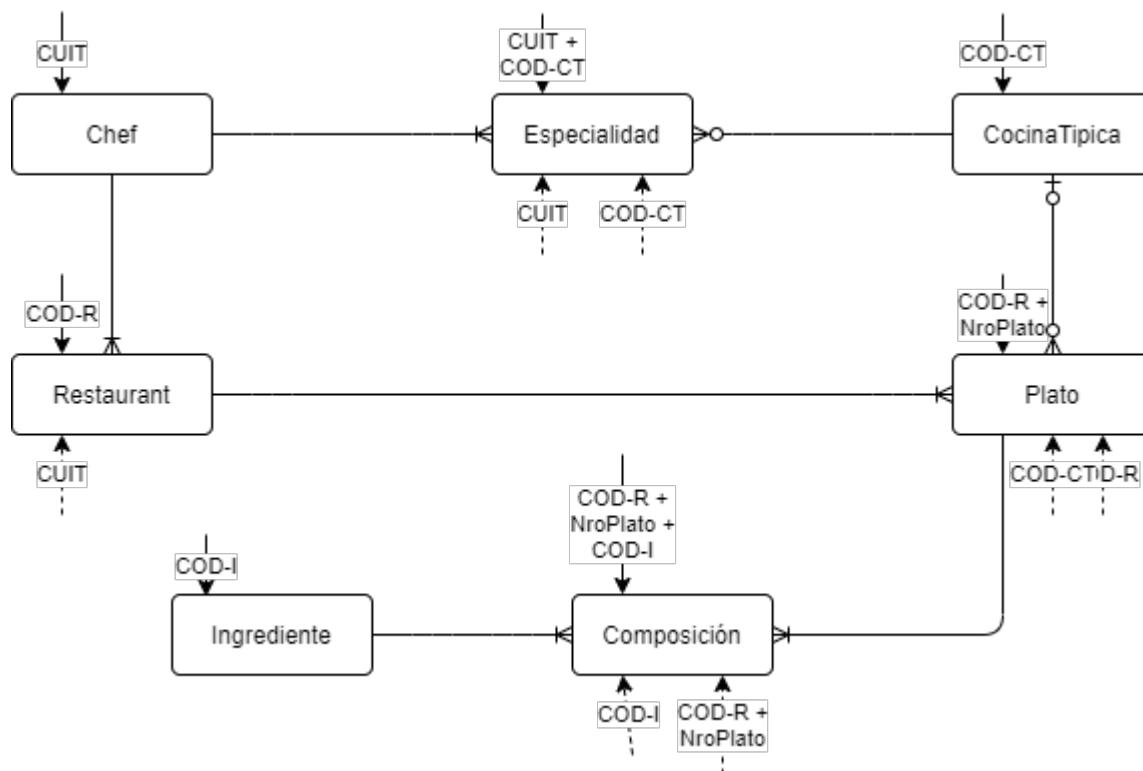
-Si uno o más de los objetos tiene conectividad 1, la clave primaria estará formada únicamente por las columnas que correspondan a los identificadores de todos los objetos de conectividad N.

-Si todos los objetos tienen conectividad 1, habrá tantas claves primarias candidatas como sea el grado de la relación; de todas ellas deberá seleccionarse una.



(En el caso de presentarse condicionalidad, se deberá hacer lo mismo, pero considerando que algunas de las columnas podrán tomar valores nulos. En el caso que dichas columnas formen parte de la clave, deberá estudiarse la forma de implementarlo en el DBMS en particular que se esté utilizando, ya que no es posible que una clave primaria tome valores nulos, según el modelo relacional).

- b. Normalizar hasta la tercera forma normal
- c. Construir la matriz de procesamiento
- d. Agregar atributos que sean acumulados (totales por período de tiempo, por entidad, etc.)



DICCIONARIO DE DATOS

CHEF	=	Nombre + @CUIT
COCINA TIPICA	=	@COD-CT + Tipo
COMPOSICION	=	@COD-R + @NroPlato + @COD-I + Cantidad + Modo
ESPECIALIDAD	=	@CUIT + @COD-CT
INGREDIENTE	=	@COD-I + Nombre + Unidad-Medida + Tipo
PLATO	=	@COD-R + @NroPlato + Nombre + Precio + COD-CT
RESTAURANTE	=	@COD-R + Nombre + Categoría + Calle + Nro + Localidad + CUIT



NORMALIZACIÓN

BREVE RESEÑA HISTÓRICA

El creador del proceso de normalización de Bases de Datos fue E. F. Codd quien entre 1968 y 1970 desarrolló los conceptos básicos del "modelo relacional" de Bases de Datos.

Posteriormente entre 1970 y 1973 E. Codd escribió varios artículos referidos a las leyes de Armstrong sobre el álgebra relacional. Todo este proceso de investigación finalizó en el desarrollo del Proyecto "Ingres", considerado una de las primeras bases de datos basada en la concepción del modelo relacional.

Para entonces E. Codd había formulado las 3 Formas Normales (FN) Básicas (objeto de nuestro estudio), a las cuales se sumaron con posterioridad 2 Formas Normales más postuladas por Fagin y una redefinición de la 3FN a la cual se llamó la Forma Normal de Boyce-Codd.

En la práctica las 3 últimas Formas Normales mencionadas, solo se aplican para fines académicos ya que es muy difícil que se presenten, por ser excepciones particulares de las 3 Formas Normales Básicas.

DEFINICIÓN:

Los datos provenientes del análisis de un sistema, volcados al Diccionario de Datos son propensos a discurrir en una maraña de repeticiones y redundancias innecesarias. El grado de complejidad puede crecer sin límites previsibles. Es posible evitarlos, recurriendo a un proceso denominado "normalización". Este proceso transforma las complejas presentaciones de los usuarios y de los almacenamientos de datos, en conjuntos estables de estructuras de datos de menor tamaño. Por ser más pequeñas y estables, son fáciles de mantener.

TEORÍA DE LA NORMALIZACIÓN

Dado un conjunto A de atributos y el conjunto D de dependencias existentes entre ellos que constituyen un esquema de relación R(A, D), se trata de transformar, por medio de sucesivas proyecciones, este esquema de partida en un conjunto de n esquemas de relación $\{R_i (A_i, D_i)\}_{i=1}^n$, tales que cumplan unas determinadas condiciones:

- El conjunto de esquemas R_i deberán ser **equivalentes** a R y **mejores** que el esquema de partida:
 - o conservación de la información
 - o conservación de las dependencias
 - o mínima redundancia de los datos (normalización de las relaciones)

CONSERVACIÓN DE LA INFORMACIÓN

Son precisas dos condiciones

1) Conservación de los atributos

El conjunto de atributos de los esquemas resultantes ha de ser igual al conjunto de atributos del esquema origen

2) Conservación del contenido

Para toda extensión r de R, la combinación (join) de las relaciones resultantes r_i ha de producir la relación origen r.

Se dice que la descomposición se ha realizado "sin pérdida de información" (SPI)

- Descomposición en **proyecciones independientes**



- Principios de Rissanen (1976) para saber si una descomposición conserva la información y las dependencias funcionales:
 - Sea R una relación y R1 y R2 dos de sus proyecciones, se dice que dichas proyecciones son independientes si y sólo si,
 - Sus atributos comunes son la clave primaria de, al menos, una relación.
 - Cada dependencia funcional en R puede deducirse de las de R1 y R2
- Hasta 3FN siempre es posible encontrar una descomposición en proyecciones independientes

CONSERVACIÓN DE LAS DEPENDENCIAS

El conjunto de DF de partida debe ser equivalente al conjunto de DF de los esquemas resultantes

En el proceso de transformación del esquema origen R (A, DF) en un conjunto de esquemas Ri (Ai, DFi),

se han conservado las dependencias si se cumple que $(\bigcup_{i=1}^n DFi)^+ = DF^+$

DEPENDENCIAS

- son propiedades inherentes al contenido semántico de los datos;
- son un tipo especial de restricción de usuario en el modelo relacional, que afecta únicamente a los atributos dentro de una única relación; y
- se han de cumplir para cualquier extensión de un esquema de relación.

A fines de simplificación, se considera que un esquema de relación es un par de la forma:

R (A, DEP)

– donde:

- A es el conjunto de atributos de la relación, y
- DEP es el conjunto de dependencias existentes entre los atributos.
- Existen distintos tipos de dependencias:
 - Funcionales (DF),
 - Multivaluadas (DM),
 - Jerárquicas (DJ), y
 - de Combinación (DC) (también llamadas producto).

Cada tipo de dependencia se caracteriza por ser una asociación particular entre los datos.

El grupo más restrictivo (y también más numeroso) es el de las dependencias funcionales. Sobre este conjunto de dependencias, se apoyan las tres primeras formas normales y la forma normal de Boyce-Codd.



DEFINICIÓN DE DF:

- Sea el esquema de relación $R(A, DF)$ y sean X e Y dos descriptores (subconjuntos de atributos de A). Se dice que existe una DF entre X e Y , de forma que X determina a Y , si y sólo si se cumple que para cualesquier dos tuplas de R , u y v tales que $u[X] = v[X]$, entonces necesariamente $u[Y] = v[Y]$.

– Esto significa que a cada valor x del atributo X , le corresponde un único valor y del atributo Y .

- Determinante:

– Un **determinante** o implicante es un conjunto de atributos del que depende funcionalmente otro conjunto de atributos al que llamamos determinado o **implicado**.

- Ejemplo:

– El código de estudiante determina el nombre del mismo:

Cód_Estudiante → Nombre

DESCRIPTORES EQUIVALENTES:

– Dos descriptores X e Y se dice que son equivalentes si $X \rightarrow Y$ y $X \leftarrow Y$

– también se puede representar como: $X \leftrightarrow Y$

- Ejemplo:

– Los atributos *Cód_Estudiante* y *DNI* son equivalentes (se supone que dos alumnos distintos no pueden tener ni el mismo código ni el mismo DNI), es decir:

Cód_Estudiante ↔ DNI

OBJETIVOS DE LA NORMALIZACIÓN:

Las interrelaciones entre las tablas mantienen información diferente relacionada con toda exactitud.

Los accesos para consultar y manipular la información son más sencillos.

Al estar los datos agrupados en tablas, la manera de identificación de un objeto o relación es prácticamente intuitiva.

La información no estará duplicada innecesariamente dentro de las estructuras o lo que es lo mismo habrá mínima redundancia.



EJEMPLO

El modelo lógico de un sistema cuenta con un almacenamiento denominado "Facturas" que contiene los siguientes elementos de datos:

N.º de Factura
Fecha de factura
N.º de pedido
Código de vendedor
Nombre de vendedor
Código de Cliente
Nombre cliente
Domicilio cliente
Localidad
Código postal
N.º CUIT cliente
Posición frente al IVA cliente
N.º ISIB cliente
Condición de pago
{Código artículo
Descripción artículo
Cantidad vendida artículo
Precio unitario artículo}
Importe neto factura



1FN – 1º FORMA NORMAL:

La primera etapa del proceso de normalización, primera forma normal (1FN), incluye la eliminación de grupos repetitivos y la identificación de la clave principal de una relación, entendiéndose por tal a aquel o aquellos elementos de datos que identifica una y sólo una ocurrencia de la relación. Luego se determinará qué relación tiene cada elemento de dato restante con la mencionada clave: uno a uno (1:1) o uno a muchos (1:N)

En el ejemplo que nos ocupa el elemento de dato que define a una factura, o sea, su clave principal, es el Nro. de Factura. Si se analiza Nro. de Factura con los restantes elementos de la estructura de datos vemos que el mismo tiene una relación 1:N con Código de artículo, Descripción artículo, Cantidad vendida artículo y Precio unitario artículo. Estos datos conforman un grupo repetitivo porque cada factura puede contener más de un artículo vendido y cada uno de esos artículos tiene su respectiva descripción, cantidad vendida y precio unitario

Aquellos datos que tengan una relación 1:N con la clave principal, deberán ser separados en una nueva relación. A la nueva estructura obtenida, que en este caso representan los artículos vendidos en cada factura, se le agregará el campo clave de la estructura original para mantener la relación. Luego se determinará la clave principal y se le aplicará, nuevamente, el procedimiento de 1FN para tener la seguridad de que no contenga otros grupos repetitivos.

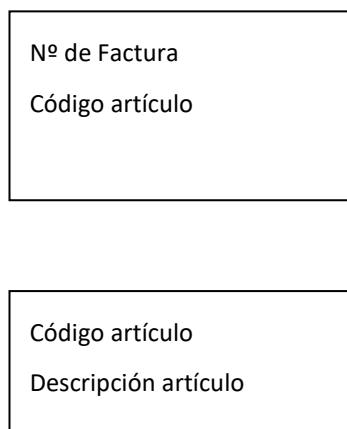
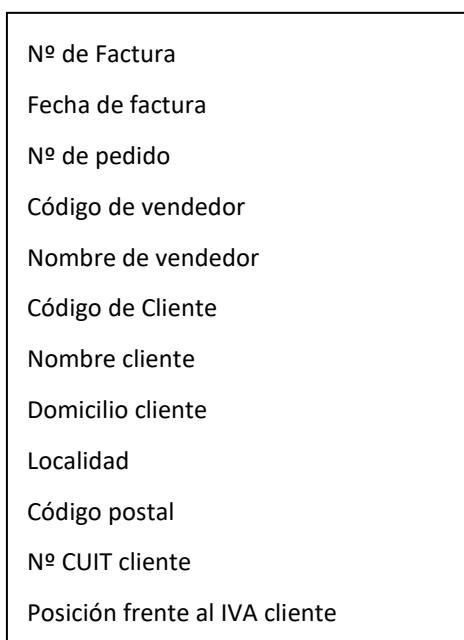
En este ejemplo, la clave principal de la nueva relación está compuesta por dos elementos de datos (Clave Compuesta) porque se necesitan 2 elementos para definir un artículo comprado en una factura

N.º de Factura	N.º de Factura
Fecha de factura	Código artículo
N.º de pedido	Descripción artículo
Código de vendedor	Cantidad vendida artículo
Nombre de vendedor	Precio unitario artículo
Código de Cliente	
Nombre cliente	
Domicilio cliente	
Localidad	
Código postal	
N.º CUIT cliente	
Posición frente al IVA cliente	
N.º ISIB cliente	
Condición de pago	
Importe neto factura	



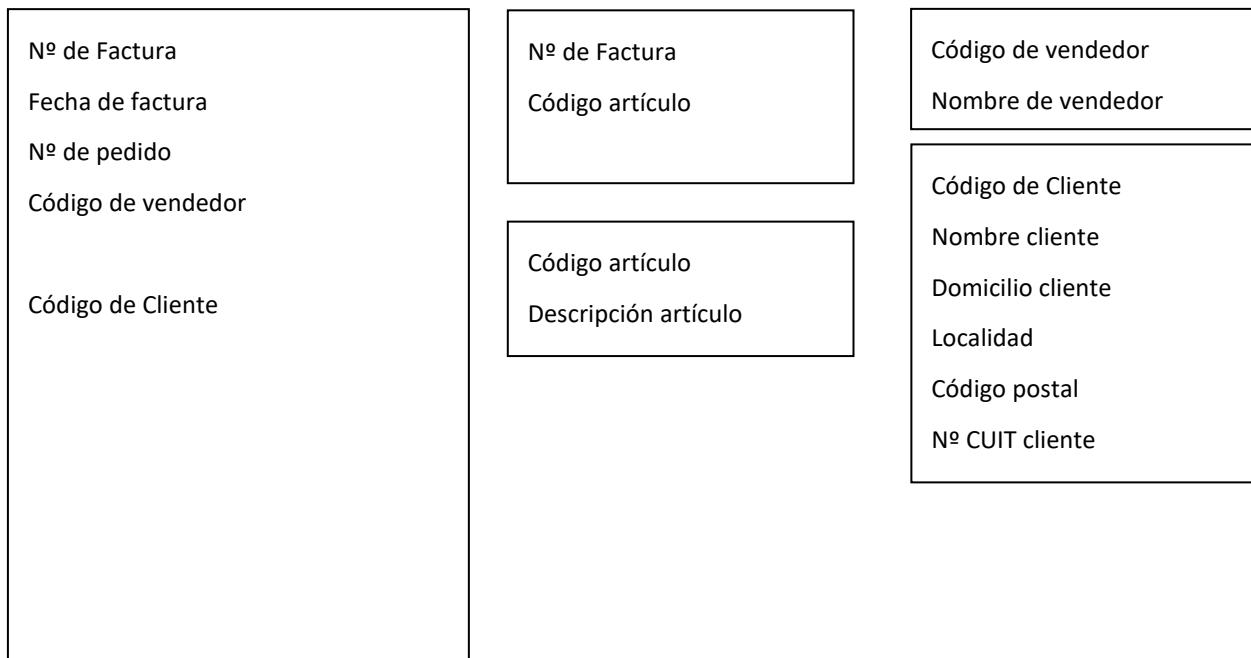
2FN – 2º FORMA NORMAL

El análisis de la segunda forma normal (2FN) parte de suponer que se cumple con 1FN y consiste en verificar, en aquellas relaciones cuya clave principal sea compuesta, si existe algún dato que tenga una relación biunívoca con parte de la mencionada clave (Dependencia Funcional Parcial). Se entiende por relación biunívoca los casos en que, si el dato clave varía hacia un determinado valor, al otro/otros datos en análisis, le corresponde siempre el mismo valor. En nuestro ejemplo: si Código de Artículo toma un valor determinado (A125), la descripción del artículo siempre será la misma (Tornillos) y su precio unitario siempre será el mismo (\$0,05). Estos elementos de datos conformarán una nueva relación (Artículos), que dará origen a una nueva tabla cuya clave principal será Código de Artículo.



3FN – 3º FORMA NORMAL

La tercera forma normal (3FN) parte de suponer que se cumple con 2FN, y consiste en analizar las relaciones para determinar si existe alguna o algunas relaciones biunívocas entre algunos elementos de datos de las mismas, con otro u otros elementos de la relación que no forman parte de la clave principal (Dependencia Funcional Transitiva). En el ejemplo existen relaciones biunívocas entre Código de Vendedor y Nombre del vendedor y entre Código de Cliente y todos los datos del cliente.



UNIDAD 04

EL ÁLGEBRA RELACIONAL

INTRODUCCIÓN

Esta unidad didáctica está dedicada al estudio de los lenguajes formales de consultas, en particular al álgebra relacional.

En esta unidad se analizan las operaciones del álgebra relacional, que sirven para hacer consultas a una base de datos. Es preciso conocer estas operaciones porque nos permiten saber qué servicios de consulta debe proporcionar un lenguaje relacional. Otra aportación del álgebra relacional es que facilita la comprensión de algunas de las construcciones del lenguaje SQL que se estudiarán en otra unidad didáctica de esta materia. Además, constituye la base para el estudio del tratamiento de las consultas que efectúan los SGBD internamente (especialmente en lo que respecta a la optimización de consultas). Este último tema queda fuera del ámbito de la presente materia, pero es relevante para estudios más avanzados sobre bases de datos.

OBJETIVOS

En los materiales didácticos de esta unidad encontraremos las herramientas indispensables para alcanzar los siguientes objetivos:

1. Conocer las operaciones del álgebra relacional.
2. Saber utilizar las operaciones del álgebra relacional para consultar una base de datos.



INTRODUCCIÓN AL ÁLGEBRA RELACIONAL

Como ya hemos comentado en el apartado dedicado a las operaciones del modelo relacional, el álgebra relacional se inspira en la teoría de conjuntos para especificar consultas en una base de datos relacional.

Para **especificar una consulta** en álgebra relacional, es preciso definir uno o más pasos que sirven para ir construyendo, mediante operaciones de álgebra relacional, una nueva relación que contenga los datos que responden a la consulta a partir de las relaciones almacenadas. Los lenguajes basados en el álgebra relacional son procedimentales, dado que los pasos que forman la consulta describen un procedimiento.

La visión que presentaremos es la de un lenguaje teórico y, por lo tanto, incluiremos sólo sus operaciones fundamentales, y no las construcciones que se podrían añadir a un lenguaje comercial para facilitar cuestiones como por ejemplo el orden de presentación del resultado, el cálculo de datos agregados, etc.

Una característica destacable de todas las operaciones del álgebra relacional es que tanto los operandos como el resultado son relaciones. Esta propiedad se denomina cierre relacional.

Las operaciones del álgebra relacional han sido clasificadas según distintos criterios; de todos ellos indicamos los tres siguientes:

- 1) Según se pueden expresar o no en términos de otras operaciones.
 - a) **Operaciones primitivas:** son aquellas operaciones a partir de las cuales podemos definir el resto. Estas operaciones son la unión, la diferencia, el producto cartesiano, la selección y la proyección.
 - b) **Operaciones no primitivas:** el resto de las operaciones del álgebra relacional que no son estrictamente necesarias, porque se pueden expresar en términos de las primitivas; sin embargo, las operaciones no primitivas permiten formular algunas consultas de forma más cómoda. Existen distintas versiones del álgebra relacional, según las operaciones no primitivas que se incluyen. Nosotros estudiaremos las operaciones no primitivas que se utilizan con mayor frecuencia: la intersección y la combinación.
- 2) Según el número de relaciones que tienen como operandos:
 - a) **Operaciones binarias:** son las que tienen dos relaciones como operandos. Son binarias todas las operaciones, excepto la selección y la proyección.
 - b) **Operaciones unarias:** son las que tienen una sola relación como operando. La selección y la proyección son unarias.
- 3) Según se parecen o no a las operaciones de la teoría de conjuntos:
 - a) **Operaciones conjuntistas:** son las que se parecen a las de la teoría de conjuntos. Se trata de la unión, la intersección, la diferencia y el producto cartesiano.
 - b) **Operaciones específicamente relacionales:** son el resto de las operaciones; es decir, la selección, la proyección y la combinación.

Como ya hemos comentado anteriormente, las operaciones del álgebra relacional obtienen como resultado una nueva relación. Es decir que si hacemos una operación del álgebra como por ejemplo *EMPLEADOS_ADM* U *EMPLEADOS_PROD*

Implicaciones del cierre relacional

El hecho de que el resultado de una operación del álgebra relacional sea una nueva relación tiene implicaciones importantes:

1. El resultado de una operación puede actuar como operando de otra operación.
2. El resultado de una operación cumplirá todas las características que ya conocemos de las relaciones: no-ordenación de las tuplas, ausencia de tuplas repetidas, etc.



para obtener la unión de las relaciones *EMPLEADOS_ADM* y *EMPLEADOS_PROD*, el resultado de la operación es una nueva relación que tiene la unión de las tuplas de las relaciones de partida.

Esta nueva relación debe tener un nombre. En principio, consideramos que su nombre es la misma expresión del álgebra relacional que la obtiene; es decir, la misma expresión *EMPLEADOS_ADM* U *EMPLEADOS_PROD*. Puesto que este nombre es largo, en ocasiones puede ser interesante cambiarlo por uno más simple. Esto nos facilitará las referencias a la nueva relación, y será especialmente útil en los casos en los que queramos utilizarla como operando de otra operación. Usaremos la operación auxiliar *asignar* con este objetivo.

La operación *asignar*, que denotaremos con el símbolo =, permite establecer un nombre R a la relación que resulta de una operación del álgebra relacional; lo hace de la forma siguiente:

$$R = E,$$

siendo E la expresión de una operación del álgebra relacional.

En el ejemplo, para dar el nombre *EMPLEADOS* a la relación resultante de la operación *EMPLEADOS_ADM* U *EMPLEADOS_PROD*, haríamos:

$$EMPLEADOS = EMPLEADOS_ADM \cup EMPLEADOS_PROD.$$

Cada operación del álgebra relacional da unos nombres por defecto a los atributos del esquema de la relación resultante, tal y como veremos más adelante.

En algunos casos, puede ser necesario cambiar estos nombres por defecto por otros nombres. Por este motivo, también permitiremos cambiar el nombre de la relación y de sus atributos mediante la operación *renombramiento*. Para la operación de renombramiento se utiliza el operador ρ (letra griega rho minúscula)

Utilizaremos también la operación renombrar para cambiar el esquema de una relación. Si una relación tiene el esquema $S(B_1, B_2, \dots, B_n)$ y queremos cambiarlo por $R(A_1, A_2, \dots, A_n)$, lo haremos de la siguiente forma:

$$R = \rho A_1 \leftarrow B_1, A_2 \leftarrow B_2, \dots, A_n \leftarrow B_n (S)$$

A continuación, presentaremos un ejemplo que utilizaremos para ilustrar las operaciones del álgebra relacional. Despues veremos con detalle las operaciones.

Supongamos que tenemos una base de datos relacional con las cuatro relaciones siguientes:

- 1) La relación *EDIFICIOS_EMP*, que contiene datos de distintos edificios de los que una empresa dispone para desarrollar sus actividades.
- 2) La relación *DESPACHOS*, que contiene datos de cada uno de los despachos que hay en los edificios anteriores.
- 3) La relación *EMPLEADOS_ADM*, que contiene los datos de los empleados de la empresa que llevan a cabo tareas administrativas.
- 4) La relación *EMPLEADOS_PROD*, que almacena los datos de los empleados de la empresa que se ocupan de tareas de producción.



A continuación, describimos los esquemas de las relaciones anteriores y sus extensiones en un momento determinado:

- Esquema y extensión de *EDIFICIOS_EMP*:

EDIFICIOS_EMP	
edificio	supmediadesp
Marina	15
Diagonal	10

- Esquema y extensión de *DESPACHOS*:

DESPACHOS		
edificio	número	superficie
Marina	120	10
Marina	230	20
Diagonal	120	10
Diagonal	440	10

- Esquema y extensión de *EMPLEADOS_ADMIN*:

EMPLEADOS_ADMIN				
DNI	nombre	apellido	edificiodesp	númerodesp
40.444.255	Juan	García	Marina	120
33.567.711	Marta	Roca	Marina	120

- Esquema y extensión de *EMPLEADOS_PROD*:

EMPLEADOS_PROD				
DNI	nombre	apellido	edificiodesp	númerodesp
33.567.711	Marta	Roca	Marina	120
55.898.425	Carlos	Buendía	Diagonal	120
77.232.144	Elena	Pla	Marina	230
21.335.245	Jorge	Soler	NULO	NULO
88.999.210	Pedro	González	NULO	NULO

Se considera que los valores nulos de los atributos *edificiodesp* y *númerodesp* de las relaciones *EMPLEADOS_PROD* y *EMPLEADOS_ADMIN* indican que el empleado correspondiente no tiene despacho.



OPERACIONES CONJUNTISTAS

Las operaciones conjuntistas del álgebra relacional son la **unión**, la **intersección**, la **diferencia** y el **producto cartesiano**.

UNIÓN

La unión es una operación que, a partir de dos relaciones, obtiene una nueva relación formada por todas las tuplas que están en alguna de las relaciones de partida.

La unión es una operación binaria, y la unión de dos relaciones T y S se indica
 $T \cup S$.

La unión de las relaciones $EMPLEADOS_ADM$ y $EMPLEADOS_PROD$ proporciona una nueva relación que contiene tanto a los empleados de administración como los empleados de producción; se indicaría así: $EMPLEADOS_ADM \cup EMPLEADOS_PROD$.

Sólo tiene sentido aplicar la unión a relaciones que tengan tuplas similares.

Por ejemplo, se puede hacer la unión de las relaciones $EMPLEADOS_ADM$ y $EMPLEADOS_PROD$ porque sus tuplas se parecen. En cambio, no se podrá hacer la unión de las relaciones $EMPLEADOS_ADM$ y $DESPACHOS$ porque, como pueden observar en las tablas, las tuplas respectivas son de tipo diferente.

Más concretamente, para poder aplicar la unión a dos relaciones, es preciso que las dos relaciones sean compatibles. Decimos que dos relaciones T y S son **relaciones compatibles** si:

- Tienen el mismo grado.
- Se puede establecer una biyección entre los atributos de T y los atributos de S que hace corresponder a cada atributo A_i de T un atributo A_j de S , de modo que se cumple que $\text{dominio}(A_i) = \text{dominio}(A_j)$.

EJEMPLO DE RELACIONES COMPATIBLES

Las relaciones $EMPLEADOS_ADM$ y $EMPLEADOS_PROD$ tienen grado 5. Podemos establecer la siguiente biyección entre sus atributos:

- A *DNI* de $EMPLEADOS_ADM$ le corresponde *DNIemp* de $EMPLEADOS_PROD$.
- A *nombre* de $EMPLEADOS_ADM$ le corresponde *nombreemp* de $EMPLEADOS_PROD$.
- A *apellido* de $EMPLEADOS_ADM$ le corresponde *apellidoemp* de $EMPLEADOS_PROD$.
- A *edificiodesp* de $EMPLEADOS_ADM$ le corresponde *edificiodesp* de $EMPLEADOS_PROD$.
- A *númerodesp* de $EMPLEADOS_ADM$ le corresponde *edificiodesp* de $EMPLEADOS_PROD$.

Además, supondremos que los dominios de sus atributos se han declarado de forma que se cumple que el dominio de cada atributo de $EMPLEADOS_ADM$ sea el mismo que el dominio de su atributo correspondiente en $EMPLEADOS_PROD$.

Por todos estos factores, podemos llegar a la conclusión de que $EMPLEADOS_ADM$ y $EMPLEADOS_PROD$ son relaciones compatibles.

A continuación, pasaremos a definir los atributos y la extensión de la relación resultante de una unión.



Los atributos del esquema de la relación resultante de $T \cup S$ coinciden con los atributos del esquema de la relación T .

La extensión de la relación resultante de $T \cup S$ es el conjunto de tuplas que pertenecen a la extensión de T , a la extensión de S o a la extensión de ambas relaciones.

No-repetición de tuplas

Noten que en caso de que una misma tupla esté en las dos relaciones que se unen, el resultado de la unión no la tendrá repetida. El resultado de la unión es una nueva relación por lo que no puede tener repeticiones de tuplas.

EJEMPLO DE UNIÓN

Si queremos obtener una relación R que tenga a todos los empleados de la empresa del ejemplo anterior, llevaremos a cabo la unión de las relaciones *EMPLEADOS_ADM* y *EMPLEADOS_PROD* de la forma siguiente:

$$R = \text{EMPLEADOS_ADM} \cup \text{EMPLEADOS_PROD}.$$

Entonces la relación R resultante será la reflejada en la tabla siguiente:

R				
DNI	nombre	apellido	edificiodesp	númerodesp
40.444.255	Juan	García	Marina	120
33.567.711	Marta	Roca	Marina	120
55.898.425	Carlos	Buendía	Diagonal	120
77.232.144	Elena	Pla	Marina	230
21.335.245	Jorge	Soler	NULO	NULO
88.999.210	Pedro	González	NULO	NULO

El hecho de que los atributos de la relación resultante coincidan con los atributos de la relación que figura en primer lugar en la unión es una convención; teóricamente, también habría sido posible convenir que coincidiesen con los de la relación que figura en segundo lugar.

INTERSECCIÓN

La intersección es una operación que, a partir de dos relaciones, obtiene una nueva relación formada por las tuplas que pertenecen a las dos relaciones de partida.

La intersección es una operación binaria; la intersección de dos relaciones T y S se indica $T \cap S$.

La intersección de las relaciones *EMPLEADOS_ADM* y *EMPLEADOS_PROD* obtiene una nueva relación que incluye a los empleados que son al mismo tiempo de administración y de producción: se indicaría como $\text{EMPLEADOS_ADM} \cap \text{EMPLEADOS_PROD}$.

La intersección, como la unión, sólo se puede aplicar a relaciones que tengan tuplas similares. Para poder hacer la intersección de dos relaciones, es preciso, pues, que las relaciones sean compatibles.



A continuación, definiremos los atributos y la extensión de la relación resultante de una intersección.

Los atributos del esquema de la relación resultante de $T \cap S$ coinciden con los atributos del esquema de la relación T.

La extensión de la relación resultante de $T \cap S$ es el conjunto de tuplas que pertenecen a la extensión de ambas relaciones.

EJEMPLO DE INTERSECCIÓN

Si queremos obtener una relación R que incluya a todos los empleados de la empresa del ejemplo que trabajan tanto en administración como en producción, realizaremos la intersección de las relaciones *EMPLEADOS_ADM* y *EMPLEADOS_PROD* de la forma siguiente:

$$R = \text{EMPLEADOS_ADM} \cap \text{EMPLEADOS_PROD}.$$

Entonces, la relación R resultante será:

R				
DNI	nombre	apellido	edificiodesp	númerodesp
33.567.711	Marta	Roca	Marina	120

Observen que se ha tomado la convención de que los atributos de la relación que resulta coincidan con los atributos de la relación que figura en primer lugar.

DIFERENCIA

La diferencia es una operación que, a partir de dos relaciones, obtiene una nueva relación formada por todas las tuplas que están en la primera relación y, en cambio, no están en la segunda. La diferencia es una operación binaria, y la diferencia entre las relaciones T y S se indica como $T - S$.

La diferencia *EMPLEADOS_ADM* menos *EMPLEADOS_PROD* da como resultado una nueva relación que contiene a los empleados de administración que no son empleados de producción, y se indicaría de este modo: *EMPLEADOS_ADM – EMPLEADOS_PROD*.

La diferencia, como ocurría en la unión y la intersección, sólo tiene sentido si se aplica a relaciones que tengan tuplas similares. Para poder realizar la diferencia de dos relaciones es necesario que las relaciones sean compatibles.

A continuación, definimos los atributos y la extensión de la relación resultante de una diferencia.

Los atributos del esquema de la relación resultante de $T - S$ coinciden con los atributos del esquema de la relación T.

La extensión de la relación resultante de $T - S$ es el conjunto de tuplas que pertenecen a la extensión de T, pero no a la de S.



EJEMPLO DE DIFERENCIA

Si queremos obtener una relación R con todos los empleados de la empresa del ejemplo que trabajan en administración, pero no en producción, haremos la diferencia de las relaciones $EMPLEADOS_ADM$ y $EMPLEADOS_PROD$ de la forma siguiente:

$$R = EMPLEADOS_ADM - EMPLEADOS_PROD$$

Entonces la relación R resultante será:

R				
DNI	nombre	apellido	edificiodesp	númerodesp
40.444.255	Juan	García	Marina	120

Se ha tomado la convención de que los atributos de la relación resultante coincidan con los atributos de la relación que figura en primer lugar.

PRODUCTO CARTESIANO

El producto cartesiano es una operación que, a partir de dos relaciones, obtiene una nueva relación formada por todas las tuplas que resultan de concatenar tuplas de la primera relación con tuplas de la segunda.

El producto cartesiano es una operación binaria. Siendo T y S dos relaciones que cumplen que sus esquemas no tienen ningún nombre de atributo común, el producto cartesiano de T y S se indica como $T \times S$.

Si calculamos el producto cartesiano de $EDIFICIOS_EMP$ y $DESPACHOS$, obtendremos una nueva relación que contiene todas las concatenaciones posibles de tuplas de $EDIFICIOS_EMP$ con tuplas de $DESPACHOS$.

Si se quiere calcular el producto cartesiano de dos relaciones que tienen algún nombre de atributo común, sólo hace falta redenominar previamente los atributos adecuados de una de las dos relaciones.

A continuación, definimos los atributos y la extensión de la relación resultante de un producto cartesiano.

Los atributos del esquema de la relación resultante de $T \times S$ son todos los atributos de T y todos los atributos de S ¹⁹.

La extensión de la relación resultante de $T \times S$ es el conjunto de todas las tuplas de la forma $\langle v_1, v_2, \dots, v_n, w_1, w_2, \dots, w_m \rangle$ para las que se cumple que $\langle v_1, v_2, \dots, v_n \rangle$ pertenece a la extensión de T y que $\langle w_1, w_2, \dots, w_m \rangle$ pertenece a la extensión de S .

EJEMPLO DE PRODUCTO CARTESIANO

El producto cartesiano de las relaciones $DESPACHOS$ y $EDIFICIOS_EMP$ del ejemplo se puede hacer como se indica (es necesario redenominar atributos previamente):

¹⁹ Recuerden que T y S no tienen ningún nombre de atributo común.



EDIFICIOS = ρ edificio->nombreedificio EDIFICIOS_EMP

$R = EDIFICIOS \times DESPACHOS$

Entonces, la relación R resultante será:

R				
nombreedificio	supmediadesp	edificio	número	superficie
Marina	15	Marina	120	10
Marina	15	Marina	230	20
Marina	15	Diagonal	120	10
Marina	15	Diagonal	440	10
Diagonal	10	Marina	120	10
Diagonal	10	Marina	230	20
Diagonal	10	Diagonal	120	10
Diagonal	10	Diagonal	440	10

Conviene señalar que el producto cartesiano es una operación que raramente se utiliza de forma explícita, porque el resultado que da no suele ser útil para resolver las consultas habituales.

A pesar de ello, el producto cartesiano se incluye en el álgebra relacional porque es una operación primitiva; a partir de la cual se define otra operación del álgebra, la combinación, que se utiliza con mucha frecuencia.

OPERACIONES ESPECÍFICAMENTE RELACIONALES

Las operaciones específicamente relacionales son la **selección**, la **proyección** y la **combinación**.

SELECCIÓN

Podemos ver la selección como una operación que sirve para elegir algunas tuplas de una relación y eliminar el resto. Más concretamente, la selección es una operación que, a partir de una relación, obtiene una nueva relación formada por todas las tuplas de la relación de partida que cumplen una condición de selección especificada.

La selección es una operación unaria.

Siendo C una condición de selección, la selección de T con la condición C se indica utilizando la letra griega sigma minúscula como $\sigma C(T)$.

Para obtener una relación que tenga todos los despachos del edificio Marina que tienen más de 12 metros cuadrados, podemos aplicar una selección a la relación *DESPACHOS* con una condición de selección que sea $edificio = \text{Marina}$ y $superficie > 12$; se indicaría $\sigma edificio = \text{Marina} \wedge superficie > 12$ (*DESPACHOS*).

En general, la condición de selección C está formada por una o más cláusulas de la forma:

$A_i \theta v_i$,



o bien:

$A_i \theta A_j$,

donde A_i y A_j son atributos de la relación T , θ es un operador de comparación²⁰ y v es un valor. Además, se cumple que:

- En las cláusulas de la forma $A_i \theta v$, v es un valor del dominio de A_i .
- En las cláusulas de la forma $A_i \theta A_j$, A_i y A_j tienen el mismo dominio.

Las cláusulas que forman una condición de selección se conectan con los siguientes operadores booleanos: "y" (\wedge) y "o" (\vee).

A continuación, definimos los atributos y la extensión de la relación resultante de una selección.

Los atributos del esquema de la relación resultante de $\sigma_C(T)$ coinciden con los atributos del esquema de la relación T .

La extensión de la relación resultante de $\sigma_C(T)$ es el conjunto de tuplas que pertenecen a la extensión de T y que satisfacen la condición de selección C .

Una tupla t satisface una condición de selección C si, después de sustituir cada atributo que hay en C por su valor en t , la condición C se evalúa en el valor cierto.

EJEMPLO DE SELECCIÓN

Si queremos obtener una relación R con los despachos de la base de datos del ejemplo que están en el edificio Marina y que tienen una superficie de más de 12 metros cuadrados, haremos la siguiente selección:

$$R = \sigma(\text{edificio} = \text{'Marina'} \wedge \text{superficie} > 12) (\text{DESPACHOS})$$

La relación R resultante será:

R		
edificio	número	superficie
Marina	230	20

PROYECCIÓN

Podemos considerar la proyección como una operación que sirve para elegir algunos atributos de una relación y eliminar el resto. Más concretamente, la proyección es una operación que, a partir de una relación, obtiene una nueva relación formada por todas las (sub)tuplas de la relación de partida que resultan de eliminar unos atributos especificados.

²⁰ Es decir, $=, \neq, <, \leq, >, \geq$



La proyección es una operación unaria. Siendo $\{A_i, A_j, \dots, A_k\}$ un subconjunto de los atributos del esquema de la relación T, la proyección de T sobre $\{A_i, A_j, \dots, A_k\}$ se indica utilizando la letra griega pi como $\pi_{A_i, A_j, \dots, A_k}(T)$.

Para obtener una relación que tenga sólo los atributos *nombre* y *apellido* de los empleados de administración, podemos hacer una proyección en la relación *EMPLEADOS_ADM* sobre estos dos atributos. Se indicaría de la forma siguiente:

$\pi_{\text{nombre}, \text{apellido}}(\text{EMPLEADOS_ADM})$

A continuación, definiremos los atributos y la extensión de la relación resultante de una proyección.

Los atributos del esquema de la relación resultante de $\pi_{A_i, A_j, \dots, A_k}(T)$ son los atributos $\{A_i, A_j, \dots, A_k\}$.

La extensión de la relación resultante de $\pi_{A_i, A_j, \dots, A_k}(T)$ es el conjunto de todas las tuplas de la forma $\langle t.A_i, t.A_j, \dots, t.A_k \rangle$, donde se cumple que t es una tupla de la extensión de T y donde $t.A_p$ denota el valor para el atributo A_p de la tupla t.

Eliminación de las tuplas repetidas

Noten que la proyección elimina implícitamente todas las tuplas repetidas. El resultado de una proyección es una relación válida y no puede tener repeticiones de tuplas.

EJEMPLO DE PROYECCIÓN

Si queremos obtener una relación R con el nombre y el apellido de todos los empleados de administración de la base de datos del ejemplo, haremos la siguiente proyección:

$R = \pi_{\text{nombre}, \text{apellido}}(\text{EMPLEADOS_ADM})$

Entonces, la relación R resultante será:

R	
nombre	apellido
Juan	García
Marta	Roca

COMBINACIÓN

La combinación es una operación que, a partir de dos relaciones, obtiene una nueva relación formada por todas las tuplas que resultan de concatenar tuplas de la primera relación con tuplas de la segunda, y que cumplen una condición de combinación especificada.

La combinación es una operación binaria. Siendo T y S dos relaciones cuyos esquemas no tienen ningún nombre de atributo común, y siendo B una condición de combinación, la combinación de T y S según la condición B se indica $T \bowtie_B S$.



Para conseguir una relación que tenga los datos de cada uno de los empleados de administración junto con los datos de los despachos donde trabajan, podemos hacer una combinación de las relaciones *EMPLEADOS_ADM* y *DESPACHOS*, donde la condición de combinación indique lo siguiente: *edificiodesp = edificio* y *númerodesp = número*. La condición de combinación hace que el resultado sólo combine los datos de un empleado con los datos de un despacho si el *edificiodesp* y el *númerodesp* del empleado son iguales que el *edificio* y el *número* del despacho, respectivamente. Es decir, la condición hace que los datos de un empleado se combinen con los datos del despacho donde trabaja, pero no con datos de otros despachos.

La combinación del ejemplo anterior se indicaría de la forma siguiente:

EMPLEADOS_ADM \bowtie (*edificiodesp = edificio* \wedge *númerodesp = número*) *DESPACHOS*.

Si se quiere combinar dos relaciones que tienen algún nombre de atributo común, sólo hace falta redenominar previamente los atributos repetidos de una de las dos.

En general, la **condición B** de una combinación $T \bowtie B S$ está formada por una o más comparaciones de la forma

$A_i \theta A_j$,

donde A_i es un atributo de la relación T , A_j es un atributo de la relación S , θ es un operador de comparación ($=, \neq, <, \leq, >, \geq$), y se cumple que A_i y A_j tienen el mismo dominio.

A continuación, definimos los atributos y la extensión de la relación resultante de una combinación.

Los atributos del esquema de la relación resultante de $T \bowtie B S$ son todos los atributos de T y todos los atributos de S ²¹.

La extensión de la relación resultante de $T \bowtie B S$ es el conjunto de tuplas que pertenecen a la extensión del producto cartesiano $T \times S$ y que satisfacen todas las comparaciones que forman la condición de combinación B. Una tupla t satisface una comparación si, después de sustituir cada atributo que figura en la comparación por su valor en t , la comparación se evalúa al valor cierto.

EJEMPLO DE COMBINACIÓN

Supongamos que se desea encontrar los datos de los despachos que tienen una superficie mayor o igual que la superficie media de los despachos del edificio donde están situados. La siguiente combinación nos proporcionará los datos de estos despachos junto con los datos de su edificio (observen que es preciso redenominar previamente los atributos):

$EDIFICIOS = \rho_{\text{edificio} \rightarrow \text{nombreedificio}} EDIFICIOS_EMP$

$R = EDIFICIOS \bowtie (\text{nombreedificio} = \text{edificio} \wedge \text{supmediadesp} \leq \text{superficie}) DESPACHOS$

Entonces, la relación R resultante será:

²¹ Recuerden que T y S no tienen ningún nombre de atributo común.



R				
nombreedificio	supmediadesp	edificio	número	superficie
Marina	15	Marina	230	20
Diagonal	10	Diagonal	120	10
Diagonal	10	Diagonal	440	10

Supongamos ahora que, para obtener los datos de cada uno de los empleados de administración, junto con los datos del despacho donde trabajan, utilizamos la siguiente combinación:

$$R = \text{EMPLEADOS_ADM} \bowtie (\text{edificiodesp} = \text{edificio} \wedge \text{númerodesp} = \text{número}) \text{ DESPACHOS}$$

La relación R resultante será:

R							
DNI	nombre	apellido	edificiodesp	númerodesp	edificio	número	superficie
40.444.255	Juan	García	Marina	120	Marina	120	10
33.567.711	Marta	Roca	Marina	120	Marina	102	10

La relación R combina los datos de cada empleado con los datos de su despacho.

En ocasiones, la combinación recibe el nombre de **θ -combinación**, y cuando todas las comparaciones de la condición de la combinación tienen el operador “ $=$ ”, se denomina **equicombinación**.

Según esto, la combinación del último ejemplo es una equicombinación.

Observen que el resultado de una equicombinación siempre incluye una o más parejas de atributos que tienen valores idénticos en todas las tuplas.

En el ejemplo anterior, los valores de $edificiodesp$ coinciden con los de $edificio$, y los valores de $númerodesp$ coinciden con los de $número$.

Puesto que uno de cada par de atributos es superfluo, se ha establecido una variante de combinación denominada **combinación natural**, con el fin de eliminarlos.

La combinación natural de dos relaciones T y S se denota como $T \bowtie S$ y consiste básicamente en una equicombinación seguida de la eliminación de los atributos superfluos; además, se considera por defecto que la condición de combinación iguala todas las parejas de atributos que tienen el mismo nombre en T y en S .

Observen que, a diferencia de la equicombinación, la combinación natural se aplica a relaciones que tienen nombres de atributos comunes.

EJEMPLO DE COMBINACIÓN NATURAL

Si hacemos:

$$R = \text{EDIFICIOS_EMP} \bowtie \text{DESPACHOS}$$



se considera que la condición es $edificio = edificio$ porque $edificio$ es el único nombre de atributo que figura tanto en el esquema de *EDIFICIOS_EMP* como en el esquema de *DESPACHOS*.

El resultado de esta combinación natural es:

R			
edificio	supmediadesp	número	superficie
Marina	15	120	10
Marina	15	230	20
Diagonal	10	120	10
Diagonal	10	440	10

Noten que se ha eliminado uno de los atributos de nombre *edificio*.

En ocasiones, antes de la combinación natural es necesario aplicar la operación *redenombrar* para hacer coincidir los nombres de los atributos que nos interesa igualar.

EJEMPLO DE COMBINACIÓN NATURAL CON REDENOMINACIÓN

Por ejemplo, si queremos obtener los datos de cada uno de los empleados de administración junto con los datos del despacho donde trabajan, pero sin repetir valores de atributos superfluos, haremos la siguiente combinación natural, que requiere una redenominación previa:

$D = \rho_{edificio} \rightarrow edificiodesp, \rho_{número} \rightarrow númerodesp \text{ DESPACHOS}$

$R = EMPLEADOS_ADM \bowtie D$

Entonces, la relación *R* resultante será:

R					
DNI	nombre	apellido	edificiodesp	númerodesp	superficie
40.444.255	Juan	García	Marina	120	10
33.567.711	Marta	Roca	Marina	120	10

SECUENCIAS DE OPERACIONES DEL ÁLGEBRA RELACIONAL

En muchos casos, para formular una consulta en álgebra relacional es preciso utilizar varias operaciones, que se aplican en un cierto orden. Para hacerlo, hay dos posibilidades:

- 1) Utilizar una sola expresión del álgebra que incluya todas las operaciones con los paréntesis necesarios para indicar el orden de aplicación.



-
- 2) Descomponer la expresión en varios pasos donde cada paso aplique una sola operación y obtenga una relación intermedia que se pueda utilizar en los pasos subsiguientes.

EJEMPLO DE UTILIZACIÓN DE SECUENCIAS DE OPERACIONES

Para obtener el nombre y el apellido de los empleados, tanto de administración como de producción, es necesario hacer una unión de *EMPLEADOS_ADM* y *EMPLEADOS_PROD*, y después hacer una proyección sobre los atributos *nombre* y *apellido*. La operación se puede expresar de las formas siguientes:

- a) Se puede utilizar una sola expresión:

$$R = \pi \text{ } nombre, \text{ } apellido \text{ } (EMPLEADOS_ADM \cup EMPLEADOS_PROD)$$

- b) O bien podemos expresarlo en dos pasos:

- $EMPS = EMPLEADOS_ADM \cup EMPLEADOS_PROD$
- $R = \pi \text{ } nombre, \text{ } apellido \text{ } EMPS$

En los casos en que una consulta requiere efectuar muchas operaciones, resulta más sencilla la segunda alternativa, porque evita expresiones complejas.

OTROS EJEMPLOS DE CONSULTAS FORMULADAS CON SECUENCIAS DE OPERACIONES

Veamos algunos ejemplos de consultas en la base de datos formuladas con secuencias de operaciones del álgebra relacional.

- 1) Para obtener el nombre del edificio y el número de los despachos situados en edificios en los que la superficie media de estos despachos es mayor que 12, podemos utilizar la siguiente secuencia de operaciones:
 - $A = \sigma (\text{supmediadesp} > 12) \text{ EDIFICIOS_EMP}$
 - $B = DESPACHOS \bowtie A$
 - $R = \pi \text{ } edificio, \text{ } n\acuteumero \text{ } B$
- 2) Supongamos ahora que se desea obtener el nombre y el apellido de todos los empleados (tanto de administración como de producción) que están asignados al despacho 120 del edificio Marina. En este caso, podemos utilizar la siguiente secuencia:
 - $A = EMPLEADOS_ADM \cup EMPLEADOS_PROD$
 - $B = \sigma (\text{edificiodesp} = 'Marina' \wedge \text{númerodesp} = 120) \text{ A}$
 - $R = \pi \text{ } nombre, \text{ } apellido \text{ } B$
- 3) Si queremos consultar el nombre del edificio y el número de los despachos que ningún empleado de administración tiene asignado, podemos utilizar esta secuencia:
 - $A = \pi \text{ } edificio, \text{ } n\acuteumero \text{ } DESPACHOS$
 - $B = \pi \text{ } edificiodesp, \text{ } n\acuteumero \text{ } EMPLEADOS_ADM$
 - $R = A - B$.
- 4) Para obtener el DNI, el nombre y el apellido de todos los empleados de administración que tienen despacho, junto con la superficie de su despacho, podemos hacer lo siguiente:
 - $A = \rho \text{ } edificio <- \text{edificiodesp}, \text{ } n\acuteumero <- \text{númerodesp} (\pi \text{ } DNI, \text{ } nombre, \text{ } apellido, \text{ } edificiodesp, \text{ } n\acuteumero \text{ } desp \text{ } EMPLEADOS_ADM)$
 - $B = A \bowtie \text{DESPACHOS}$
 - $R = \pi \text{ } DNI, \text{ } nombre, \text{ } apellido, \text{ } superficie \text{ } B$

EXTENSIONES: COMBINACIONES EXTERNAS

Para finalizar el tema del álgebra relacional, analizaremos algunas extensiones útiles de la combinación.



Las combinaciones que se han descrito obtienen las tuplas del producto cartesiano de dos relaciones que satisfacen una condición de combinación. Las tuplas de una de las dos relaciones que no tienen en la otra relación una tupla como mínimo con la cual, una vez concatenadas, satisfagan la condición de combinación, no aparecen en el resultado de la combinación, y podríamos decir que sus datos se pierden.

Por ejemplo, si hacemos la siguiente combinación natural (con una redenominación previa):

$D = \rho_{\text{edificio}} \rightarrow \text{edificiodesp}$, $\text{número} \rightarrow \text{númerodesp}$ DESPACHOS

$R = \text{EMPLEADOS_ADM} \bowtie D$.

Puesto que se trata de una combinación natural, se considera que la condición de combinación es $\text{edificio} = \text{edificio}$ y $\text{número} = \text{número}$, y la relación R resultante será:

R					
DNIemp	nombreemp	apellidoemp	edificiodesp	númerodesp	superficie
33.567.711	Marta	Roca	Marina	120	10
55.898.425	Carlos	Buendía	Diagonal	120	10
77.232.144	Elena	Pla	Marina	230	20

Noten que en esta relación R no están los empleados de producción que no tienen despacho asignado (con valores nulos en edificiodesp y númerodesp), y tampoco los despachos que no tienen ningún empleado de producción, porque no cumplen la condición de combinación.

Conviene destacar que las tuplas que tienen un valor nulo para alguno de los atributos que figuran en la condición de combinación se pierden siempre, porque en estos casos la condición de combinación siempre se evalúa a falso.

En algunos casos, puede interesar hacer combinaciones de los datos de dos relaciones sin que haya pérdida de datos de las relaciones de partida. Entonces, se utilizan las combinaciones externas.

Las combinaciones externas entre dos relaciones T y S consisten en variantes de combinación que conservan en el resultado todas las tuplas de T , de S o de ambas relaciones. Pueden ser de los tipos siguientes:

- 1) La combinación externa izquierda entre dos relaciones T y S , que denotamos como $T \bowtie S$, conserva en el resultado todas las tuplas de la relación T .
- 2) La combinación externa derecha entre dos relaciones T y S , que denotamos como $T \bowtie L S$, conserva en el resultado todas las tuplas de la relación S .
- 3) Finalmente, la combinación externa plena entre dos relaciones T y S , que denotamos como $T \bowtie R S$, conserva en el resultado todas las tuplas de T y todas las tuplas de S .

Estas extensiones también se aplican al caso de la combinación natural entre dos relaciones, $T \bowtie S$, concretamente:



- a) La combinación natural externa izquierda entre dos relaciones T y S , que se indica como $T \bowtie S$, conserva en el resultado todas las tuplas de la relación T .
- b) La combinación natural externa derecha entre dos relaciones T y S , que se indica como $T \bowtie S$, conserva en el resultado todas las tuplas de la relación S .
- c) Finalmente, la combinación natural externa plena entre dos relaciones T y S , que se indica como $T \bowtie S$, conserva en el resultado todas las tuplas de T y todas las tuplas de S .

Las tuplas de una relación T que se conservan en el resultado R de una combinación externa con otra relación S , a pesar de que no satisfacen la condición de combinación, tienen valores nulos en el resultado R para todos los atributos que provienen de la relación S .

EJEMPLOS DE COMBINACIONES NATURALES EXTERNAS

- 1) Si hacemos la siguiente combinación natural derecha (con una redenominación previa):
- 2) $D = p \text{ edificio} \rightarrow \text{edificiodesp}$, número $\rightarrow \text{númerodesp}$ DESPACHOS

$$R = \text{EMPLADOS_PROD} \bowtie D$$

la relación R resultante será:

R					
DNIemp	nombreemp	apellidoemp	edificiodesp	númerodesp	superficie
33.567.711	Marta	Roca	Marina	120	10
55.898.425	Carlos	Buendía	Diagonal	120	10
77.232.144	Elena	Pla	Marina	230	20
NULO	NULO	NULO	Diagonal	440	10

Ahora obtenemos todos los despachos en la relación resultante, tanto si tienen un empleado de producción asignado como si no. Noten que los atributos DNI , $nombre$ y $apellido$ para los despachos que no tienen empleado reciben valores nulos.

- 3) Si hacemos la siguiente combinación natural izquierda (con una redenominación previa):

$$D = p \text{ edificio} \rightarrow \text{edificiodesp}, \text{número} \rightarrow \text{númerodesp} \text{ DESPACHOS}$$

$$R := \text{EMPLEADOS_PROD} \bowtie D,$$

entonces la relación R resultante será:

R					
DNIemp	nombreemp	apellidoemp	edificiodesp	númerodesp	superficie
33.567.711	Marta	Roca	Marina	120	10
55.898.425	Carlos	Buendía	Diagonal	120	10
77.232.144	Elena	Pla	Marina	230	20
21.335.245	Jorge	Soler	NULO	NULO	NULO
88.999.210	Pedro	González	NULO	NULO	NULO

Esta combinación externa nos permite obtener en la relación resultante a todos los empleados de producción, tanto si tienen despacho como si no. Observen que el atributo superficie para los empleados que no tienen despacho contiene un valor nulo.



4) Finalmente, si hacemos la siguiente combinación natural plena (con una redenominación previa):

5) $D = \rho \text{ edificio} \rightarrow \text{edificiodesp}$, $\text{número} \rightarrow \text{númerodesp}$ DESPACHOS

$R := \text{EMPLEADOS_PROD} \bowtie D$,

entonces la relación R resultante será:

R					
DNIemp	nombreemp	apellidoemp	edificiodesp	númerodesp	superficie
33.567.711	Marta	Roca	Marina	120	10
55.898.425	Carlos	Buendía	Diagonal	120	10
77.232.144	Elena	Pla	Marina	230	20
21.335.245	Jorge	Soler	NULO	NULO	NULO
88.999.210	Pedro	González	NULO	NULO	NULO
NULO	NULO	NULO	Diagonal	440	10

En este caso, en la relación resultante obtenemos a todos los empleados de producción y también todos los despachos.

OPERACIÓN DIVISIÓN

La operación división, denotada por \div , resulta adecuada para las consultas que incluyen la expresión “para todos”. Supóngase que se desea hallar a todos los clientes que tengan abierta una cuenta en todas las sucursales ubicadas en Arganzuela. Se pueden obtener todas las sucursales de Arganzuela mediante la expresión:

$$r1 = \pi \text{ nombre_sucursal} (\sigma \text{ ciudad_sucursal} = \text{'Arganzuela'} (\text{sucursal}))$$

Se pueden encontrar todos los pares $(\text{nombre_cliente}, \text{nombre_sucursal})$ para los que el cliente tiene una cuenta en una sucursal escribiendo

$$r2 = \pi \text{ nombre_cliente, nombre_sucursal} (\text{impositor} \bowtie \text{cuenta})$$

Ahora hay que hallar los clientes que aparecen en $r2$ con los nombres de *todas* las sucursales de $r1$. La operación que proporciona exactamente esos clientes es la operación división. La consulta se formula escribiendo

$$\begin{aligned} & \pi \text{ nombre_cliente, nombre_sucursal} (\text{impositor} \bowtie \text{cuenta}) \\ & \div \pi \text{ nombre_sucursal} (\sigma \text{ ciudad_sucursal} = \text{'Arganzuela'} (\text{sucursal})) \end{aligned}$$

Formalmente, sean $r(R)$ y $s(S)$ relaciones y $S \subset R$; es decir, todos los atributos del esquema S están también en el esquema R . La relación $r \div s$ es una relación del esquema $R - S$ (es decir, del esquema que contiene todos los atributos del esquema R que no están en el esquema S). Una tupla t está en $r \div s$ si y sólo si se cumplen estas dos condiciones:

1. t está en $\pi_{R-S}(r)$
2. Para cada tupla t_s de s hay una tupla t_r de r que cumple las dos condiciones siguientes:
 - a. $t_r[S] = t_s[S]$
 - b. $t_r[R - S] = t$

Puede resultar sorprendente descubrir que, dados una operación división y los esquemas de las relaciones, se pueda definir la operación división en términos de las operaciones fundamentales. Sean $r(R)$ y $s(S)$ dadas, con $S \subset R$:

$$r \div s = \pi_{R-S}(r) - \pi_{R-S}(\pi_{R-S}(r) \times s) - \pi_{R-S,S}(r)$$

Para comprobar que esta expresión es cierta, obsérvese que $\pi_{R-S}(r)$ proporciona todas las tuplas t que cumplen la primera condición de la definición de la división. La expresión del lado derecho del operador diferencia de conjuntos,

$$\pi_{R-S}(\pi_{R-S}(r) \times s) - \pi_{R-S,S}(r)$$



elimina las tuplas que no cumplen la segunda condición de la definición de la división. Esto se logra de la manera siguiente. Considérese $\pi_{R-S}(r) \times s$. Esta relación está en el esquema R y empareja cada tupla de $\pi_{R-S}(r)$ con cada tupla de s . La expresión $\pi_{R-S,S(r)}$ sólo reordena los atributos de r .

Por tanto, $(\pi_{R-S}(r) \times s) - \pi_{R-S,S(r)}$ genera los pares de tuplas de $\pi_{R-S}(r)$ y de s que no aparecen en r . Si una tupla t_j está en

$$\pi_{R-S}((\pi_{R-S}(r) \times s) - \pi_{R-S,S(r)})$$

existe alguna tupla t_s de s que no se combina con la tupla t_j para formar una tupla de r . Por tanto, t_j guarda un valor de los atributos $R - S$ que no aparece en $r \div s$. Estos valores son los que se eliminan de $\pi_{R-S}(r)$.

RESUMEN

En esta unidad didáctica hemos presentado los **conceptos fundamentales del modelo relacional de datos** y, a continuación, hemos explicado las **operaciones del álgebra relacional**:

- 1) Los aspectos más relevantes del modelo relacional que hemos descrito son los siguientes:
 - a. En lo que respecta a la **estructura de los datos**:
 - Consiste en un conjunto de relaciones.
 - Una relación permite almacenar datos relacionados entre sí.
 - La clave primaria de una relación permite identificar sus datos.
 - Las claves foráneas de las relaciones permiten referenciar claves primarias y, de este modo, establecer conexiones entre los datos de las relaciones.
 - b. En lo que respecta a la integridad de los datos:
 - La regla de integridad de unicidad y de entidad de la clave primaria: las claves primarias no pueden contener valores repetidos ni valores nulos.
 - La regla de integridad referencial: los valores de las claves foráneas deben existir en la clave primaria referenciada o bien deben ser valores nulos.
 - La regla de integridad de dominio: los valores no nulos de un atributo deben pertenecer al dominio del atributo, y los operadores que es posible aplicar sobre los valores dependen de los dominios de estos valores.
- 2) El álgebra relacional proporciona un conjunto de operaciones para manipular relaciones. Estas operaciones se pueden clasificar de la forma siguiente:
 - a. Operaciones conjuntistas: unión, intersección, diferencia y producto cartesiano.
 - b. Operaciones específicamente relacionales: selección, proyección y combinación.

Las operaciones del álgebra relacional pueden formar secuencias que permiten resolver consultas complejas.



UNIDAD 05

SQL

El modelo relacional fue propuesto por E. F. Codd en un artículo ya famoso "A relational model of data for large shared data banks"²². Desde ese momento se instituyeron varios proyectos de investigación con el propósito de construir sistemas de gestión de bases de datos relacionales. Entre estos proyectos podemos mencionar:

- Sistema R del IBM San José Research Laboratory.
- Ingres de la Universidad de California en Berkeley.
- Query-by-example del IBM T. J. Watson Research Center.
- PRTV (Peterlee Relational Test Vehicle) del IBM Científica Center en Peterlee, Inglaterra.

El lenguaje SQL se introdujo como lenguaje de consulta del Sistema R. Posteriormente, varios sistemas comerciales lo adoptaron como lenguaje para sus bases de datos.

El nombre SQL está formado por las iniciales de **Structured Query Language** (Lenguaje de consultas estructuradas).

Si bien el SQL está definido como un estándar, las bases de datos relacionales tienen sus extensiones propietarias y variaciones.

A continuación, se enumera las versiones del estándar a lo largo de los años, y sus principales características:

Año	Nombre	Alias	Comentarios
1986	SQL-86	SQL-87	Primera publicación hecha por ANSI. Confirmada por ISO en 1987.
1989	SQL-89	FIPS127-1	Revisión menor, el agregado más importante fueron las restricciones de integridad. Adoptado como FIPS 127-1.
1992	SQL-92	SQL2, FIPS 127-2	Revisión mayor (ISO 9075)
1999	SQL:1999	SQL3	Se agregaron expresiones regulares, consultas recursivas (para relaciones jerárquicas), triggers y algunas características orientadas a objetos.
2003	SQL:2003	SQL 2003	Introduce algunas características de XML, cambios en las funciones, estandarización del objeto sequence y de las columnas auto numéricas.
2006	SQL:2006	SQL 2006	ISO/IEC 9075-14:2006 Define las maneras en las cuales el SQL se puede utilizar conjuntamente con XML. Define maneras de importar y guardar datos XML en una base de datos SQL, manipulándolos dentro de la base de datos y publicando el XML y los datos SQL convencionales en forma XML. Además, proporciona facilidades que permiten a las aplicaciones integrar dentro de su código SQL el uso de XQuery, lenguaje de consulta XML publicado por el W3C (World Wide Web Consortium) para acceso concurrente a datos ordinarios SQL y documentos XML.
2008	SQL:2008	SQL 2008	Permite el uso de la cláusula ORDER BY fuera de las definiciones de los cursorios. Incluye los disparadores del tipo INSTEAD OF. Añade la sentencia TRUNCATE
2011	SQL:2011		Soporte para bases de datos temporales. Mejoras en las funciones de ventana y de la cláusula FETCH
2016	SQL:2016		Búsqueda de patrones, funciones de tabla polimórficas y compatibilidad con los ficheros JSON.
2019	SQL:2019		Arrays Multidimensionales (SQL/MDA)

²² "A relational model of data for large shared data banks" - Comm. ACM 13,6 - June 1970



Veremos como ejemplo dos sistemas de gestión de base de datos, con filosofías diferentes. Por un lado, veremos Microsoft SQL Server, que es un sistema propietario desarrollado y mantenido por Microsoft. Por otro lado veremos MySQL, que es desarrollado bajo licencia dual por Oracle Corporation, es decir que se distribuye básicamente de dos modos, bajo licencia GPL (General Public License), que es una licencia de software libre, y también bajo licencia comercial.

HISTORIA DE MICROSOFT SQL SERVER

Es el sistema de gestión de base de datos de Microsoft.

Microsoft comenzó el desarrollo de SQL Server en conjunto con Sybase Corporation para ser utilizado en la plataforma OS/2 de IBM. Con la salida de su sistema operativo de red (Windows NT Advanced Server) Microsoft decidió continuar con el desarrollo para su propia plataforma por su cuenta. El producto resultante fue el Microsoft SQL Server 4.21

Versión	Año	Nombre Release	Nombre Clave
1.0 (OS/2)	1989	SQL Server 1.0 (16 bit)	Ashton-Tate /Microsoft SQL Server
1.1 (OS/2)	1991	SQL Server 1.1 (16 bit)	-
4.21 (WinNT)	1993	SQL Server 4.21	SQLNT
6.0	1995	SQL Server 6.0	SQL95
6.5	1996	SQL Server 6.5	Hydra
7.0	1998	SQL Server 7.0	Sphinx
-	1999	SQL Server 7.0 OLAP Tools	Palato mania
8.0	2000	SQL Server 2000	Shiloh
8.0	2003	SQL Server 2000 64-bit Edition	Liberty
9.0	2005	SQL Server 2005	Yukon
10.0	2008	SQL Server 2008	Katmai
10.25	2010	Azure SQL DB	Cloud Database or CloudDB
10.50	2010	SQL Server 2008 R2	Kilimanjaro (aka KJ)
11.0	2012	SQL Server 2012	Denali
12.0	2014	SQL Server 2014	SQL14
13.0	2016	SQL Server 2016	SQL16
14.0	2017	SQL Server 2017	Helsinki
15.0	2019	SQL Server 2019	Seattle
16.0	2022	SQL Server 2022	Dallas

Las versiones 7.0 y 2000 fueron modificaciones y extensiones al código desarrollado en conjunto con Sybase. Para la versión 2005 el código fue reescrito completamente.

Desde el lanzamiento de la versión Microsoft SQL Server 2000 se han realizado avances en la performance, las IDE²³ de cliente, y se han agregado paquetes y sistemas complementarios incluyendo:

- ETL²⁴
- Reporting Server
- OLAP²⁵ y Data Mining Server (Analysis Services)
- Tecnologías de mensajería y comunicación (Service Broker, Notification Services)

²³ Integrated Development Environment: Ambiente de desarrollo interactivo o Entorno de desarrollo integrado

²⁴ Extract Transform Load

²⁵ OnLine Analytical Processing



La implementación del SQL de Microsoft recibe comúnmente el nombre de **T-SQL** (o **Transact-SQL**).

Algunas herramientas que se pueden utilizar para acceder a las bases de datos SQL Server son:

- [SQL Server Management Studio \(SSMS\)](#)
- [Azure Data Studio](#)
- [SQL Server Data Tools \(SSDT\)](#)
- [sqlcmd](#)

A continuación, se describen los puntos destacados de las versiones.

SQL SERVER 2005

Lanzado en octubre de 2005. Incluyó soporte nativo para XML²⁶, con XQuery como lenguaje de consultas para datos en XML, que pueden ser embebidos dentro del T-SQL.

SQL SERVER 2008

Lanzado en agosto de 2008. Incluyó la tecnología SQL Server Always On, con el objetivo de hacer que la administración de los datos sea auto optimizante, auto organizada y auto mantenida para lograr un tiempo de caídas de casi cero.

Incluyó además soporte para datos estructurados y semiestructurados, incluyendo formatos para medios digitales como imágenes, audio, video y otros datos multimedia (que anteriormente se almacenaban como objetos genéricos (BLOBs)).

Se incorpora además la funcionalidad Full-text Search que simplifica la administración y mejora la performance.

Con SQL Reporting Services se incorpora la facilidad de obtener reportes y gráficos de manera nativa.

La versión de SQL Server Management Studio incluye IntelliSense para consultas SQL, que permite completar código de manera inteligente y chequeo de sintaxis.

SQL SERVER 2008 R2

Lanzado en abril de 2010. Incluye Master Data Services, que permite una administración centralizada de entidades de datos maestras y su jerarquía.

Se incluyen nuevos servicios orientados al análisis de datos como Power Pivot, StreamInsight, Report Builder 3.0, un agregado de Reporting Services para Sharepoint, funcionalidades de capa de datos para Visual Studio que permite empaquetar bases de datos como parte de una aplicación.

SQL SERVER 2012

Lanzado en marzo de 2012. Incluye mejoras orientadas a la alta disponibilidad como AlwaysOn SQL Server Failover Cluster Instances y Availability Groups, Contained Databases que simplifica la tarea de mover una base de datos entre instancias.

²⁶ Extensible Markup Language



Mejoras en cuanto a la programación, almacenamiento de ubicaciones, metadata, objetos de secuencia y mejoras de performance y seguridad

SQL SERVER 2014

Lanzado en abril de 2014. Incluye mejoras de performance que permite utilizar la velocidad de discos SSD como intermedio entre la memoria RAM y los discos mecánicos, mejoras en los servicios de alta disponibilidad

SQL SERVER 2016

Fue liberada el 1 de junio de 2016. La versión RTM es 13.0.1601.5. SQL Server 2016 es soportado solo en procesadores x64. No existe versión de 32 bits.

SQL SERVER 2017

Microsoft liberó SQL Server 2017 el 2 de octubre de 2017 y es la primera versión con soporte para Linux.

<https://www.microsoft.com/es-es/sql-server/sql-server-2017>

SQL SERVER 2019

Esta versión se liberó al público en noviembre de 2019. Incluye además de mejoras sobre el motor, la capacidad de manejar clústeres de Big Data

SQL SERVER 2022

Incluye innovaciones en integración de datos y bases de datos habilitadas para Azure. Características como la alta disponibilidad bidireccional por primera vez y la recuperación ante desastres entre Azure SQL Managed Instance y SQL Server, la integración de Azure Synapse Link con SQL para ETL, informes y análisis casi en tiempo real sobre sus datos operativos, y nueva inteligencia de consultas integrada de próxima generación con optimización de planes sensible a parámetros.

EDICIONES PRINCIPALES

Enterprise: Edición completa, con performance para aplicaciones de misión crítica y con requerimientos de inteligencia de negocios. Provee los más altos niveles de servicio y performance.

Standard: Administración de datos central y capacidad para inteligencia de negocios para aplicaciones que no sean de misión crítica con el mínimo de recursos de IT

Developer: Versión con todas las características orientada a desarrolladores, que permite desarrollar, probar y demostrar aplicaciones basadas en software SQL Server.

Web: Versión especial para proveedores de hosting.

Express: Nivel básico y gratuita. Ideal para el aprendizaje y construcción de aplicaciones de escritorio o pequeños servidores. Permite hasta 10 GB de almacenamiento.

En el siguiente cuadro podemos ver un resumen de los límites para cada edición.

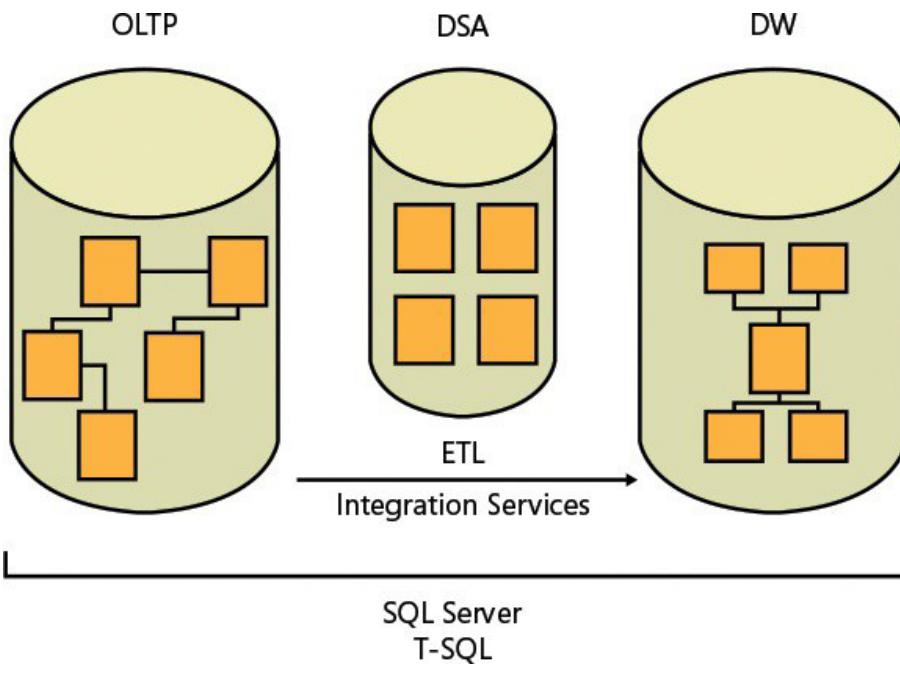


Características	SQL Server Enterprise	SQL Server Standard	SQL Server Express	SQL Server Developer
Número máximo de núcleos	Máximo del SO	24	4	Máximo del SO
Memoria: Tamaño máximo de pool de buffer por instancia	Máximo del SO	128 GB	1410 MB	Máximo del SO
Memoria: Máximo segmento chache Columnstore por instancia	Máximo del SO	32 GB	352 MB	Máximo del SO
Memoria: Máximo memoria optimizada para datos por base de datos	Máximo del SO	32 GB	352 MB	Máximo del SO
Tamaño máximo de base de datos	524 PB	524 PB	10 GB	524 PB
Derecho de uso en producción	Sí	Sí	Sí	No

TIPOS DE SISTEMAS DE BASES DE DATOS

Hay dos tipos de sistemas que utilizan SQL Server como base de datos y T-SQL para administrar y manipular los datos:

- OLTP (*Online transactional processing*)
- DWs (*Data Warehouses*)



- DSA: *Data-staging area*
- ETL: *Extract, Transform and Load*

En la materia vamos a trabajar sobre bases de datos **OLTP**.



ARQUITECTURA DE SQL SERVER

Para contar con SQL server hay distintas alternativas de plataforma, y dentro de cada una de las alternativas veremos cómo se organizan los distintos componentes.

ALTERNATIVAS ABC

Proviene de los nombres en inglés de cada alternativa. A de **Appliance** (aparato), B de **Box** (caja), C de **Cloud** (nube).

CAJA (BOX)

Es la alternativa tradicional. Se aloja habitualmente en las instalaciones del cliente. El cliente es responsable de todo, obtener el hardware, instalar el software, aplicar parches, montar la estrategia para lograr alta disponibilidad, recupero ante desastres, seguridad y todo.

El cliente puede instalar múltiples instancias del producto en el mismo servidor y puede realizar consultas que interactúen con múltiples bases de datos.

El lenguaje de consulta utilizado es el T-SQL.

APARATO (APPLIANCE)

Es una solución llave en mano, con hardware y software preconfigurado. El aparato se aloja habitualmente en las instalaciones del cliente. Estos productos son configurados en conjunto por Microsoft y fabricantes de hardware, como Dell o HP.

Hay aparatos especializados para Data Warehouse, y para manejo de Big Data.

NUBE (CLOUD)

Se proveen recursos a demanda disponibles en la nube. Se puede utilizar una nube pública o una nube privada. La nube privada es similar a la caja, pero por el concepto de nube, sirve para contabilizar el uso de los recursos.

Para la nube pública, Microsoft provee dos alternativas:

- IaaS (**Infrastructure as a Service**)
- PaaS (**Platform as a Service**)

Con la alternativa IaaS, se suministra una máquina virtual que reside en la infraestructura cloud de Microsoft. Hay distintas variantes de máquinas virtuales preconfiguradas, que pueden traer instalada alguna versión de SQL Server. El hardware es mantenido por Microsoft, pero el cliente es responsable del mantenimiento y aplicación de parches del software.

Con PaaS, Microsoft suministra la base de datos en la nube como un servicio. Está alojada en datacenters de Microsoft. La instalación y mantenimiento del hardware y el software, la alta disponibilidad, el recupero ante desastres, son responsabilidad de Microsoft.

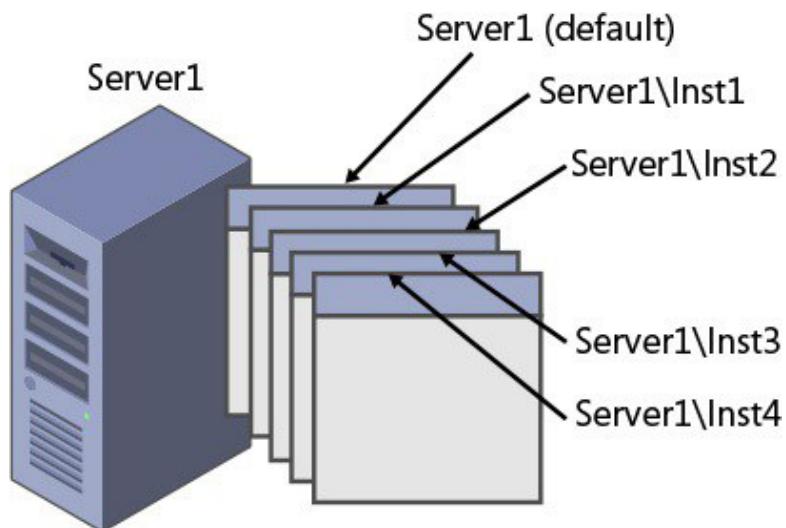
Microsoft cuenta con distintas ofertas de PaaS. Para OLTP ofrece el servicio Azure SQL Database, o SQL Database. También cuenta con una versión para Data Warehouse, llamada Microsoft Azure SQL Data Warehouse.

Microsoft ofrece además otros servicios de datos en la nube, como Data Lake para servicios relacionados con Big Data, Azure DocumentDB para servicios de documentos NoSQL., y otros.



INSTANCIAS DE SQL SERVER

Dentro de un mismo servidor se pueden instalar múltiples instancias de SQL Server. Cada instancia es completamente independiente de las otras en cuanto a seguridad, la información que manejan y en definitiva en todos los aspectos.

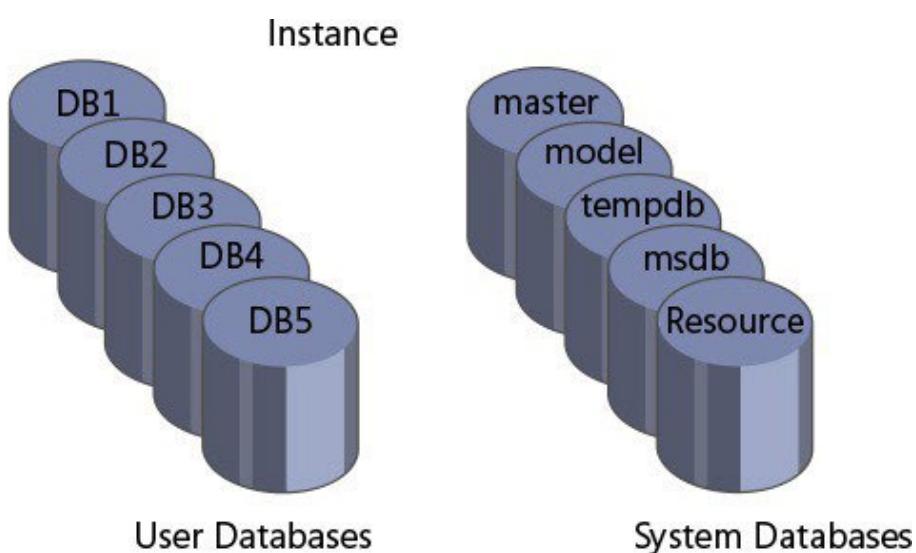


A nivel lógico dos instancias instaladas en el mismo equipo son tan independientes entre sí como si estuvieran instaladas en distintos equipos físicos.

En mismo equipo puede instalarse una instancia default, que es accedida a través del nombre del equipo, y múltiples instancias nominadas, que son accedidas a través del nombre del equipo y el nombre de la instancia separados por una barra invertida (\), por ejemplo, PCJUAN\SQLEXPRESS (el equipo llamado PCJUAN y la instancia SQLEXPRESS).

BASES DE DATOS

Una base de datos puede verse como un contenedor de objetos, como tablas, vistas, stored procedures y otros. Cada instancia de SQL Server puede contener múltiples bases de datos.



Al instalar una instancia, se crean varias bases de datos de sistema, que contienen información propia del sistema y se utilizan de manera interna para la administración de la base de datos.

Una vez que está instalada la base de datos se pueden crear múltiples bases de datos de usuario.

Las bases de datos del sistema son:

- **master**: contiene metadata de la instancia, configuración del servidor, información sobre las bases de datos de la instancia e información de inicialización.
- **Resource**: es una base oculta y de sólo lectura que contiene la definición de los objetos de sistema.
- **model**: es usada como template para las nuevas bases de datos que se crean en la instancia.
- **tempdb**: es la base donde SQL Server almacena la información temporal, como tablas de trabajo, espacio de ordenamiento, versionado de filas, etc.
- **msdb**: es utilizada por el servicio SQL Server Agent. Este agente es encargado de la automatización de procesos

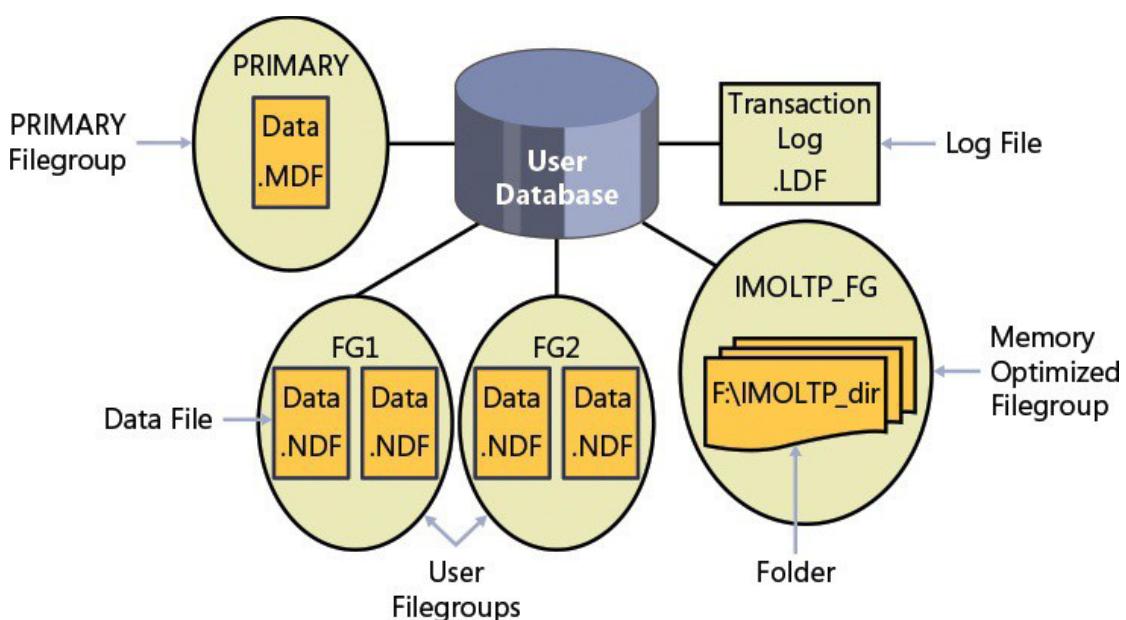
En una instancia pueden crearse tantas bases de datos como sean necesarias.

Para poder ejecutar comandos T-SQL contra la base de datos es necesario que una aplicación cliente se conecte a la instancia y especificar el contexto o base de datos necesaria.

En términos de seguridad, para poder conectarse a una instancia, administrador de la base de datos (DBA) debe crear un logon para el usuario. Veremos este tema más en profundidad cuando veamos los aspectos de seguridad.

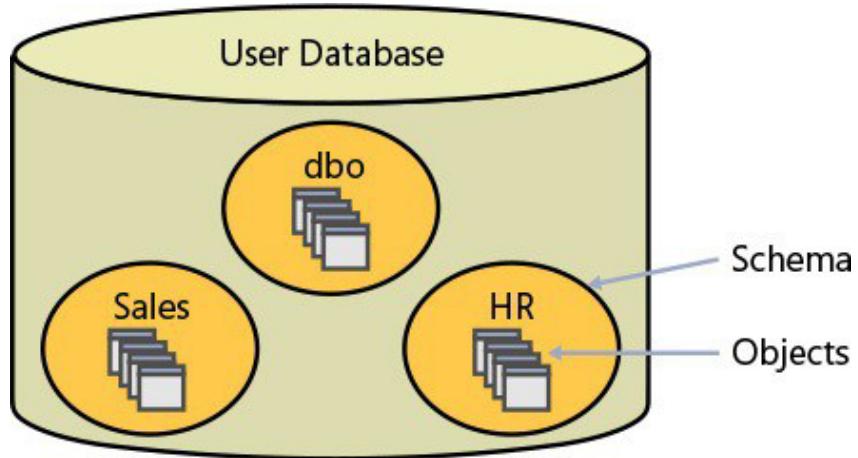
Esto es la parte lógica de la base de datos. Como usuarios, es probable que hasta este nivel nos interese conocer. Pero hay un aspecto físico en cuanto a la base de datos, que necesitamos conocer sobre todo si estamos en el rol de administrador de la base de datos.

La base de datos está formada por archivos de datos, archivos de log de transacciones, y opcionalmente por archivos de control (checkpoint) que contienen los datos optimizados en memoria (In-Memory OLTP)



ESQUEMAS Y OBJETOS

Al decir que una base de datos contenía objetos, en realidad simplificamos un poco las cosas. En realidad, una base de datos contiene esquemas y son los esquemas los que contienen objetos.



Se pueden controlar los permisos de acceso a nivel esquema, de modo de poder separar los diferentes permisos de acceso a objetos de manera organizacional.

El esquema es también un espacio de nombres, ya que es usado como prefijo de los nombres de objeto. Por ejemplo, dentro del esquema del departamento de ventas (**Sales**) podemos tener una tabla para almacenar las órdenes (**Orders**) a la que accederemos como **Sales.Orders**.

Si se omite el nombre del esquema al hacer referencia a un objeto, SQL Server buscará primero en el esquema por default del usuario y si no encuentra el objeto lo buscará en el esquema dbo.

MYSQL²⁷

MySQL, el sistema de gestión de bases de datos SQL de código abierto más popular, es desarrollado, distribuido y respaldado por Oracle Corporation.

El sitio web de MySQL (<http://www.mysql.com/>) proporciona la información más reciente sobre el software MySQL.

¿QUÉ ES MYSQL?

MYSQL ES UN SISTEMA DE GESTIÓN DE BASES DE DATOS.

Una base de datos es una colección estructurada de datos. Puede ser cualquier cosa, desde una simple lista de compras hasta una galería de imágenes o la gran cantidad de información en una red corporativa. Para agregar, acceder y procesar datos almacenados en una base de datos de computadora, necesitamos un sistema de gestión de bases de datos como MySQL Server. Dado que las computadoras son muy buenas para manejar grandes cantidades de datos, los sistemas de gestión de bases de datos desempeñan un papel central en la informática, como utilidades independientes o como parte de otras aplicaciones

²⁷ <https://www.mysql.com/>



LAS BASES DE DATOS MYSQL SON RELACIONALES.

Una base de datos relacional almacena datos en tablas separadas en lugar de poner todos los datos en un gran almacén. Las estructuras de la base de datos están organizadas en archivos físicos optimizados para la velocidad. El modelo lógico, con objetos como bases de datos, tablas, vistas, filas y columnas, ofrece un entorno de programación flexible. Configura reglas que rigen las relaciones entre diferentes campos de datos, como uno a uno, uno a muchos, único, obligatorio u opcional, y "punteros" entre diferentes tablas. La base de datos hace cumplir estas reglas, de modo que, con una base de datos bien diseñada, su aplicación nunca ve datos incoherentes, duplicados, huérfanos, desactualizados o faltantes.

La parte SQL de "MySQL" significa "Lenguaje de consulta estructurado". SQL es el lenguaje estandarizado más común utilizado para acceder a las bases de datos. Según su entorno de programación, puede ingresar SQL directamente (por ejemplo, para generar informes), incrustar declaraciones de SQL en código escrito en otro idioma o usar una API específica del idioma que oculta la sintaxis de SQL.

SQL está definido por el estándar ANSI/ISO SQL. El estándar SQL ha estado evolucionando desde 1986 y existen varias versiones. En este apartado, "SQL-92" se refiere al estándar lanzado en 1992, "SQL:1999" se refiere al estándar lanzado en 1999 y "SQL:2003" se refiere a la versión actual del estándar. Usamos la frase "el estándar SQL" para referirnos a la versión actual del estándar SQL en cualquier momento.

EL SOFTWARE MYSQL ES DE CÓDIGO ABIERTO.

Código abierto significa que cualquiera puede usar y modificar el software. Cualquiera puede descargar el software MySQL de Internet y usarlo sin pagar nada. Si lo desea, puede estudiar el código fuente y modificarlo para adaptarlo a sus necesidades. El software MySQL utiliza la GPL (Licencia Pública General GNU), <http://www.fsf.org/licenses/>, para definir lo que puede y no puede hacer con el software en diferentes situaciones. Si no se siente cómodo con la GPL o necesita incrustar el código MySQL en una aplicación comercial, puede comprarle a Oracle una versión con licencia comercial.

Consulte la Descripción general de licencias de MySQL para obtener más información (<http://www.mysql.com/company/legal/licensing/>).

EL SERVIDOR DE BASE DE DATOS MYSQL ES MUY RÁPIDO, CONFIABLE, ESCALABLE Y FÁCIL DE USAR.

Si eso es lo que estás buscando, deberías darle una oportunidad. MySQL Server puede ejecutarse cómodamente en una computadora de escritorio o portátil, junto con otras aplicaciones, servidores web, etc., que requieren poca o ninguna atención. Si dedica una máquina completa a MySQL, puede ajustar la configuración para aprovechar toda la memoria, la potencia de la CPU y la capacidad de E/S disponibles. MySQL también puede escalar a grupos de máquinas conectadas en red.

MySQL Server se desarrolló originalmente para manejar grandes bases de datos mucho más rápido que las soluciones existentes y se ha utilizado con éxito en entornos de producción muy exigentes durante varios años. Aunque está en constante desarrollo, MySQL Server hoy ofrece un conjunto rico y útil de funciones. Su conectividad, velocidad y seguridad hacen que MySQL Server sea muy adecuado para acceder a bases de datos en Internet.

MYSQL SERVER FUNCIONA EN SISTEMAS CLIENTE/SERVIDOR O INTEGRADOS.

El software de base de datos MySQL es un sistema cliente/servidor que consta de un servidor SQL multiproceso que admite diferentes back-ends, varios programas y bibliotecas de clientes diferentes, herramientas administrativas y una amplia gama de interfaces de programación de aplicaciones (API).



También se proporciona MySQL Server como una biblioteca integrada de subprocessos múltiples que puede vincular a su aplicación para obtener un producto independiente más pequeño, más rápido y fácil de administrar.

HAY DISPONIBLE UNA GRAN CANTIDAD DE SOFTWARE MYSQL CONTRIBUIDO.

MySQL Server tiene un conjunto práctico de funciones desarrolladas en estrecha colaboración con sus usuarios. Es muy probable que su aplicación o lenguaje favorito sea compatible con el servidor de base de datos MySQL.

La forma oficial de pronunciar "MySQL" es "My Ess Que Ell" (no "my sequel"), pero no nos importa si lo pronuncia como "my sequel" o de alguna otra forma localizada.

LAS CARACTERÍSTICAS PRINCIPALES DE MYSQL

Esta sección describe algunas de las características importantes del software de base de datos MySQL. En la mayoría de los aspectos, la hoja de ruta se aplica a todas las versiones de MySQL. Para obtener información sobre las características a medida que se introducen en MySQL en una serie específica, consulte la sección "En pocas palabras" del Manual correspondiente:

- MySQL 8.0: [Section 1.3, "What Is New in MySQL 8.0"](#)
- MySQL 5.7: [What Is New in MySQL 5.7](#)
- MySQL 5.6: [What Is New in MySQL 5.6](#)

INTERNOS Y PORTABILIDAD

- Escrito en C y C++.
- Probado con una amplia gama de diferentes compiladores.
- Funciona en muchas plataformas diferentes. Consulte <https://www.mysql.com/support/supportedplatforms/database.html>
- Para portabilidad, configurado usando CMake .
- Probado con Purify (un detector de pérdida de memoria comercial) así como con Valgrind, una herramienta GPL (<http://developer.kde.org/~sewardj/>).
- Utiliza un diseño de servidor de varias capas con módulos independientes.
- Diseñado para ser completamente multiproceso utilizando subprocessos del kernel, para usar fácilmente múltiples CPU si están disponibles.
- Proporciona motores de almacenamiento transaccional y no transaccional.
- Utiliza tablas de disco de B-tree muy rápidas (MyISAM) con compresión de índice.
- Diseñado para que sea relativamente fácil agregar otros motores de almacenamiento. Esto es útil si desea proporcionar una interfaz SQL para una base de datos interna.
- Utiliza un sistema de asignación de memoria basado en subprocessos muy rápido.
- Ejecuta joins muy rápidos mediante un join de bucle anidado optimizado.
- Implementa tablas hash en memoria, que se utilizan como tablas temporales.
- Implementa funciones SQL utilizando una biblioteca de clases altamente optimizada que debería ser lo más rápida posible. Por lo general, no hay ninguna asignación de memoria después de la inicialización de la consulta.
- Proporciona el servidor como un programa independiente para su uso en un entorno de red cliente/servidor y como una biblioteca que se puede incrustar (enlazar) en aplicaciones independientes. Estas aplicaciones se pueden utilizar de forma aislada o en entornos donde no hay red disponible.



TIPOS DE DATOS

- Muchos tipos de datos: enteros con/sin signo de 1, 2, 3, 4 y 8 bytes de longitud, FLOAT, DOUBLE, CHAR, VARCHAR, BINARY, VARBINARY, TEXT, BLOB, DATE, TIME, DATETIME, TIMESTAMP, YEAR, SET, ENUM y tipos espaciales de OpenGIS.
- Tipos de cadena de longitud fija y de longitud variable.

DECLARACIONES Y FUNCIONES

- Soporte completo de operadores y funciones en el SELECT WHERE en consultas. Por ejemplo:

```
mysql> SELECT CONCAT(first_name, ' ', last_name)
      -> FROM citizen
      -> WHERE income/dependents > 10000 AND age > 30;
```

- Soporte completo para cláusulas SQL GROUP BY y ORDER BY. Compatibilidad con funciones de grupo (COUNT(), AVG(), STD(), SUM(), MAX(), MIN() y GROUP_CONCAT()).
- Compatibilidad con LEFT OUTER JOIN y RIGHT OUTER JOIN con la sintaxis estándar de SQL y ODBC.
- Compatibilidad con alias en tablas y columnas según lo requiere el SQL estándar.
- Compatibilidad con DELETE, INSERT, REPLACE y UPDATE para devolver la cantidad de filas que se cambiaron (afectadas) o para devolver la cantidad de filas que coincidieron configurando un indicador al conectarse al servidor.
- Compatibilidad con declaraciones específicas de MySQL SHOW que recuperan información sobre bases de datos, motores de almacenamiento, tablas e índices. Soporte para la base de datos INFORMATION_SCHEMA, implementado según estándar SQL.
- Una sentencia EXPLAIN para mostrar cómo el optimizador resuelve una consulta.
- Independencia de los nombres de funciones de los nombres de tablas o columnas. Por ejemplo, ABS es un nombre de columna válido. La única restricción es que para una llamada de función, no se permiten espacios entre el nombre de la función y el "(" que le sigue.
- Puede hacer referencia a tablas de diferentes bases de datos en la misma declaración.

SEGURIDAD

- Un sistema de privilegios y contraseñas que es muy flexible y seguro, y que permite la verificación basada en host.
- Seguridad de contraseñas mediante el cifrado de todo el tráfico de contraseñas cuando se conecta a un servidor.

ESCALABILIDAD Y LÍMITES

- Soporte para grandes bases de datos. Se usa MySQL Server con bases de datos que contienen 50 millones de registros. También sabemos de usuarios que utilizan MySQL Server con 200.000 tablas y alrededor de 5.000.000.000 de filas.
- Soporte para hasta 64 índices por tabla. Cada índice puede constar de 1 a 16 columnas o partes de columnas. El ancho de índice máximo para tablas InnoDB es 767 bytes o 3072 bytes. El ancho de índice máximo para tablas MyISAM es de 1000 bytes. Un índice puede usar un prefijo de una columna para los tipos de columna CHAR, VARCHAR, BLOB o TEXT.

CONECTIVIDAD

- Los clientes pueden conectarse a MySQL Server usando varios protocolos:
 - Los clientes pueden conectarse mediante sockets TCP/IP en cualquier plataforma.



- En los sistemas Windows, los clientes pueden conectarse mediante canalizaciones con nombre si el servidor se inicia con la variable de sistema named_pipe habilitada. Los servidores de Windows también admiten conexiones de memoria compartida si se inician con la variable de sistema shared_memory habilitada. Los clientes pueden conectarse a través de la memoria compartida utilizando la opción --protocol=memory.
- En los sistemas Unix, los clientes pueden conectarse mediante archivos de socket de dominio Unix.
- Los programas cliente de MySQL se pueden escribir en muchos lenguajes. Una biblioteca cliente escrita en C está disponible para clientes escritos en C o C++, o para cualquier lenguaje que proporcione enlaces C.
- Las API para C, C++, Eiffel, Java, Perl, PHP, Python, Ruby y Tcl están disponibles, lo que permite que los clientes de MySQL se escriban en muchos lenguajes
- La interfaz Connector/ODBC (MyODBC) proporciona compatibilidad con MySQL para programas cliente que utilizan conexiones ODBC (Open Database Connectivity). Por ejemplo, puede usar MS Access para conectarse a su servidor MySQL. Los clientes se pueden ejecutar en Windows o Unix. La fuente del conector/ODBC está disponible. Se admiten todas las funciones de ODBC 2.5, al igual que muchas otras.
- La interfaz Connector/J proporciona compatibilidad con MySQL para programas de cliente Java que utilizan conexiones JDBC. Los clientes se pueden ejecutar en Windows o Unix. La fuente del conector/J está disponible.
- MySQL Connector/.NET permite a los desarrolladores crear fácilmente aplicaciones .NET que requieren conectividad de datos segura y de alto rendimiento con MySQL. Implementa las interfaces ADO.NET requeridas y se integra en las herramientas compatibles con ADO.NET. Los desarrolladores pueden crear aplicaciones usando su lenguaje de elección .NET. MySQL Connector/.NET es un controlador ADO.NET totalmente administrado escrito en C# 100%.

LOCALIZACIÓN

- El servidor puede proporcionar mensajes de error a los clientes en muchos idiomas.
- Soporte completo para varios juegos de caracteres diferentes, incluidos latin1 (cp1252), german, big5, ujis, varios juegos de caracteres Unicode y más. Por ejemplo, los caracteres escandinavos “å”, “ä” y “ö” están permitidos en los nombres de tablas y columnas.
- Todos los datos se guardan en el conjunto de caracteres elegido.
- El ordenamiento y las comparaciones se realizan de acuerdo con el juego de caracteres y la intercalación predeterminados. Es posible cambiar esto cuando se inicia el servidor MySQL. Para ver un ejemplo de ordenamiento muy avanzado, consulte el código de clasificación checo. MySQL Server admite muchos conjuntos de caracteres diferentes que se pueden especificar en tiempo de compilación y tiempo de ejecución.
- La zona horaria del servidor se puede cambiar dinámicamente y los clientes individuales pueden especificar su propia zona horaria.

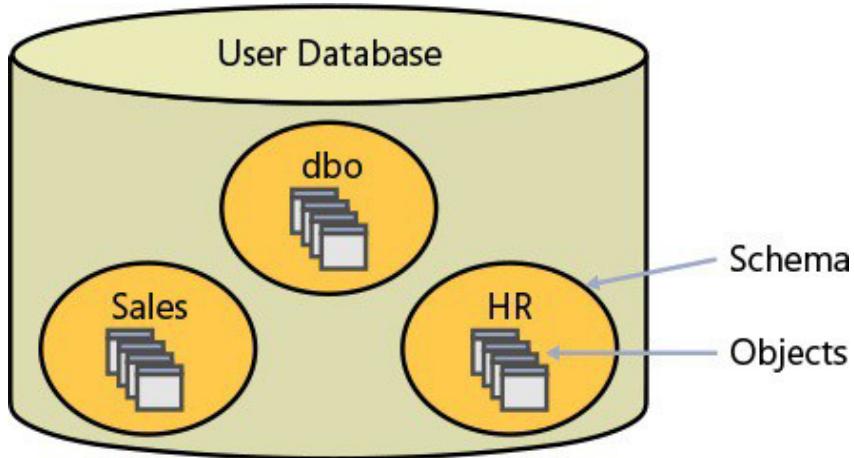
CLIENTES Y HERRAMIENTAS

- MySQL incluye varios programas de utilidad y de cliente. Estos incluyen tanto programas de línea de comandos como mysqldump y mysqladmin, como programas gráficos como MySQL Workbench.
- MySQL Server tiene soporte incorporado para declaraciones SQL para verificar, optimizar y reparar tablas. Estas sentencias están disponibles desde la línea de comandos a través del cliente mysqlcheck. MySQL también incluye myisamchk, una utilidad de línea de comandos muy rápida para realizar estas operaciones en las tablas MyISAM.
- Los programas MySQL se pueden invocar con la opción --help o -? para obtener asistencia en línea.
- En octubre de 2022 se publicó una extensión para Azure Data Studio que permite conectarse con bases MySQL, de modo que podemos utilizar Azure Data Studio como cliente para bases MySQL.



ESQUEMAS

Hay una diferencia importante en la organización entre SQL Server y MySQL con respecto a los esquemas. En SQL Server, al igual que en Oracle, los esquemas representan una parte de la base de datos.



En la imagen podemos ver por ejemplo tres esquemas diferentes en la misma base de datos. El esquema dbo, el esquema de Ventas (Sales) y el esquema de Recursos Humanos (HR). La utilización de esquemas permite dar un nivel adicional de organización en la base de datos, y también puede ser utilizado para mejorar la seguridad, simplificando los modelos de permisos (que pueden ser asignados por esquemas) (ver Arquitectura de los SGBD en la Unidad 01).

En cambio, para MySQL, esquema es un sinónimo de base de datos, es decir que es equivalente utilizar **CREATE SCHEMA** o **CREATE DATABASE** para crear una nueva base de datos.

CATEGORÍAS DE LAS INSTRUCCIONES SQL

El lenguaje SQL puede dividirse en cuatro partes:

- **DML** (Database Manipulation Language o Lenguaje de manipulación de datos): SELECT, INSERT, UPDATE, DELETE, TRUNCATE
- **DDL** (Database Definition Language o Lenguaje de definición de datos): CREATE, ALTER, DROP
- **DCL** (Database Control Language o Lenguaje de control de datos): GRANT, DENY, REVOKE
- **TCL** (Transaction Control Language o Lenguaje de control de transacciones): COMMIT, ROLLBACK, SAVEPOINT

RESTRICCIONES (CONSTRAINTS)

Las restricciones ayudan a mantener la integridad de los datos. Son reglas que los datos deben cumplir. Se definen en el modelo de datos y el sistema de gestión de base de datos se encarga de que se cumplan.

El primer tipo de restricción, aunque no es una restricción en sí, es el uso de tipos de datos. Si un atributo está definido como fecha, sólo aceptará como valores fechas que sean válidas.

Junto con el tipo de datos se define para cada campo si el mismo puede tomar valores indefinidos (NULL).

Para cumplir la regla de integridad de entidad, se define para cada tabla una clave primaria, que es un identificador único de cada registro (tupla), y por lo tanto no admiten valores indefinidos (NULL).

Las claves candidatas son otro tipo de restricción que también colaboran con la integridad de datos.



Las claves foráneas permiten cumplir con la regla de integridad referencial. Una clave foránea es un atributo, o conjunto de atributos, que referencian a una clave candidata de otra tabla (o podría ser de la misma tabla). Por ejemplo, en una tabla de *Empleados*, podemos tener una clave foránea definida en el atributo *iddepartamento*, que referencia a la clave primaria *iddepartamento* de la tabla *Departamentos*. Esto quiere decir que los únicos valores que puede tomar *Empleados.iddepartamento* son los que aparecen en *Departamentos.iddepartamento*.

NULL Y LA LÓGICA DE TRES VALORES

El SQL utiliza una lógica de tres valores (verdadero, falso y desconocido) con las siguientes tablas de verdad

OR	Verdadero	Falso	Desconocido
Verdadero	Verdadero	Verdadero	Verdadero
Falso	Verdadero	Falso	Desconocido
Desconocido	Verdadero	Desconocido	Desconocido

AND	Verdadero	Falso	Desconocido
Verdadero	Verdadero	Falso	Desconocido
Falso	Falso	Falso	Desconocido
Desconocido	Desconocido	Desconocido	Desconocido

NOT
Verdadero
Falso
Desconocido

TIPOS DE DATOS EN SQL²⁸

Cuando definimos una tabla, variable o constante debemos asignar un tipo de dato que indica los posibles valores. El tipo de datos define el formato de almacenamiento, espacio de disco-memoria que va a ocupar un campo o variable, restricciones y rango de valores válidos (datos enteros, cadenas de caracteres, números reales, fechas y horas, cadenas binarias, etc.)

SQL Server proporciona un conjunto de tipos de datos de sistemas, que define todos los tipos de datos que pueden ser utilizados con SQL Server. Se pueden definir además tipos de datos propios en Transact-SQL o Microsoft .NET Framework. Los tipos de datos alias están basados en tipos de datos del sistema.

Cuando dos expresiones que tienen tipos de datos diferentes, cotejos, precisión, escala o longitud son combinados por un operador, las características del resultado son determinadas por:

- El tipo de datos del resultado es determinado aplicando las reglas de precedencia de tipos de datos a los datos de entrada.
- El cotejo del resultado es determinado por las reglas de precedencia de cotejo cuando el tipo de datos del resultado es char, varchar, text, nchar, nvarchar, o ntext.

²⁸ <https://docs.microsoft.com/en-us/sql/t-sql/data-types/data-types-transact-sql>
<https://dev.mysql.com/doc/refman/8.0/en/data-types.html>



- La precisión, escala y longitud del resultado dependen de la precisión, escala y longitud de la expresión de entrada.

SQL Server suministra sinónimos de tipos de datos para mantener la compatibilidad con el estándar ISO.

CATEGORÍAS DE TIPOS DE DATOS

Los tipos de datos se pueden organizar en las siguientes categorías:

Números Exactos	Cadenas de caracteres Unicode
Números Aproximados	Cadenas binarias
Fecha y Hora	Otros tipos de datos
Cadenas de caracteres	

En SQL Server, basados en las características de almacenamiento, algunos tipos de datos son clasificados como parte de alguno de los siguientes grupos:

- Tipos de datos grandes: varchar(max), nvarchar(max), y varbinary(max)
- Tipos de datos de objetos grandes: text, ntext, image, varchar(max), nvarchar(max), varbinary(max), y xml

NÚMEROS EXACTOS

ENTEROS

SQL Server

bigint	bit
int	smallint
tinyint	

bigint: -2^{63} (-9,223,372,036,854,775,808) a $2^{63}-1$ (9,223,372,036,854,775,807)

bit: 1, 0 o NULL

int: -2^{31} (-2,147,483,648) a $2^{31}-1$ (2,147,483,647)

smallint: -2^{15} (-32,768) a $2^{15}-1$ (32,767)

tinyint: 0 a 255

MySQL

bigint	int
mediumint	smallint
tinyint	

bigint: -2^{63} a $2^{63}-1$

int: -2,147,483,648 a 2,147,483,647

mediumint: -8,388,608 a 8,388,607

smallint: -32,768 a 32,767

tinyint: 0 a 255

El tipo bit tiene un uso y significado diferente en MySQL. El tipo de datos BIT se utiliza para almacenar valores de bits. Un tipo de BIT(M) permite el almacenamiento de valores de M bits. M puede variar de 1 a 64.

Para especificar valores de bits, se puede utilizar la notación b'value'. valor es un valor binario escrito usando ceros y unos. Por ejemplo, b'111' y b'10000000' representan 7 y 128, respectivamente.

Si asigna un valor a una columna BIT(M) que tiene una longitud inferior a M bits, el valor se rellena a la izquierda con ceros. Por ejemplo, asignar un valor de b'101' a una columna BIT(6) es, en efecto, lo mismo que asignar b'000101'.

En MySQL tenemos disponibles los tipos BOOL y BOOLEAN, que se comportan como sinónimos de TINYINT(1).



TIPOS DE PUNTO FIJO

SQL Server

decimal	money
numeric	smallmoney

decimal (p,s): p de 1 a 38. s de 0 a p

money: -922,337,203,685,477.5808 a
922,337,203,685,477.5807

numeric: equivalente a decimal

smallmoney: - 214,748.3648 a 214,748.3647

MySQL

decimal	numeric
---------	---------

decimal (p,s): p de 1 a 65. s de 0 a p

numeric: equivalente a decimal

Para armar un equivalente a money, se puede utilizar decimal(19,4) y para smallmoney se puede utilizar decimal(10,4)

NÚMEROS APROXIMADOS

SQL Server

float	real
-------	------

float (n): n de 1 a 53, siendo n la cantidad de bits usados para la mantisa. - 1.79E+308 a -2.23E-308, 0 y 2.23E-308 a 1.79E+308

real: float(24)

MySQL

float	double
-------	--------

float (n): n de 0 a 23

double(n): n de 24 a 53

FECHA Y HORA

SQL Server

date	datetime2
datetime	datetimeoffset
smalldatetime	time

date: YYYY-MM-DD. De 0001-01-01 hasta 9999-12-31

datetime2: YYYY-MM-DD hh:mm:ss[.fracción de segundo]. De 0001-01-01 hasta 9999-12-31 y 00:00:00 hasta 23:59:59.999999

datetime: De 1 de enero de 1753 hasta 31 de diciembre de 9999 y 00:00:00 hasta 23:59:59.997

datetimeoffset: YYYY-MM-DD hh:mm:ss[.nnnnnnnn] [{+|-}hh:mm]. El offset va de -14:00 hasta +14:00

smalldatetime: De 1900-01-01 hasta 2079-06-06 y 00:00:00 hasta 23:59:59

time: hh:mm:ss[.nnnnnnnn] De 00:00:00.0000000 hasta 23:59:59.9999999

MySQL

date	datetime
time	timestamp
year	

date: YYYY-MM-DD. De 1001-01-01 hasta 9999-12-31

datetime: De 1001-01-01 hasta 9999-12-31 y 00:00:00.000000 hasta 23:59:59.999999

time: hh:mm:ss[.nnnnnnnn] De -853:59:59.000000 a 838:59:59.000000

timestamp: De 1970-01-01 00:00:01.000000 a 2038-01-19 03:14:07.999999

year: De 1901 a 2155, o 0000



CADENAS DE CARACTERES

SQL Server

char	varchar
text	

char(n): n de 1 a 8000. Longitud fija.

text: será removido. No usar

varchar(n | max): n de 1 a 8000. Longitud variable

MySQL

char	longtext
mediumtext	text
tinytext	varchar

char(n): n de 0 a 255. Longitud fija.

longtext: longitud máxima de 4.294.967.295 o 4GB ($2^{32} - 1$) bytes

mediumtext: longitud máxima de 16.777.215 ($2^{24} - 1$) bytes

text(n): longitud máxima de 65535 ($2^{16} - 1$) bytes

tinytext: longitud máxima 255 ($2^8 - 1$) bytes

varchar(n | max): n de 1 a 65535. Longitud variable

CADENAS DE CARACTERES UNICODE

SQL Server

nchar	nvarchar
ntext	

nchar(n): n de 1 a 4000. Longitud fija. Soporta caracteres Unicode

ntext: será removido. No usar

nvarchar(n | max): n de 1 a 4000. Longitud variable

MySQL

No maneja tipos diferenciados, si no que permite utilizar los mismos tipos, diferenciando la capacidad de manejar caracteres UNICODE mediante la utilización de CHARACTER SET, por ejemplo:

```
CREATE TABLE t
(
c1 VARCHAR(20) CHARACTER SET utf8,
c2 TEXT CHARACTER SET latin1 COLLATE
latin1_general_ci
);
```

CADENAS BINARIAS

SQL Server

binary	varbinary
image	

binary(n): n de 1 a 8000. cadena binaria de longitud fija

image: será removido. No usar

varbinary(n): n de 1 a 8000. cadena binaria de longitud variable

MySQL

binary	blob
longblob	mediumblob
tinyblob	varbinary

binary(n): similar a char, pero almacena cadena binaria

blob(n): longitud máxima 65.535 ($2^{16} - 1$) bytes. Cada blob es almacenado usando un prefijo de 2 bytes que indica la cantidad de bytes en el valor.

longblob: columna blob con longitud máxima de 4.294.967.295 o 4GB ($2^{32} - 1$) bytes. Usa prefijo de 4 bytes.

mediumblob: columna blob con longitud máxima de 16.777.215 ($2^{24} - 1$) bytes. Usa prefijo de 3 bytes.

tinyblob: columna blob con longitud máxima de 255 ($2^8 - 1$) bytes

varbinary(n): similar a varchar, pero almacena cadena binaria



OTROS TIPOS DE DATOS

SQL Server

cursor	hierarchyid
sql_variant	Spatial Geometry
table	rowversion
uniqueidentifier	xml
Spatial Geography	

MySQL

JSON	Spatial Types
------	---------------

ACLARACIONES BÁSICAS: COMENTARIOS

La manera estándar de comentar una línea es mediante dos guiones (--). Sin embargo, distintos motores han implementado distintas alternativas:

Ejemplo	-- Comentario	# Comentario	/* Comentario */
ANSI	Sí	No	No
SQL Server	Sí	No	Sí
Oracle	Sí	No	Sí
MySQL	Sí	Sí	Sí

ESPACIOS EN BLANCO Y EL PUNTO Y COMA

Los espacios en blanco y los saltos de línea son ignorados por los intérpretes SQL, así que se pueden utilizar libremente para darle mayor claridad y legibilidad a las consultas.

El indicador de finalización de una consulta es el punto y coma (;). En algunas herramientas puede omitirse (como en el SQL Management Studio), pero en otras es fundamental (como en SQLCMD).

Tanto SQL Server como MySQL permiten el uso de espacios en blanco en nombres de tablas y nombres de campos, aunque no es recomendable su uso, pues es probable que genere confusiones o errores. En SQL Server pueden utilizarse las comillas dobles ("") o los corchetes ([]), por ejemplo, podemos hacer referencia la tabla "Order Details" o a la tabla [Order Details]. En cambio, en MySQL deberemos utilizar las comillas simples invertidas (`), así podemos hacer referencia a `Order Details`.

SENSIBILIDAD A MAYÚSCULAS Y MINÚSCULAS

Las instrucciones SQL no son *case-sensitive*, esto quiere decir que se pueden escribir las consultas indistintamente en mayúsculas o minúsculas. Sin embargo, es una práctica habitual utilizar los comandos y palabras reservadas en MAYÚSCULAS.

Los nombres definidos por el usuario, como nombres de tablas y nombres de campos, pueden ser case-sensitive o no, dependiendo del sistema operativo utilizado y la configuración del sistema de gestión de base de datos. Por ejemplo, MySQL en Linux es *case-sensitive* por defecto.



DDL

CREATE TABLE

La sintaxis básica es:

```
CREATE TABLE < tabla > (
    atributo1 dominio1 [NOT NULL],
    ...
    atributoN dominioN [NOT NULL]);
```

La sintaxis completa en SQL Server es:

```
CREATE TABLE
    [ database_name . [ schema_name ] . | schema_name . ] table_name
    [ AS FileTable ]
    ( { <column_definition> | <computed_column_definition>
        | <column_set_definition> | [ <table_constraint> ]
    | [ <table_index> ] [ ,...n ] } )
    [ ON { partition_scheme_name ( partition_column_name ) | filegroup
        | "default" } ]
    [ { TEXTIMAGE_ON { filegroup | "default" } ]
    [ FILESTREAM_ON { partition_scheme_name | filegroup
        | "default" } ]
    [ WITH ( <table_option> [ ,...n ] ) ]
[ ; ]
```

<column_definition> ::=
 column_name <data_type>
 [FILESTREAM]
 [COLLATE collation_name]
 [SPARSE]
 [NULL | NOT NULL]
 [
 [CONSTRAINT constraint_name] DEFAULT constant_expression]
 | [IDENTITY [(seed,increment)] [NOT FOR REPLICATION]
]
 [ROWGUIDCOL]
 [<column_constraint> [...n]]
 [<column_index>]

<data type> ::=
 [type_schema_name .] type_name
 [(precision [, scale] | max |
 [{ CONTENT | DOCUMENT }] xml_schema_collection)]

<column_constraint> ::=
 [CONSTRAINT constraint_name]
 { PRIMARY KEY | UNIQUE }
 [CLUSTERED | NONCLUSTERED]
 [
 WITH FILLFACTOR = fillfactor
 | WITH (< index_option > [, ...n])
]
 [ON { partition_scheme_name (partition_column_name) }



```

| filegroup | "default" } ]



| [ FOREIGN KEY ]
  REFERENCES [ schema_name . ] referenced_table_name [ ( ref_column ) ]
  [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
  [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
  [ NOT FOR REPLICATION ]



| CHECK [ NOT FOR REPLICATION ] ( logical_expression )
}

<column_index> ::=

INDEX index_name [ CLUSTERED | NONCLUSTERED ]
[ WITH ( <index_option> [ ,... n ] ) ]
[ ON { partition_scheme_name (column_name )
      | filegroup_name
      | default
      }
]
[ FILESTREAM_ON { filestream_filegroup_name | partition_scheme_name | "NULL" } ]



<computed_column_definition> ::=
column_name AS computed_column_expression
[ PERSISTED [ NOT NULL ] ]
[
  [ CONSTRAINT constraint_name ]
  { PRIMARY KEY | UNIQUE }
  [ CLUSTERED | NONCLUSTERED ]
  [
    WITH FILLFACTOR = fillfactor
    | WITH ( <index_option> [ , ...n ] )
  ]
  [ ON { partition_scheme_name ( partition_column_name )
        | filegroup | "default" } ]

| [ FOREIGN KEY ]
  REFERENCES referenced_table_name [ ( ref_column ) ]
  [ ON DELETE { NO ACTION | CASCADE } ]
  [ ON UPDATE { NO ACTION } ]
  [ NOT FOR REPLICATION ]



| CHECK [ NOT FOR REPLICATION ] ( logical_expression )
]

<column_set_definition> ::=
column_set_name XML COLUMN_SET FOR ALL_SPARSE_COLUMNS


< table_constraint > ::=
[ CONSTRAINT constraint_name ]
{
  { PRIMARY KEY | UNIQUE }
  [ CLUSTERED | NONCLUSTERED ]
  (column [ ASC | DESC ] [ ,...n ] )
  [
    WITH FILLFACTOR = fillfactor
    |WITH ( <index_option> [ , ...n ] )
  ]
}

```



```

[ ON { partition_scheme_name (partition_column_name)
      | filegroup | "default" } ]
| FOREIGN KEY
  ( column [ ,...n ] )
  REFERENCES referenced_table_name [ ( ref_column [ ,...n ] ) ]
  [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
  [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
  [ NOT FOR REPLICATION ]
| CHECK [ NOT FOR REPLICATION ] ( logical_expression )

<table_index> ::=

INDEX index_name [ CLUSTERED | NONCLUSTERED ] (column [ ASC | DESC ] [ ,... n ] )

[ WITH ( <index_option> [ ,... n ] ) ]
[ ON { partition_scheme_name (column_name )
      | filegroup_name
      | default
      }
]
[ FILESTREAM_ON { filestream_filegroup_name | partition_scheme_name | "NULL" } ]

}

<table_option> ::=
{

[DATA_COMPRESSION = { NONE | ROW | PAGE }
 [ ON PARTITIONS ( { <partition_number_expression> | <range> }
 [ , ...n ] ) ]]
[ FILETABLE_DIRECTORY = <directory_name> ]
[ FILETABLE_COLLATE_FILENAME = { <collation_name> | database_default } ]
[ FILETABLE_PRIMARY_KEY_CONSTRAINT_NAME = <constraint_name> ]
[ FILETABLE_STREAMID_UNIQUE_CONSTRAINT_NAME = <constraint_name> ]
[ FILETABLE_FULLPATH_UNIQUE_CONSTRAINT_NAME = <constraint_name> ]
}

<index_option> ::=
{
  PAD_INDEX = { ON | OFF }
  | FILLFACTOR = fillfactor
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF}
  | ALLOW_PAGE_LOCKS ={ ON | OFF}
  | DATA_COMPRESSION = { NONE | ROW | PAGE }
    [ ON PARTITIONS ( { <partition_number_expression> | <range> }
    [ , ...n ] ) ]
}

<range> ::=
<partition_number_expression> TO <partition_number_expression>

```



Y en MySQL la sintaxis completa es:

```

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
  (create_definition,...)
  [table_options]
  [partition_options]

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
  [(create_definition,...)]
  [table_options]
  [partition_options]
  [IGNORE | REPLACE]
  [AS] query_expression

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
  { LIKE old_tbl_name | (LIKE old_tbl_name) }

create_definition: {
  col_name column_definition
  | {INDEX | KEY} [index_name] [index_type] (key_part,...)
    [index_option] ...
  | {FULLTEXT | SPATIAL} [INDEX | KEY] [index_name] (key_part,...)
    [index_option] ...
  | [CONSTRAINT [symbol]] PRIMARY KEY
    [index_type] (key_part,...)
    [index_option] ...
  | [CONSTRAINT [symbol]] UNIQUE [INDEX | KEY]
    [index_name] [index_type] (key_part,...)
    [index_option] ...
  | [CONSTRAINT [symbol]] FOREIGN KEY
    [index_name] (col_name,...)
    reference_definition
  | check_constraint_definition
}

column_definition: {
  data_type [NOT NULL | NULL] [DEFAULT {literal | (expr)} ]
  [VISIBLE | INVISIBLE]
  [AUTO_INCREMENT] [UNIQUE [KEY]] [[PRIMARY] KEY]
  [COMMENT 'string']
  [COLLATE collation_name]
  [COLUMN_FORMAT {FIXED | DYNAMIC | DEFAULT}]
  [ENGINE_ATTRIBUTE [=] 'string']
  [SECONDARY_ENGINE_ATTRIBUTE [=] 'string']
  [STORAGE {DISK | MEMORY}]
  [reference_definition]
  [check_constraint_definition]
  | data_type
    [COLLATE collation_name]
    [GENERATED ALWAYS AS (expr)]
    [VIRTUAL | STORED] [NOT NULL | NULL]
    [VISIBLE | INVISIBLE]
    [UNIQUE [KEY]] [[PRIMARY] KEY]
    [COMMENT 'string']
    [reference_definition]
    [check_constraint_definition]
}

```



```

data_type:
  (Ver Categorías de Tipos de Datos o Documentación online Data Types)

key_part: {col_name [(Length)] | (expr)} [ASC | DESC]

index_type:
  USING {BTREE | HASH}

index_option: {
  KEY_BLOCK_SIZE [=] value
  index_type
  WITH PARSER parser_name
  COMMENT 'string'
  {VISIBLE | INVISIBLE}
  ENGINE_ATTRIBUTE [=] 'string'
  SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
}

check_constraint_definition:
  [CONSTRAINT [symbol]] CHECK (expr) [[NOT] ENFORCED]

reference_definition:
  REFERENCES tbl_name (key_part,...)
    [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
    [ON DELETE reference_option]
    [ON UPDATE reference_option]

reference_option:
  RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT

table_options:
  table_option [[,] table_option] ...

table_option: {
  AUTOEXTEND_SIZE [=] value
  AUTO_INCREMENT [=] value
  AVG_ROW_LENGTH [=] value
  [DEFAULT] CHARACTER SET [=] charset_name
  CHECKSUM [=] {0 | 1}
  [DEFAULT] COLLATE [=] collation_name
  COMMENT [=] 'string'
  COMPRESSION [=] {'ZLIB' | 'LZ4' | 'NONE'}
  CONNECTION [=] 'connect_string'
  {DATA | INDEX} DIRECTORY [=] 'absolute path to directory'
  DELAY_KEY_WRITE [=] {0 | 1}
  ENCRYPTION [=] {'Y' | 'N'}
  ENGINE [=] engine_name
  ENGINE_ATTRIBUTE [=] 'string'
  INSERT_METHOD [=] {NO | FIRST | LAST}
  KEY_BLOCK_SIZE [=] value
  MAX_ROWS [=] value
  MIN_ROWS [=] value
  PACK_KEYS [=] {0 | 1 | DEFAULT}
  PASSWORD [=] 'string'
  ROW_FORMAT [=] {DEFAULT | DYNAMIC | FIXED | COMPRESSED | REDUNDANT | COMPACT}
  SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
  STATS_AUTO_RECALC [=] {DEFAULT | 0 | 1}
  STATS_PERSISTENT [=] {DEFAULT | 0 | 1}
  STATS_SAMPLE_PAGES [=] value
  TABLESPACE tablespace_name [STORAGE {DISK | MEMORY}]
}

```



```

| UNION [=] (tbl_name[,tbl_name]...)
}

partition_options:
    PARTITION BY
        { [LINEAR] HASH(expr)
        | [LINEAR] KEY [ALGORITHM={1 | 2}] (column_list)
        | RANGE{expr) | COLUMNS(column_list)}
        | LIST{expr) | COLUMNS(column_list)} }
    [PARTITIONS num]
    [SUBPARTITION BY
        { [LINEAR] HASH(expr)
        | [LINEAR] KEY [ALGORITHM={1 | 2}] (column_list) }
    [SUBPARTITIONS num]
    ]
    [(partition_definition [, partition_definition] ...)]
}

partition_definition:
    PARTITION partition_name
        [VALUES
            {LESS THAN {(expr | value_list) | MAXVALUE}
            |
            IN (value_list)}]
        [[STORAGE] ENGINE [=] engine_name]
        [COMMENT [=] 'string']
        [DATA DIRECTORY [=] 'data_dir']
        [INDEX DIRECTORY [=] 'index_dir']
        [MAX_ROWS [=] max_number_of_rows]
        [MIN_ROWS [=] min_number_of_rows]
        [TABLESPACE [=] tablespace_name]
        [(subpartition_definition [, subpartition_definition] ...)]]

subpartition_definition:
    SUBPARTITION logical_name
        [[STORAGE] ENGINE [=] engine_name]
        [COMMENT [=] 'string']
        [DATA DIRECTORY [=] 'data_dir']
        [INDEX DIRECTORY [=] 'index_dir']
        [MAX_ROWS [=] max_number_of_rows]
        [MIN_ROWS [=] min_number_of_rows]
        [TABLESPACE [=] tablespace_name]

query_expression:
    SELECT ... (Some valid select or union statement)

```

Como podemos ver las sintaxis son muy completas y variadas, hay algunas diferencias entre los distintos sistemas de gestión, pero siempre la base es el estándar SQL.

Por ejemplo, para crear una tabla llamada *Productos*, cuyos atributos son *ProductoID*, *NombreProducto*, *Precio*, *DescripcionProducto*, donde *ProductoID* es la clave principal y de tipo int. El resto de los campos son de tipo varchar(25), money (en MySQL no existe el tipo money, así que lo reemplazamos por su equivalente) y varchar(max) (en MySQL no existe la opción max, así que vamos a darle un número) respectivamente.



SQL Server

```
CREATE TABLE Productos
(ProductoID int PRIMARY KEY NOT NULL,
NombreProducto varchar(25) NOT NULL,
Precio money NULL,
DescripcionProducto varchar(max) NULL);
```

MySQL

```
CREATE TABLE Productos
(ProductoID int PRIMARY KEY NOT NULL,
NombreProducto varchar(25) NOT NULL,
Precio decimal(19,4) NULL,
DescripcionProducto varchar(8000) NULL);
```

DROP TABLE

La sintaxis básica es:

```
DROP TABLE < tabla >
```

La sintaxis completa es:

```
DROP TABLE [ database_name . [ schema_name ] . | schema_name . ]
table_name [ ,...n ]
[ ; ]
```

Por ejemplo, para eliminar la tabla que creamos antes, el comando sería:

```
DROP TABLE Productos;
```

ALTER TABLE

La sintaxis básica para agregar un campo es:

```
ALTER TABLE < tabla > ADD (atributo1 dominio1 [NOT NULL]...)
```

La sintaxis completa en SQL Server es:

```
ALTER TABLE [ database_name . [ schema_name ] . | schema_name . ] table_name
{
    ALTER COLUMN column_name
    {
        [ type_schema_name. ] type_name
        [ ( 
            {
                precision [ , scale ]
                | max
                | xml_schema_collection
            }
        ) ]
        [ COLLATE collation_name ]
        [ NULL | NOT NULL ] [ SPARSE ]
        | {ADD | DROP }
        { ROWGUIDCOL | PERSISTED | NOT FOR REPLICATION | SPARSE }
    }
    | [ WITH { CHECK | NOCHECK } ]

    | ADD
    {
        <column_definition>
        | <computed_column_definition>
        | <table_constraint>
    }
}
```



```

| <column_set_definition>
} [ ,...n ]

| DROP
{
    [ CONSTRAINT ]
    {
        constraint_name
        [ WITH
            ( <drop_clustered_constraint_option> [ ,...n ] )
        ]
    } [ ,...n ]
    | COLUMN
    {
        column_name
    } [ ,...n ]
} [ ,...n ]
| [ WITH { CHECK | NOCHECK } ] { CHECK | NOCHECK } CONSTRAINT
{ ALL | constraint_name [ ,...n ] }

| { ENABLE | DISABLE } TRIGGER
{ ALL | trigger_name [ ,...n ] }

| { ENABLE | DISABLE } CHANGE_TRACKING
[ WITH ( TRACK_COLUMNS_UPDATED = { ON | OFF } ) ]

| SWITCH [ PARTITION source_partition_number_expression ]
TO target_table
[ PARTITION target_partition_number_expression ]
[ WITH ( <low_lock_priority_wait> ) ]
| SET ( FILESTREAM_ON =
    { partition_scheme_name | filegroup | "default" | "NULL" }
)

| REBUILD
[ [PARTITION = ALL]
[ WITH ( <rebuild_option> [ ,...n ] ) ]
| [ PARTITION = partition_number
    [ WITH ( <single_partition_rebuild_option> [ ,...n ] ) ]
]
]

| <table_option>

| <filetable_option>

}
[ ; ]

<column_set_definition> ::=

    column_set_name XML COLUMN_SET FOR ALL_SPARSE_COLUMNS

<drop_clustered_constraint_option> ::=
{
    MAXDOP = max_degree_of_parallelism
    | ONLINE = { ON | OFF }
    | MOVE TO

```



```

        { partition_scheme_name ( column_name ) | filegroup | "default" }
    }
<table_option> ::==
{
    SET ( LOCK_ESCALATION = { AUTO | TABLE | DISABLE } )
}

<filetable_option> ::=
{
    [ { ENABLE | DISABLE } FILETABLE_NAMESPACE ]
    [ SET ( FILETABLE_DIRECTORY = directory_name ) ]
}

<single_partition_rebuild_option> ::=
{
    SORT_IN_TEMPDB = { ON | OFF }
    | MAXDOP = max_degree_of_parallelism
    | DATA_COMPRESSION = { NONE | ROW | PAGE | COLUMNSTORE | COLUMNSTORE_ARCHIVE }
    | ONLINE = { ON [( <low_priority_lock_wait> ) ] | OFF }
}

<low_priority_lock_wait>::=
{
    WAIT_AT_LOW_PRIORITY ( MAX_DURATION = <time> [ MINUTES ], ABORT_AFTER_WAIT = { NONE
    | SELF | BLOCKERS } )
}

```

Y la sintaxis completa en MySQL es:

```

ALTER TABLE tbl_name
[alter_option [, alter_option] ...]
[partition_options]

alter_option: {
    table_options
    | ADD [COLUMN] col_name column_definition
      [FIRST | AFTER col_name]
    | ADD [COLUMN] (col_name column_definition,...)
    | ADD {INDEX | KEY} [index_name]
      [index_type] (key_part,...) [index_option] ...
    | ADD {FULLTEXT | SPATIAL} [INDEX | KEY] [index_name]
      (key_part,...) [index_option] ...
    | ADD [CONSTRAINT [symbol]] PRIMARY KEY
      [index_type] (key_part,...)
      [index_option] ...
    | ADD [CONSTRAINT [symbol]] UNIQUE [INDEX | KEY]
      [index_name] [index_type] (key_part,...)
      [index_option] ...
    | ADD [CONSTRAINT [symbol]] FOREIGN KEY
      [index_name] (col_name,...)
      reference_definition
    | ADD [CONSTRAINT [symbol]] CHECK (expr) [[NOT] ENFORCED]
    | DROP {CHECK | CONSTRAINT} symbol
    | ALTER {CHECK | CONSTRAINT} symbol [NOT] ENFORCED
    | ALGORITHM [=] {DEFAULT | INSTANT | INPLACE | COPY}
    | ALTER [COLUMN] col_name {
        SET DEFAULT {literal | (expr)}
        | SET {VISIBLE | INVISIBLE}
        | DROP DEFAULT
    }
}

```



```

ALTER INDEX index_name {VISIBLE | INVISIBLE}
| CHANGE [COLUMN] old_col_name new_col_name column_definition
|   [FIRST | AFTER col_name]
| [DEFAULT] CHARACTER SET [=] charset_name [COLLATE [=] collation_name]
| CONVERT TO CHARACTER SET charset_name [COLLATE collation_name]
| {DISABLE | ENABLE} KEYS
| {DISCARD | IMPORT} TABLESPACE
| DROP [COLUMN] col_name
| DROP {INDEX | KEY} index_name
| DROP PRIMARY KEY
| DROP FOREIGN KEY fk_symbol
| FORCE
| LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE}
| MODIFY [COLUMN] col_name column_definition
|   [FIRST | AFTER col_name]
| ORDER BY col_name [, col_name] ...
| RENAME COLUMN old_col_name TO new_col_name
| RENAME {INDEX | KEY} old_index_name TO new_index_name
| RENAME [TO | AS] new_tbl_name
| {WITHOUT | WITH} VALIDATION
}

partition_options:
    partition_option [partition_option] ...

partition_option: {
    ADD PARTITION (partition_definition)
    DROP PARTITION partition_names
    DISCARD PARTITION {partition_names | ALL} TABLESPACE
    IMPORT PARTITION {partition_names | ALL} TABLESPACE
    TRUNCATE PARTITION {partition_names | ALL}
    COALESCE PARTITION number
    REORGANIZE PARTITION partition_names INTO (partition_definitions)
    EXCHANGE PARTITION partition_name WITH TABLE tbl_name [{WITH | WITHOUT} VALIDATION]
    ANALYZE PARTITION {partition_names | ALL}
    CHECK PARTITION {partition_names | ALL}
    OPTIMIZE PARTITION {partition_names | ALL}
    REBUILD PARTITION {partition_names | ALL}
    REPAIR PARTITION {partition_names | ALL}
    REMOVE PARTITIONING
}

key_part: {col_name [(Length)] | (expr)} [ASC | DESC]

index_type:
    USING {BTREE | HASH}

index_option: {
    KEY_BLOCK_SIZE [=] value
    | index_type
    | WITH PARSER parser_name
    | COMMENT 'string'
    | {VISIBLE | INVISIBLE}
}

table_options:
    table_option [[,] table_option] ...

table_option: {
    AUTOEXTEND_SIZE [=] value
}

```



```

AUTO_INCREMENT [=] value
AVG_ROW_LENGTH [=] value
[DEFAULT] CHARACTER SET [=] charset_name
CHECKSUM [=] {0 | 1}
[DEFAULT] COLLATE [=] collation_name
COMMENT [=] 'string'
COMPRESSION [=] {'ZLIB' | 'LZ4' | 'NONE'}
CONNECTION [=] 'connect_string'
{DATA | INDEX} DIRECTORY [=] 'absolute path to directory'
DELAY_KEY_WRITE [=] {0 | 1}
ENCRYPTION [=] {'Y' | 'N'}
ENGINE [=] engine_name
ENGINE_ATTRIBUTE [=] 'string'
INSERT_METHOD [=] {NO | FIRST | LAST}
KEY_BLOCK_SIZE [=] value
MAX_ROWS [=] value
MIN_ROWS [=] value
PACK_KEYS [=] {0 | 1 | DEFAULT}
PASSWORD [=] 'string'
ROW_FORMAT [=] {DEFAULT | DYNAMIC | FIXED | COMPRESSED | REDUNDANT | COMPACT}
SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
STATS_AUTO_RECALC [=] {DEFAULT | 0 | 1}
STATS_PERSISTENT [=] {DEFAULT | 0 | 1}
STATS_SAMPLE_PAGES [=] value
TABLESPACE tablespace_name [STORAGE {DISK | MEMORY}]
UNION [=] (tbl_name[,tbl_name]...)
}

partition_options:
(see CREATE TABLE options)

```

Básicamente, ALTER TABLE permite:

- 1) Agregar, eliminar o modificar atributos
- 2) Agregar y eliminar restricciones (constraints)
- 3) Habilitar y Deshabilitar restricciones

PARA AGREGAR UNA COLUMNA

ALTER TABLE nombre_tabla ADD nombre_columna tipo;

PARA ELIMINAR UNA COLUMNA

ALTER TABLE nombre_tabla DROP nombre_columna;

PARA MODIFICAR UNA COLUMNA

ALTER TABLE table_name ALTER COLUMN column_name datatype;



DML

Para los ejemplos se utilizará la base de ejemplo Northwind

```
USE northwind;
```

SELECCIONANDO TODAS LAS COLUMNAS Y TODAS LAS FILAS

SINTAXIS

SQL Server

```
SELECT tabla.*  
FROM tabla;  
    -- 0  
SELECT *  
FROM tabla;  
    -- 0  
SELECT *  
FROM [base].[esquema].tabla;
```

MySQL

```
SELECT tabla.*  
FROM tabla;  
    -- 0  
SELECT *  
FROM tabla;  
    -- 0  
SELECT *  
FROM [base].tabla;
```

EJEMPLO

```
/* Obtener todas las columnas de la tabla Region*/
```

```
SELECT *  
FROM Region;
```

```
SELECT Region.*  
FROM Region;
```

SQL Server

```
SELECT *  
FROM Northwind.dbo.Region;
```

MySQL

```
SELECT *  
FROM northwind.Region;
```

RegionID	RegionDescription

1	Eastern
2	Western
3	Northern
4	Southern

SELECCIONANDO COLUMNAS ESPECÍFICAS

SINTAXIS

```
SELECT tabla.nombre_columna1, tabla.nombre_columna2  
FROM tabla;  
    -- 0  
SELECT nombre_columna1, nombre_columna2  
FROM tabla;
```



EJEMPLO

/ Selecciona nombre (FirstName) y apellido (LastName) de la tabla Empleados (Employees) */*

```
SELECT Employees.FirstName, Employees.LastName  
FROM Employees;
```

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

FirstName	LastName
Nancy	Davolio
Andrew	Fuller
Janet	Leverling
Margaret	Peacock
Steven	Buchanan
Michael	Suyama
Robert	King
Laura	Callahan
Anne	Dodsworth

ORDENANDO REGISTROS

Se utiliza la cláusula ORDER BY de la sentencia SELECT.

ORDENANDO POR UNA COLUMNA - SINTAXIS

```
SELECT columna1, columna2  
FROM tabla  
ORDER BY columna1;
```

EJEMPLO

/ Selecciona nombre (FirstName) y apellido (LastName) de la tabla Empleados (Employees).
Ordena por apellido. */*

```
SELECT FirstName, LastName  
FROM Employees  
ORDER BY LastName;
```

FirstName	LastName
Steven	Buchanan
Laura	Callahan
Nancy	Davolio
Anne	Dodsworth
Andrew	Fuller
Robert	King
Janet	Leverling
Margaret	Peacock
Michael	Suyama



ORDENAR POR VARIAS COLUMNAS - SINTAXIS

```
SELECT columna1, columna2
FROM tabla
ORDER BY columna1, columna2;
```

EJEMPLO

```
/* Selecciona puesto (Title), nombre (FirstName) y apellido (LastName) de la tabla Empleados
(Employees).
Ordena por puesto y luego por apellido. */
```

```
SELECT Title, FirstName, LastName
FROM Employees
ORDER BY Title, LastName;
```

Title	FirstName	LastName
Inside Sales Coordinator	Laura	Callahan
Sales Manager	Steven	Buchanan
Sales Representative	Nancy	Davolio
Sales Representative	Anne	Dodsworth
Sales Representative	Robert	King
Sales Representative	Janet	Leverling
Sales Representative	Margaret	Peacock
Sales Representative	Michael	Suyama
Vice President, Sales	Andrew	Fuller

ORDENAR POR POSICIÓN DE LA COLUMNA - SINTAXIS

```
SELECT columna1, columna2
FROM tabla
ORDER BY posicion_columna1, posicion_columna2;
```

EJEMPLO

```
/* Selecciona puesto (Title), nombre (FirstName) y apellido (LastName) de la tabla Empleados
(Employees).
Ordena por puesto (posición 1) y luego por apellido (posición 3). */
```

```
SELECT Title, FirstName, LastName
FROM Employees
ORDER BY 1, 3;
```

Title	FirstName	LastName
Inside Sales Coordinator	Laura	Callahan
Sales Manager	Steven	Buchanan
Sales Representative	Nancy	Davolio
Sales Representative	Anne	Dodsworth
Sales Representative	Robert	King
Sales Representative	Janet	Leverling
Sales Representative	Margaret	Peacock
Sales Representative	Michael	Suyama
Vice President, Sales	Andrew	Fuller



ORDEN ASCENDENTE Y DESCENDENTE

Cuando se utiliza ORDER BY los registros son ordenados de manera Ascendente por defecto. Esto puede ser especificado explícitamente con la palabra clave ASC. Para ordenar en orden descendente se utiliza DESC

SINTAXIS

```
SELECT columna1, columna2
FROM tabla
ORDER BY posicion_columna1 DESC, posicion_columna2 ASC;
```

EJEMPLO

```
/* Selecciona puesto (Title), nombre (FirstName) y apellido (LastName) de la tabla Empleados
(Employees).
Ordena por puesto ascendente y luego por apellido descendente. */


```

```
SELECT Title, FirstName, LastName
FROM Employees
ORDER BY Title ASC, LastName DESC;
```

Title	FirstName	LastName
Inside Sales Coordinator	Laura	Callahan
Sales Manager	Steven	Buchanan
Sales Representative	Michael	Suyama
Sales Representative	Margaret	Peacock
Sales Representative	Janet	Leverling
Sales Representative	Robert	King
Sales Representative	Anne	Dodsworth
Sales Representative	Nancy	Davolio
Vice President, Sales	Andrew	Fuller

LA CLÁUSULA WHERE Y LOS SÍMBOLOS OPERADORES

La cláusula WHERE se utiliza para obtener filas específicas de la tabla. Puede contener una o más condiciones que especifican cuáles son las filas que deben ser devueltas.

SINTAXIS

```
SELECT columna1, columna2
FROM tabla
WHERE condiciones;
```

En las condiciones pueden ser utilizados los siguientes operadores:

=	Igual a
<>	No Igual a
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que



Cuando se utilicen valores no numéricos (por ejemplo, fechas o cadenas de texto) en la cláusula WHERE deben ser encerrados entre comillas simples.

VERIFICANDO POR IGUALDAD - EJEMPLO

```
/* Selecciona puesto (Title), nombre (FirstName) y apellido (LastName) de la tabla Empleados (Employees) de todos los representantes de ventas (Sales Representative). */
```

```
SELECT Title, FirstName, LastName
FROM Employees
WHERE Title = 'Sales Representative';
```

Title	FirstName	LastName
Sales Representative	Nancy	Davolio
Sales Representative	Janet	Leverling
Sales Representative	Margaret	Peacock
Sales Representative	Michael	Suyama
Sales Representative	Robert	King
Sales Representative	Anne	Dodsworth

VERIFICANDO POR NO IGUALDAD – EJEMPLO

```
/* Selecciona nombre (FirstName) y apellido (LastName) de la tabla Empleados (Employees) excluyendo a los representantes de ventas (Sales Representative). */
```

```
SELECT FirstName, LastName
FROM Employees
WHERE Title <> 'Sales Representative';
```

FirstName	LastName
Andrew	Fuller
Steven	Buchanan
Laura	Callahan

VERIFICANDO POR MAYOR O MENOR QUE

Estos operadores pueden utilizarse para comparar número, fechas y cadenas de texto

```
/* Seleccionar nombre (FirstName) y apellido (LastName) de los empleados (Employees) cuyo apellido comience con N en adelante */
```

```
SELECT FirstName, LastName
FROM Employees
WHERE LastName >= 'N';
```

FirstName	LastName
Margaret	Peacock
Michael	Suyama

VERIFICANDO POR NULL

Se dice que un valor el nulo (NULL) cuando el campo no tiene un valor asignado. No es lo mismo que una cadena vacía. Lo que significa es que el campo no tiene ningún valor asignado.



Para verificar si una columna tiene valor nulo no se utiliza el operador “=”, en su lugar se utiliza la expresión **IS NULL**.

EJEMPLO

```
/* Seleccionar nombre (FirstName) y apellido (LastName) de todos los empleados (Employees)
que no tengan especificada una región (Region). */
```

```
SELECT FirstName, LastName
FROM Employees
WHERE Region IS NULL;
```

FirstName	LastName
Steven	Buchanan
Michael	Suyama
Robert	King
Anne	Dodsworth

```
/* Seleccionar nombre (FirstName) y apellido (LastName) de todos los empleados (Employees)
que tengan especificada una región (Region). */
```

```
SELECT FirstName, LastName
FROM Employees
WHERE Region IS NOT NULL;
```

FirstName	LastName
Nancy	Davolio
Andrew	Fuller
Janet	Leverling
Margaret	Peacock
Laura	Callahan

COMBINANDO WHERE Y ORDER BY

Cuando los utilizamos juntos, la cláusula WHERE debe ir antes de la cláusula ORDER BY.

EJEMPLO

```
SELECT FirstName, LastName
FROM Employees
WHERE LastName >= 'N'
ORDER BY LastName DESC;
```

FirstName	LastName
Michael	Suyama
Margaret	Peacock

LA CLÁUSULA WHERE Y LAS PALABRAS OPERADORES

BETWEEN	Devuelve valores en un rango inclusivo
IN	Devuelve valores que se encuentran en un conjunto específico
LIKE	Devuelve valores que coinciden con un patrón simple
NOT	Niega una operación



EJEMPLOS

```
/* Seleccionar nombre y apellido de todos los empleados cuyos nombres comiencen con una letra entre "J" y "M". */
```

```
SELECT FirstName, LastName
FROM Employees
WHERE LastName BETWEEN 'J' AND 'M';
```

-- Lo anterior es equivalente a lo siguiente.

```
SELECT FirstName, LastName
FROM Employees
WHERE LastName >= 'J' AND LastName <= 'M';
```

FirstName	LastName
Janet	Leverling
Robert	King

```
/* Seleccionar saludo (TitleOfCourtesy), nombre (FirstName) y apellido (LastName) de todos los empleados cuyo saludo sea "Mrs." o "Ms.". */
```

```
SELECT TitleOfCourtesy, FirstName, LastName
FROM Employees
WHERE TitleOfCourtesy IN ('Ms.', 'Mrs.');
```

-- o su equivalente

```
SELECT TitleOfCourtesy, FirstName, LastName
FROM Employees
WHERE TitleOfCourtesy = 'Ms.' OR TitleOfCourtesy = 'Mrs.';
```

TitleOfCourtesy	FirstName	LastName
Ms.	Nancy	Davolio
Ms.	Janet	Leverling
Mrs.	Margaret	Peacock
Ms.	Laura	Callahan
Ms.	Anne	Dodsworth

EL OPERADOR LIKE

Se utiliza para verificar si un campo coincide con una determinada expresión o patrón.

En estas expresiones se utilizan caracteres comodines.

% (porcentaje): representa una cadena de cualquier tamaño, incluyendo la cadena vacía.

_ (guion bajo): representa un único carácter

En SQL Server también pueden utilizarse los corchetes para definir una lista de caracteres.

[lista de caracteres]: le puede definir una lista de caracteres entre corchetes, que representan a un único carácter.

[carácter_desde-carácter_hasta]: representa un único carácter, comprendido en el rango establecido entre corchetes.



[^carácter_desde-carácter_hasta]: representa un único carácter, que no esté en el rango establecido. Es la negación de la expresión anterior.

En MySQL no se pueden usar estas expresiones, pero MySQL cuenta con un operador llamado RLIKE que permite realizar comparaciones de patrones utilizando expresiones regulares, lo que permite utilizar patrones con mayor complejidad.

Carácter de escape: si en alguna expresión se quiere utilizar un carácter que puede ser utilizado como comodín (_ %, [,]), se lo antecede con el carácter de escape, para indicar que no está funcionando como comodín, si no como un carácter normal. Se lo declara luego de la expresión, por ejemplo, col1 LIKE '%!_%' ESCAPE '!'.

Los comodines % y _ se pueden escapar también colocándolos entre corchetes, por ejemplo, col1 LIKE '%[_]%'.

EJEMPLOS

```
/* Seleccionar saludo (TitleOfCourtesy), nombre (FirstName) y apellido (LastName) de todos los empleados cuyo saludo comience con "M". */
```

```
SELECT TitleOfCourtesy, FirstName, LastName
FROM Employees
WHERE TitleOfCourtesy LIKE 'M%';
```

TitleOfCourtesy	FirstName	LastName
Ms.	Nancy	Davolio
Ms.	Janet	Leverling
Mrs.	Margaret	Peacock
Mr.	Steven	Buchanan
Mr.	Michael	Suyama
Mr.	Robert	King
Ms.	Laura	Callahan
Ms.	Anne	Dodsworth

```
/* Seleccionar saludo (TitleOfCourtesy), nombre (FirstName) y apellido (LastName) de todos los empleados cuyo saludo comience con una "M" seguida de cualquier carácter y luego un punto (.) */
```

```
SELECT TitleOfCourtesy, FirstName, LastName
FROM Employees
WHERE TitleOfCourtesy LIKE 'M_.';
```

TitleOfCourtesy	FirstName	LastName
Ms.	Nancy	Davolio
Ms.	Janet	Leverling
Mr.	Steven	Buchanan
Mr.	Michael	Suyama
Mr.	Robert	King
Ms.	Laura	Callahan
Ms.	Anne	Dodsworth

El uso de comodines puede reducir la performance de las consultas, especialmente si se utilizan al comienzo de la expresión. Deben ser utilizados con moderación.

EL OPERADOR NOT

Se utiliza para negar una condición o una operación.



```
/* Seleccionar saludo (TitleOfCourtesy), nombre (FirstName) y apellido (LastName) de todos los empleados cuyo saludo no sea "Ms." o "Mrs.". */
```

```
SELECT TitleOfCourtesy, FirstName, LastName
FROM Employees
WHERE NOT TitleOfCourtesy IN ('Ms.', 'Mrs.');
```

```
-- O
SELECT TitleOfCourtesy, FirstName, LastName
FROM Employees
WHERE TitleOfCourtesy NOT IN ('Ms.', 'Mrs.');
```

TitleOfCourtesy	FirstName	LastName
Dr.	Andrew	Fuller
Mr.	Steven	Buchanan
Mr.	Michael	Suyama
Mr.	Robert	King

VERIFICANDO MÚLTIPLES CONDICIONES

AND

Se utiliza en la cláusula WHERE para encontrar los registros que coincidan con más de una condición.

```
/* Seleccionar nombre (FirstName) y apellido (LastName) de todos los representantes de ventas (Sales Representative) cuyo saludo (TitleOfCourtesy) es "Mr.". */
```

```
SELECT FirstName, LastName
FROM Employees
WHERE Title = 'Sales Representative' AND TitleOfCourtesy = 'Mr.';
```

FirstName	LastName
Michael	Suyama
Robert	King

OR

Se utiliza en la cláusula WHERE para encontrar los registros que coincidan con al menos una condición.

```
/* Seleccionar nombre (FirstName), apellido (LastName) y ciudad (City) de todos los empleados de la ciudad de Seattle o Redmond. */
```

```
SELECT FirstName, LastName, City
FROM Employees
WHERE City = 'Seattle' OR City = 'Redmond';
```

FirstName	LastName	City
Nancy	Davolio	Seattle
Margaret	Peacock	Redmond
Laura	Callahan	Seattle



ORDEN DE EVALUACIÓN

Por defecto, SQL procesa los operadores AND antes de los operadores OR. Para ilustrar cómo funciona esto, veamos el siguiente ejemplo.

```
/* Seleccionar nombre (FirstName), apellido (LastName), ciudad (City) y puesto (Title) de todos los representantes de ventas (Sales Representative) que sean de la ciudad de Seattle o Redmond. */
```

```
SELECT FirstName, LastName, City, Title
FROM Employees
WHERE City = 'Seattle' OR City = 'Redmond' AND Title = 'Sales Representative';
```

FirstName	LastName	City	Title
Nancy	Davolio	Seattle	Sales Representative
Margaret	Peacock	Redmond	Sales Representative
Laura	Callahan	Seattle	Inside Sales Coordinator

Noten que Laura no es representante de ventas (Sales Representative) y sin embargo está incluida en los resultados.

Esto es porque la consulta está devolviendo empleados de Seattle O representantes de ventas de Redmond.

Para solucionar esto, colocaremos la condición OR entre paréntesis.

```
/* Seleccionar nombre (FirstName), apellido (LastName), ciudad (City) y puesto (Title) de todos los representantes de ventas (Sales Representative) que sean de la ciudad de Seattle o Redmond. */
```

```
SELECT FirstName, LastName, City, Title
FROM Employees
WHERE (City = 'Seattle' OR City = 'Redmond') AND Title = 'Sales Representative';
```

FirstName	LastName	City	Title
Nancy	Davolio	Seattle	Sales Representative
Margaret	Peacock	Redmond	Sales Representative

De esta manera se evalúa primero la expresión OR y el resultado logrado es el buscado.

Se recomienda entonces el uso de paréntesis en todo caso en el que la precedencia de operadores puede resultar ambigua.

CAMPOS CALCULADOS

Los campos calculados son campos que no existen en las tablas, son creados en la sentencia SELECT. Por ejemplo, uno podría crear el campo “Nombre Completo” concatenando “Nombre” y “Apellido”.

CONCATENACIÓN

La concatenación consiste en encadenar diferentes palabras o caracteres.

SQL Server proporciona el operador de signo más (+), la función CONCAT y la función CONCAT_WS para concatenar cadenas.



MySQL proporciona la función CONCAT y la función CONCAT_WS para concatenar cadenas. El operador de signo más (+) queda reservado para operaciones aritméticas.

El estándar incluye al doble pipe (||) como operador de concatenación, pero SQL Server no lo implementa, y en MySQL no viene habilitado por defecto.

SINTAXIS

SQL Server

```
SELECT columna1/expresión + columna2/expresión [+ columnan/expresión ]
```

SQL Server y MySQL

```
SELECT CONCAT(columna1/expresión, columna2/expresión [,columnan/expresión])
```

```
SELECT CONCAT_WS(separador, columna1/expresión, columna2/expresión [,columnan/expresión])
```

EJEMPLO

SQL Server

```
SELECT FirstName + ' ' + LastName  
FROM Employees;
```

SQL Server y MySQL

```
SELECT CONCAT(FirstName, ' ', LastName)  
FROM Employees;
```

```
SELECT CONCAT_WS(' ', FirstName, LastName)  
FROM Employees;
```

Nancy Davolio
Andrew Fuller
Janet Leverling
Margaret Peacock
Steven Buchanan
Michael Suyama
Robert King
Laura Callahan
Anne Dodsworth

La concatenación funciona sólo con cadenas de texto. Para concatenar otros tipos de datos, hay que convertirlos primero a cadena de texto.



CÁLCULOS MATEMÁTICOS

SQL Server

+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo

MySQL

+	Suma
-	Resta
*	Multiplicación
/	División
DIV	División Entera
% o MOD	Módulo

/* Si el costo de flete (Freight) es mayor o igual a \$500.00, hay que aplicarle un impuesto del 10%. Crear un reporte que muestre ID de la orden (OrderID), costo de flete (Freight), y costo de flete con el impuesto para las órdenes con costo de flete de \$500 o más. */

```
SELECT OrderID, Freight, Freight * 1.1
FROM Orders
WHERE Freight >= 500
```

OrderID Freight

10372	890,78	979.85800
10479	708,95	779.84500
10514	789,95	868.94500
10540	1007,64	1108.40400
10612	544,08	598.48800
10691	810,05	891.05500
10816	719,78	791.75800
10897	603,54	663.89400
10912	580,91	639.00100
10983	657,54	723.29400
11017	754,26	829.68600
11030	830,75	913.82500
11032	606,19	666.80900

ALIAS

Como habrán notado, las columnas correspondientes a las funciones no tienen título o nombre de columna. Mediante la palabra clave AS se puede agregar un encabezado a estas columnas.

Tengan en cuenta que los alias no pueden ser utilizados en la cláusula WHERE.

/* Si el costo de flete (Freight) es mayor o igual a \$500.00, hay que aplicarle un impuesto del 10%. Crear un reporte que muestre ID de la orden (OrderID), costo de flete (Freight), y costo de flete con el impuesto para las órdenes con costo de flete de \$500 o más. */

```
SELECT OrderID, Freight, Freight * 1.1 AS FleteTotal
FROM Orders
WHERE Freight >= 500;
```



OrderID	Freight	FleteTotal
10372	890,78	979.85800
10479	708,95	779.84500
10514	789,95	868.94500
10540	1007,64	1108.40400
10612	544,08	598.48800
10691	810,05	891.05500
10816	719,78	791.75800
10897	603,54	663.89400
10912	580,91	639.00100
10983	657,54	723.29400
11017	754,26	829.68600
11030	830,75	913.82500
11032	606,19	666.80900

FUNCIONES DE AGREGACIÓN Y AGRUPAMIENTO

Las funciones de agregación son utilizadas para calcular resultados utilizando campos de múltiples registros. Las cinco más comunes son:

COUNT()	Devuelve la cantidad de filas que contienen valores NO nulos en un campo específico
SUM()	Devuelve la suma
AVG()	Devuelve el promedio
MAX()	Devuelve el valor máximo
MIN()	Devuelve el valor mínimo

SQL Server cuenta además con las siguientes funciones:

APPROX_COUNT_DISTINCT()	Devuelve el número aproximado de valores únicos no nulos en un grupo.
CHECKSUM_AGG()	Devuelve un checksum del grupo. Puede ser utilizado para detectar cambios
COUNT_BIG()	Como el COUNT, pero devuelve un valor bigint en lugar int
GROUPING()	Devuelve 1 para agregados o 0 para no agregados
GROUPING_ID()	Devuelve el nivel de agrupación
STDEV() / STDEVP()	Devuelve el desvío estándar estadístico o poblacional
STRING_AGG()	Devuelve una cadena concatenada
VAR / VARP	Devuelve la varianza estadística o poblacional

MySQL cuenta además con las siguientes funciones:

BIT_AND()	Devuelve AND bit a bit
BIT_OR()	Devuelve OR bit a bit
BIT_XOR()	Devuelve XOR bit a bit
GROUP_CONCAT()	Devuelve una cadena concatenada
JSON_ARRAYAGG()	Devuelve resultado como un array JSON
JSON_OBJECTAGG()	Devuelve resultado como un objeto JSON
STDDEV() / STDDEV_POP()	Devuelve el desvío estándar estadístico o poblacional
VARIANCE() / VAR_POP()	Devuelve la varianza estadística o poblacional



EJEMPLOS

```
-- Cantidad de empleados
SELECT COUNT(*) AS NumEmpleados
FROM Employees;
```

NumEmpleados

9

-- Total de unidades (Quantity) ordenadas del producto 3

SQL Server

```
SELECT SUM(Quantity) AS TotalUnidades
FROM [Order Details]
WHERE ProductID = 3;
```

MySQL

```
SELECT SUM(Quantity) AS TotalUnidades
FROM `Order Details`
WHERE ProductID = 3;
```

TotalUnidades

328

-- Precio Unitario (UnitPrice) Promedio de los productos

```
SELECT AVG(UnitPrice) AS PrecioPromedio
FROM Products;
```

PrecioPromedio

28,8663

-- Encontrar la fecha de la primer y última contratación (HireDate) de empleados

```
SELECT MIN(HireDate) AS PrimerFechaAlta, MAX(HireDate) AS UltimaFechaAlta
FROM Employees;
```

PrimerFechaAlta UltimaFechaAlta

1992-04-01 00:00:00.000 1994-11-15 00:00:00.000

Las fechas son 1 de abril de 1992 y 15 de noviembre de 1994. Los formatos pueden variar entre distintas bases de datos e instalaciones, por las configuraciones de lenguaje. (Ver ANEXO: Trabajando con datos de fecha y hora en SQL SERVER y ANEXO: Trabajando con datos de fecha y hora en MySQL)

AGRUPANDO DATOS

Con GROUP BY las funciones de agrupación pueden ser aplicadas a grupos basados en los valores de sus campos. Por ejemplo, el número de empleados por ciudad:

```
SELECT City, COUNT(EmployeeID) AS NumEmpleados
FROM Employees
GROUP BY City;
```

City	NumEmpleados
Kirkland	1
London	4
Redmond	1
Seattle	2
Tacoma	1



HAVING

Mediante HAVING se pueden filtrar los resultados una vez que las funciones fueron calculadas.

EJEMPLO

```
/* Obtener el número de empleados de cada ciudad, en la que haya al menos dos empleados */
```

```
SELECT City, COUNT(EmployeeID) AS NumEmpleados
FROM Employees
GROUP BY City
HAVING COUNT(EmployeeID) > 1;
```

City	NumEmpleados
London	4
Seattle	2

ORDEN DE LAS CLÁUSULAS

1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. ORDER BY

EJEMPLO

```
/* Hallar el número (cantidad) de representantes de ventas en cada ciudad que cuenta con al menos dos. Ordenar según el número de empleados. */
```

```
SELECT City, COUNT(EmployeeID) AS NumEmpleados
FROM Employees
WHERE Title = 'Sales Representative'
GROUP BY City
HAVING COUNT(EmployeeID) > 1
ORDER BY NumEmpleados;
```

City	NumEmpleados
London	3

REGLAS DE AGRUPAMIENTO

- Cada columna no calculada que aparece en el SELECT debe aparecer también en el GROUP BY.
- No se pueden utilizar alias en el HAVING
- Se pueden utilizar alias en el ORDER BY
- Sólo se pueden utilizar campos calculados en el HAVING
- Se deben utilizar alias de campos calculados o campos reales en el ORDER BY



SELECCIONANDO REGISTROS SIN REPETICIÓN

La palabra clave DISTINCT se utiliza para seleccionar combinaciones distintas de los valores de las columnas de una tabla. Por ejemplo, a continuación, se muestran sin repetición las ciudades que tienen empleados.

EJEMPLO

```
/* Hallar las distintas ciudades que tienen empleados. */
```

```
SELECT DISTINCT City
FROM Employees
ORDER BY City;
```

City

```
-----
Kirkland
London
Redmond
Seattle
Tacoma
```

DISTINCT puede utilizarse con funciones agregadas.

EJEMPLO

```
/* Hallar cuantas ciudades diferentes tienen empleados. */
```

```
SELECT COUNT(DISTINCT City) AS NumCiudades
FROM Employees;
```

NumCiudades

```
-----
5
```

FUNCIONES DE MANIPULACIÓN INCLUIDAS

Mencionamos sólo algunas funciones básicas.

Descripción	SQL Server	MySQL
Valor Absoluto	ABS	ABS
Menor entero que sea >=	CEILING	CEIL / CEILING
Mayor entero que sea <=	FLOOR	FLOOR
Potencia	POWER	POWER
Redondeo	ROUND	ROUND
Raíz Cuadrada	SQRT	SQRT
Formatear números con dos decimales	CAST(num AS decimal(8,2))	FORMAT(num,2) O CAST(num AS decimal(8,2))



EJEMPLO

```
/* Obtener el costo de flete, como está y redondeado al primer decimal */
```

```
SELECT Freight, ROUND(Freight, 1) AS FleteRedondeado
FROM Orders;
```

SQL Server

Freight	FleteRedondeado
32,38	32,40
11,61	11,60
65,83	65,80
41,34	41,30
51,30	51,30
58,17	58,20
22,98	23,00
148,33	148,30
13,97	14,00
81,91	81,90
140,51	140,50
...	
6,19	6,20
38,28	38,30
8,53	8,50

MySQL

Freight	FleteRedondeado
32.3800	32.4
11.6100	11.6
65.8300	65.8
41.3400	41.3
51.3000	51.3
58.1700	58.2
22.9800	23.0
148.3300	148.3
13.9700	14.0
81.9100	81.9
140.5100	140.5
...	
6.1900	6.2
38.2800	38.3
8.5300	8.5

El formato de los números puede variar de acuerdo con la herramienta que se utiliza para realizar las consultas y su configuración. Si se requiere un formato específico, se deberán utilizar las funciones que las herramientas brindan para tales efectos.

```
/* Obtener el precio unitario como está y como CHAR(10) */
```

```
SELECT UnitPrice, CAST(UnitPrice AS char(10))
FROM Products;
```

SQL Server

UnitPrice
18,00
19,00
10,00
22,00
21,35
25,00
...
7,75
18,00
13,00

MySQL

UnitPrice CAST(UnitPrice AS char(10))
18.0000 18.0000
19.0000 19.0000
10.0000 10.0000
22.0000 22.0000
21.3500 21.3500
25.0000 25.0000
...
7.7500 7.7500
18.0000 18.0000
13.0000 13.0000



```
*****  
AGREGAR CONCATENACION  
*****
```

```
SELECT UnitPrice, CONCAT('$' , CAST(UnitPrice AS char(10)))  
FROM Products;
```

SQL Server

UnitPrice		
18,00	\$	18.00
19,00	\$	19.00
10,00	\$	10.00
22,00	\$	22.00
21,35	\$	21.35
25,00	\$	25.00
30,00	\$	30.00
...		
7,75	\$	7.75
18,00	\$	18.00
13,00	\$	13.00

MySQL

UnitPrice CONCAT('\$' , CAST(UnitPrice AS char(10)))		
18.0000 \$18.0000		
19.0000 \$19.0000		
10.0000 \$10.0000		
22.0000 \$22.0000		
21.3500 \$21.3500		
25.0000 \$25.0000		
30.0000 \$30.0000		
...		
7.7500 \$7.7500		
18.0000 \$18.0000		
13.0000 \$13.0000		

FUNCIONES DE MANIPULACIÓN DE CADENAS

Descripción	SQL Server	MySQL
Convertir a minúsculas	LOWER	LOWER
Convertir a mayúsculas	UPPER	UPPER
Remover espacios de atrás	RTRIM	RTRIM
Remover espacios de adelante	LTRIM	LTRIM
Subcadena	SUBSTRING	SUBSTRING

EJEMPLO

```
/* Seleccionar nombre (FirstName) y apellido (LastName) de los empleados y mostrar en mayúsculas */
```

```
SELECT UPPER(FirstName), UPPER(LastName)  
FROM Employees;
```

```
-----  
NANCY      DAVOLIO  
ANDREW     FULLER  
JANET      LEVERLING  
MARGARET   PEACOCK  
STEVEN     BUCHANAN  
MICHAEL    SUYAMA  
ROBERT     KING  
LAURA      CALLAHAN  
ANNE       DODSWORTH
```

```
-- Seleccionar los primeros 10 caracteres de cada dirección de cliente
```

```
SELECT SUBSTRING(Address, 1, 10)  
FROM Customers;
```



 Obere Str.
 Avda. de 1
 Mataderos
 120 Hanove
 Berguvväg
 Forsterstr
 24, place
 C/ Araquil
 12, rue de
 23 Tsawass

...
 305 - 14th
 Keskuskatu
 ul. Filtro

FUNCIONES DE FECHAS

El manejo de fechas es un tema complejo, y más aún porque los distintos motores implementan las funciones asociadas de manera diferente, con nombres diferentes, o con el mismo nombre pero usando parámetros diferentes. Por ejemplo, la función DATEDIFF en SQL Server permite indicar la unidad de tiempo de la respuesta, que recibe en el primer parámetro. En cambio MySQL solo recibe como parámetros las fechas a comparar, y el resultado es devuelto siempre en días.

Descripción	SQL Server	MySQL
Suma	DATEADD	DATE_ADD
Resta	DATEDIFF (permite incluir la unidad de tiempo de la respuesta)	DATEDIFF (la respuesta es en días)
Convertir fecha a cadena	DATENAME	DATE_FORMAT
Convertir fecha a número	DATEPART	EXTRACT
Obtener fecha y hora actual	GETDATE	NOW

EJEMPLO

-- Obtener la edad aproximada²⁹ a la que fueron contratados los empleados
 SQL Server

```
SELECT LastName, BirthDate, HireDate,
DATEDIFF(day, BirthDate, HireDate) / 365 AS
EdadAlAlta
FROM Employees
ORDER BY EdadAlAlta;
```

MySQL

```
SELECT LastName, BirthDate, HireDate,
DATEDIFF(HireDate, BirthDate) DIV 365 AS
EdadAlAlta
FROM Employees
ORDER BY EdadAlAlta;
```

LastName	BirthDate	HireDate	EdadAlAlta
Dodsworth	1966-01-27 00:00:00.000	1994-11-15 00:00:00.000	28
Leverling	1963-08-30 00:00:00.000	1992-04-01 00:00:00.000	29
Suyama	1963-07-02 00:00:00.000	1993-10-17 00:00:00.000	30
King	1960-05-29 00:00:00.000	1994-01-02 00:00:00.000	34
Callahan	1958-01-09 00:00:00.000	1994-03-05 00:00:00.000	36
Buchanan	1955-03-04 00:00:00.000	1993-10-17 00:00:00.000	38
Fuller	1952-02-19 00:00:00.000	1992-08-14 00:00:00.000	40
Davolio	1948-12-08 00:00:00.000	1992-05-01 00:00:00.000	44
Peacock	1937-09-19 00:00:00.000	1993-05-03 00:00:00.000	56

²⁹ La edad es aproximada porque la estamos calculando en días y dividiendo por 365, sin contemplar años bisiestos



-- Hallar el mes de nacimiento de cada empleado y ordenar por mes cronológicamente
 SQL Server

```
SELECT FirstName, LastName, DATENAME(month,
BirthDate) AS MesNacimiento
FROM Employees
ORDER BY MONTH(BirthDate);
```

```
SELECT FirstName, LastName,
DATE_FORMAT(BirthDate, '%M') AS
MesNacimiento
FROM Employees
ORDER BY MONTH(BirthDate);
```

FirstName	LastName	MesNacimiento
Laura	Callahan	January
Anne	Dodsworth	January
Andrew	Fuller	February
Steven	Buchanan	March
Robert	King	May
Michael	Suyama	July
Janet	Leverling	August
Margaret	Peacock	September
Nancy	Davolio	December



SUBCONSULTAS

Las subconsultas son consultas embebidas dentro de otras consultas. Se utilizan para obtener información de una tabla basada en información de otra tabla. Por lo general las tablas deben tener algún tipo de relación entre ellas. Su uso más simple es en el WHERE para comparar claves foráneas con los valores de clave primaria a los que hacen referencia, o viceversa. Pero existen tipos de subconsultas que pueden ser usados en otros comandos, como en el HAVING, SELECT o FROM.

Por ejemplo, en la base Northwind, la tabla Orders tiene el campo CustomerID, que hace referencia a la tabla Customers (Clave foránea).

Obtener el clienteid de una orden específica es muy sencillo.

```
/* Hallar el ID de cliente (CustomerID) de la compañía que realizó la orden 10290. */
```

```
SELECT CustomerID  
FROM Orders  
WHERE OrderID = 10290;
```

```
CustomerID  
-----  
COMM
```

COMM es probable que no signifique mucho para quien pueda estar leyendo el resultado del reporte. Mediante la siguiente consulta, que utiliza subconsulta podemos ver un resultado más útil.

```
/* Hallar el Nombre de la compañía (CompanyName) que realizó la orden 10290. */
```

```
SELECT CompanyName  
FROM Customers  
WHERE CustomerID = (SELECT CustomerID  
                      FROM Orders  
                      WHERE OrderID = 10290);
```

CompanyName

Comércio Mineiro

La subconsulta puede contener cualquier SELECT válido, pero debe retornar en este caso una única columna con el número esperado de resultados.

Si la subconsulta devuelve un solo valor se puede comparar por igualdad, desigualdad, mayor, menor, etc.

Pero si la subconsulta devuelve más de un registro, la consulta deberá preguntar si el campo está dentro (IN) o no (NOT IN) del conjunto de valores devueltos.

```
-- Hallar las compañías que hicieron ordenes en 1997  
SELECT CompanyName  
FROM Customers  
WHERE CustomerID IN (SELECT CustomerID  
                      FROM Orders  
                      WHERE YEAR(OrderDate)=1997);
```



CompanyName

Alfreds Futterkiste
Ana Trujillo Emparedados y helados
Antonio Moreno Taquería
Around the Horn
Berglunds snabbköp
Blauer See Delikatessen
...
White Clover Markets
Wilman Kala
Wolski Zajazd

CLASIFICACIÓN DE LAS SUBCONSULTAS

De acuerdo a la cantidad esperada de valores que puede devolver una subconsulta, se las puede clasificar en:

- Escalar
- Multivaluada
- Expresión de Tabla

De acuerdo a la dependencia de la consulta principal, se las puede clasificar en:

- Autocontenido
- Correlacionada

SUBCONSULTAS AUTOCONTENIDAS

Las subconsultas autocontenidoas son independientes de la consulta principal a la que pertenecen, es decir, pueden ser ejecutadas de manera independiente. Por lo tanto, son simples para probar, ya que pueden probarse por separado

SUBCONSULTAS AUTOCONTENIDAS ESCALARES

Dado que devuelve un único valor y es independiente de la consulta principal, puede aparecer en cualquier lugar de la consulta principal en que se necesite como en el SELECT o WHERE.

Por ejemplo, si necesitamos obtener los datos de la orden con el mayor id (ordenid) podemos ejecutar la siguiente consulta:

```
SELECT OrderID, OrderDate, EmployeeID, CustomerID
FROM Orders
WHERE OrderID = (SELECT MAX(OrderID)
                  FROM Orders);
```

OrderID	OrderDate	EmployeeID	CustomerID
11077	1998-05-06 00:00:00.000	1	RATTC

Para que una subconsulta escalar sea válida debe retornar a lo sumo un valor. Si llega a devolver más de un valor se producirá un error en tiempo de ejecución.

La siguiente consulta se ejecuta sin problema:



```

SELECT OrderID
FROM Orders
WHERE EmployeeID = ( SELECT EmployeeID
                      FROM Employees
                      WHERE LastName LIKE 'B%' );
OrderID
-----
10248
10254
10269
10297
10320
10333
10358
...
10899
10922
10954
11043

```

Devuelve las órdenes de los empleados cuyo apellido comienza con B. Como hay un solo empleado cuyo apellido comienza con B, entonces no da error, pero esta consulta podría devolver más de un valor. Esto pasa si en lugar de pedir que empiecen con B pedimos que empiecen con D:

```

SELECT OrderID
FROM Orders
WHERE EmployeeID = ( SELECT EmployeeID
                      FROM Employees
                      WHERE LastName LIKE 'D%' );

```

SQL Server

```

OrderID
-----
Msg 512, Level 16, State 1, Line 1
Subquery returned more than 1 value. This
is not permitted when the subquery follows
=, !=, <, <= , >, >= or when the subquery
is used as an expression.

```

MySQL

```

SQL Error [1242] [21000]: Subquery
returns more than 1 row

```

Si una subconsulta escalar no devuelve ningún valor, devuelve NULL, por lo tanto, el predicado evalúa una comparación con NULL, que da como resultado Desconocido y por lo tanto la consulta principal no devuelve resultados.

Por ejemplo, si buscamos empleado cuyo apellido empieza con A:

```

SELECT OrderID
FROM Orders
WHERE EmployeeID = ( SELECT EmployeeID
                      FROM Employees
                      WHERE LastName LIKE 'A%' );

```

OrderID



SUBCONSULTAS AUTOCONTENIDAS MULTIVALUADAS

Son consultas que devuelven múltiples valores para una misma columna. Los resultados de estas subconsultas deben evaluarse con predicados como IN.

El formato de un predicado utilizando IN es:

<expresión escalar> [NOT] IN (<subconsulta multivaluada>)

El predicado devuelve verdadero si la expresión escalar coincide con alguno de los valores devueltos por la subconsulta.

Si reescribimos uno de los ejemplos anteriores utilizando IN, ya no tendríamos problemas en tiempo de ejecución:

```
SELECT OrderID  
FROM Orders  
WHERE EmployeeID IN ( SELECT EmployeeID  
                      FROM Employees  
                      WHERE LastName LIKE 'D%');
```

```
OrderID  
-----  
10258  
10270  
10275  
10285  
10292  
10293  
10304  
10306  
...  
11016  
11017  
11022  
11058
```

SUBCONSULTAS CORRELACIONADAS

Las subconsultas correlacionadas son subconsultas en las que se hace referencia a atributos que forman parte de la consulta principal. Esto significa que la subconsulta es dependiente de la consulta principal y no puede ser ejecutada de manera independiente. Lógicamente, es como si la subconsulta es evaluada por cada fila de la consulta principal.

Por ejemplo, la siguiente consulta que devuelve las órdenes con el máximo número para cada cliente (CustomerID):

```
SELECT CustomerID, OrderID, OrderDate, EmployeeID  
FROM Orders AS O1  
WHERE OrderID =  
(SELECT MAX(O2.OrderID)  
FROM Orders AS O2  
WHERE O2.CustomerID = O1.CustomerID);
```

La consulta principal se realiza sobre una instancia de la tabla Orders que llamamos O1, que devuelve los valores para los que el campo OrderID coinciden con el resultado de la subconsulta.

La subconsulta filtra los resultados de una segunda instancia de la tabla Orders que llamamos O2, en donde el CustomerID de la tabla de la subconsulta es igual al CustomerID de la tabla principal (O1).



CustomerID	OrderID	OrderDate	EmployeeID
WOLZA	11044	1998-04-23 00:00:00.000	4
WILMK	11005	1998-04-07 00:00:00.000	2
WHITC	11066	1998-05-01 00:00:00.000	7
WELLI	10935	1998-03-09 00:00:00.000	4
...			
AROUT	11016	1998-04-10 00:00:00.000	9
ANTON	10856	1998-01-28 00:00:00.000	3
ANATR	10926	1998-03-04 00:00:00.000	4
ALFKI	11011	1998-04-09 00:00:00.000	3

EXISTS

También contamos con el predicado EXISTS, que devuelve verdadero (true) en caso de que la subconsulta devuelva alguna fila, de lo contrario devuelve falso (false).

El formato de un predicado utilizando EXISTS es:

[NOT] EXISTS (<subconsulta multivaluada>)

Por ejemplo, una consulta para obtener los clientes de España (Spain) que han realizado órdenes:

```
SELECT CustomerID, CompanyName
FROM Customers AS C
WHERE Country = 'Spain'
AND EXISTS
(SELECT *
FROM Orders AS O
WHERE O.CustomerID = C.CustomerID);
```

CustomerID CompanyName

BOLID	Bólido Comidas preparadas
GALED	Galería del gastrónomo
GODOS	Godos Cocina Típica
ROMEY	Romero y tomillo



JOINS (JUNTAR)

JOIN es, por lejos, el tipo de operador de tabla más utilizado y una de las características más utilizadas en SQL en general. La mayoría de las consultas implican la combinación de datos de varias tablas y una de las principales herramientas que se utilizan para este propósito es el JOIN.

Esta unidad cubre los tipos de JOIN fundamentales (CROSS, INNER y OUTER), auto joins, equi y non-equi.

CROSS JOIN

Entre los tres tipos de join fundamentales (cross, inner y outer), el primero es el más simple, aunque se usa con menos frecuencia que los demás. Un cross join produce un producto cartesiano de las dos tablas de entrada. Si una tabla contiene M filas y la otra N filas, obtiene un resultado con $M \times N$ filas.

En términos de sintaxis, SQL estándar admite dos sintaxis para cross join. Una es una sintaxis más antigua, en la que especifica una coma entre los nombres de las tablas, así:

```
SELECT E1.FirstName AS nombre1, E1.LastName AS apellido1,  
E2.FirstName AS nombre2, E2.LastName AS apellido2  
FROM Employees AS E1, Employees AS E2;
```

Otra es una sintaxis más nueva que se agregó en el estándar SQL-92, en la que especifica una palabra clave que indica el tipo de unión (CROSS, en nuestro caso) seguida de la palabra clave JOIN entre los nombres de las tablas, así:

```
SELECT E1.FirstName AS nombre1, E1.LastName AS apellido1,  
E2.FirstName AS nombre2, E2.LastName AS apellido2  
FROM Employees AS E1 CROSS JOIN Employees AS E2;
```

Ambas sintaxis son estándar y ambas son compatibles con T-SQL y MySQL. No hay una diferencia lógica ni una diferencia de optimización entre los dos. Quizás te preguntes, entonces, ¿por qué SQL estándar se molestó en crear dos sintaxis diferentes si tienen el mismo significado? La razón de esto no tiene nada que ver con los cross e inner joins; más bien, está relacionado con los outer joins. Un outer join implica un predicado coincidente, que tiene un papel muy diferente al de un predicado de filtrado. El primero define solo qué filas del lado no preservado de la combinación coinciden con las filas del lado preservado, pero no puede filtrar las filas del lado preservado. Este último se utiliza como filtro básico y ciertamente puede descartar filas de ambos lados.

El comité de estándares SQL decidió agregar soporte para outer joins en el estándar SQL-92, pero pensaron que la sintaxis tradicional basada en comas no se presta para admitir el nuevo tipo de join. Esto se debe a que esta sintaxis no le permite definir un predicado coincidente como parte del operador de tabla de join, que, como se mencionó, se supone que juega un papel diferente al predicado de filtrado que especifica en la cláusula WHERE. Así que terminaron creando la sintaxis más nueva con la palabra clave JOIN con una cláusula ON designada donde uno indica el predicado coincidente. Esta sintaxis más nueva con la palabra clave JOIN se conoce comúnmente como sintaxis SQL-92, y la sintaxis antigua basada en comas se conoce como sintaxis SQL-89.

Para permitir un estilo de codificación consistente para los diferentes tipos de combinaciones, el estándar SQL-92 también agregó soporte para una sintaxis similar basada en palabras clave JOIN para cross e inner joins. Esa es la historia que explica por qué ahora tenemos dos sintaxis estándar para cross e inner joins y sólo una sintaxis estándar para outer joins. Se recomienda encarecidamente ceñirse a utilizar la sintaxis basada en palabras clave JOIN en todos los ámbitos. Una razón es que da como resultado un estilo de codificación consistente que es mucho más fácil de



mantener que un estilo mixto. Además, la sintaxis basada en comas es más propensa a errores, como veremos más adelante cuando analicemos los inner joins.

SINTAXIS

SQL-92

```
SELECT tabla1.columna1, tabla2.columna2
FROM tabla1 CROSS JOIN tabla2
[WHERE condiciones]
```

SQL-89

```
SELECT tabla1.columna1, tabla2.columna2
FROM tabla1, tabla2
[WHERE condiciones]
```

INNER JOIN

Desde una perspectiva de procesamiento lógico de consultas, un inner join implica dos pasos. Comienza con un producto cartesiano entre las dos tablas de entrada, como en un cross join. Luego aplica un filtro que generalmente involucra elementos coincidentes de ambos lados.

Desde una perspectiva de procesamiento físico de consultas, por supuesto, las cosas se pueden hacer de manera diferente, siempre que el resultado sea correcto.

Al igual que con los cross joins, los inner joins actualmente tienen dos sintaxis estándar.

SINTAXIS

SQL-92

```
SELECT tabla1.columna1, tabla2.columna2
FROM tabla1 [INNER] JOIN tabla2
    ON (tabla1.columna1=tabla2.columna1)
[WHERE condiciones]
```

SQL-89

```
SELECT tabla1.columna1, tabla2.columna2
FROM tabla1, tabla2
WHERE tabla1.columna1=tabla2.columna1
```

Una es la sintaxis SQL-89, donde especifica comas entre los nombres de las tablas y luego aplica todos los predicados de filtrado en la cláusula WHERE, así:

```
SELECT C.CustomerID, C.CompanyName, O.OrderID
FROM Customers AS C, Orders AS O
WHERE C.CustomerID = O.CustomerID
AND C.Country = 'USA';
```

Como podemos ver, la sintaxis no distingue realmente entre un cross join y un inner join; más bien, expresa un inner join como un cross join con una condición de filtro.

La otra sintaxis es SQL-92, que es más compatible con la sintaxis de cross join estándar. Especificamos las palabras clave INNER JOIN entre los nombres de las tablas (o simplemente JOIN porque INNER es el predeterminado) y el predicado de join en la cláusula obligatoria ON. Si tenemos condiciones de filtro adicionales, podemos especificarlas en la cláusula ON del join o en la cláusula WHERE habitual, así:

```
SELECT C.CustomerID, C.CompanyName, O.OrderID
FROM Customers AS C INNER JOIN Orders AS O
ON C.CustomerID = O.CustomerID
WHERE C.Country = 'USA';
```



A pesar de que la cláusula ON es obligatoria en la sintaxis SQL-92 para inner joins, no hay distinción entre un predicado coincidente y un predicado de filtrado como en los outer joins. Ambos predicados se tratan como predicados de filtrado. Entonces, aunque desde una perspectiva de procesamiento lógico de consultas, se supone que la cláusula WHERE se evalúa después de la cláusula FROM con todos sus joins, el optimizador podría muy bien decidir procesar los predicados de la cláusula WHERE antes de comenzar a procesar los joins en la cláusula FROM. De hecho, si examinamos los planes para las dos consultas que se acaban de mostrar, notaremos que en ambos casos el plan comienza con una exploración del índice agrupado en la tabla Clientes y aplica el filtro de país como parte de la exploración.

Mencionamos anteriormente que se recomienda ceñirse a la sintaxis SQL-92 para los joins y evitar la sintaxis SQL-89 aunque es estándar para los cross e inner joins. Ya proporcionamos una razón para esta recomendación: usar un estilo consistente para todo tipo de joins. Otra razón es que la sintaxis SQL-89 es más propensa a errores; específicamente, es más probable que se olvide un predicado de join y que tal error pase desapercibido. Con la sintaxis SQL-92, generalmente especificamos cada predicado de join justo después del respectivo join, así:

```
FROM T1
INNER JOIN T2
ON T2.col1 = T1.col1
INNER JOIN T3
ON T3.col2 = T2.col2
INNER JOIN T4
ON T4.col3 = T3.col3
```

Con esta sintaxis, la probabilidad de que pierda un predicado de join es baja, e incluso si lo hace, el analizador generará un error porque la cláusula ON es obligatoria en los inner joins (y outer). Por el contrario, con la sintaxis SQL-89, especificamos todos los nombres de tabla separados por comas en la cláusula FROM y todos los predicados como una conjunción en la cláusula WHERE, así:

```
FROM T1, T2, T3, T4
WHERE T2.col1 = T1.col1 AND T3.col2 = T2.col2 AND T4.col3 = T3.col3
```

Claramente, la probabilidad de que olvidemos un predicado por error es mayor, y si lo hacemos, terminaremos con un cross join involuntario.

Una pregunta común es si existe una diferencia entre usar las palabras clave INNER JOIN y simplemente JOIN. No hay ninguna. SQL estándar hizo que los inner join sean los predeterminados porque probablemente son el tipo de join más utilizado. De manera similar, SQL estándar hizo que la palabra clave OUTER fuera opcional en los outer joins. Por ejemplo, LEFT OUTER JOIN y LEFT JOIN son equivalentes. En cuanto a lo que se recomienda usar, algunos prefieren la sintaxis completa y la encuentran más clara. Otros prefieren la sintaxis breve porque da como resultado un código más corto. Creo que lo que es más importante que utilizar la sintaxis completa o breve es ser coherente entre los miembros del equipo de desarrollo. Cuando diferentes personas usan diferentes estilos, puede resultar difícil mantener el código de los demás. Una buena idea es tener reuniones para discutir opciones de estilo, como sintaxis completa o breve, sangría, mayúsculas, etc., y elaborar un documento de estilo de codificación que todos deban seguir. A veces, las personas tienen opiniones sólidas sobre ciertos aspectos de estilo en función de sus preferencias personales, por lo que cuando surgen conflictos, se puede votar para determinar qué opción prevalecerá. Una vez que todos comienzan a usar un estilo coherente, es mucho más fácil para diferentes personas mantener el mismo código.



-- Crear un reporte que muestre las ordenes de un empleado.

```
SELECT Employees.EmployeeID, Employees.FirstName, Employees.LastName,
Orders.OrderID, Orders.OrderDate
FROM Employees JOIN Orders ON (Employees.EmployeeID = Orders.EmployeeID)
ORDER BY Orders.OrderDate;
```

EmployeeID	FirstName	LastName	OrderID	OrderDate
5	Steven	Buchanan	10248	1996-07-04 00:00:00.000
6	Michael	Suyama	10249	1996-07-05 00:00:00.000
4	Margaret	Peacock	10250	1996-07-08 00:00:00.000
3	Janet	Leverling	10251	1996-07-08 00:00:00.000
4	Margaret	Peacock	10252	1996-07-09 00:00:00.000
3	Janet	Leverling	10253	1996-07-10 00:00:00.000
...				
8	Laura	Callahan	11075	1998-05-06 00:00:00.000
4	Margaret	Peacock	11076	1998-05-06 00:00:00.000
1	Nancy	Davolio	11077	1998-05-06 00:00:00.000

OUTER JOIN

Desde una perspectiva de procesamiento lógico de consultas, los outer joins comienzan con los mismos dos pasos que los inner joins. Pero, además, tienen un tercer paso lógico que garantiza que se devuelvan todas las filas de la tabla o tablas que marque como conservadas, incluso si no tienen coincidencias en la otra tabla según el predicado de join. Con las palabras clave LEFT (o LEFT OUTER), RIGHT o FULL, marca la tabla de la izquierda, la tabla de la derecha o ambas tablas como conservadas. MySQL no soporta el FULL OUTER JOIN, pero puede ser emulado realizando un LEFT JOIN y un RIGHT JOIN y aplicando una operación de UNION entre ambos resultados. Las filas de salida que representan no coincidencias (también conocidas como filas externas) tienen NULL utilizados como marcadores de posición en las columnas del lado no preservado.

SINTAXIS

SQL Server

```
SELECT tabla1.columna1, tabla2.columna2
FROM tabla1 {LEFT|RIGHT|FULL} [OUTER]
JOIN tabla2
    ON (tabla1.columna1=tabla2.columna1)
[WHERE condiciones]
```

MySQL

```
SELECT tabla1.columna1, tabla2.columna2
FROM tabla1 {LEFT|RIGHT} [OUTER] JOIN
tabla2
    ON (tabla1.columna1=tabla2.columna1)
[WHERE condiciones]
```

Como ejemplo, la siguiente consulta devuelve clientes y sus pedidos, incluidos los clientes sin pedidos:

```
SELECT C.CustomerID, C.CompanyName, C.Country,
O.OrderID, O.ShipCountry
FROM Customers AS C
LEFT OUTER JOIN Orders AS O
ON C.CustomerID = O.CustomerID;
```

En los outer joins, el papel del predicado en la cláusula ON es diferente que en los inner joins. Con el último, sirve como una herramienta de filtrado simple, mientras que con el primero sirve como una herramienta de comparación más sofisticada. Todas las filas del lado preservado se devolverán tanto si encuentran filas coincidentes según el predicado como si no. Si necesitamos que se apliquen filtros simples, los especificamos en la cláusula WHERE. Por ejemplo, una forma clásica de identificar solo las no coincidencias es usar un outer join y agregar una cláusula WHERE que filtra solo las filas que tienen un NULL en una columna del lado no preservado que no permite NULL normalmente. La columna de



clave principal es una buena opción para este propósito. Por ejemplo, la siguiente consulta solo devuelve clientes que no realizaron pedidos:

```
SELECT C.CustomerID, C.CompanyName, O.OrderID
FROM Customers AS C
LEFT OUTER JOIN Orders AS O
ON C.CustomerID = O.CustomerID
WHERE O.OrderID IS NULL;
```

Es mucho más común que las personas utilicen left outer joins que right outer joins. Podría deberse a que la mayoría de las personas tienden a especificar tablas referenciadas antes de hacer referencia a las de la consulta y, por lo general, la tabla referenciada es la que puede tener filas sin coincidencias en la referencia. Es cierto que, en un caso de JOIN simple, como T1 LEFT OUTER JOIN T2, podemos expresar lo mismo con un T2 RIGHT OUTER JOIN T1 simétrico. Sin embargo, en ciertos casos más complejos con múltiples joins involucrados, encontraremos que usar un right outer join permite una solución más simple que usar left outer joins.

EJEMPLO

```
/* Crear un reporte que muestre el número de empleados y clientes de cada ciudad que tenga empleados en ella */
```

```
SELECT COUNT(DISTINCT e.EmployeeID) AS numEmpleados,
COUNT(DISTINCT c.CustomerID) AS numEmpresas,
e.City, c.City
FROM Employees e LEFT JOIN Customers c ON (e.City = c.City)
GROUP BY e.City, c.City
ORDER BY numEmpleados DESC;
```

numEmpleados	numEmpresas	City	City
4	6	London	London
2	1	Seattle	Seattle
1	0	Tacoma	NULL
1	0	Redmond	NULL
1	1	Kirkland	Kirkland

Warning: Null value is eliminated by an aggregate or other SET operation.³⁰

Todos los registros de la tabla Empleados son contabilizados, aunque no haya registros con la misma ciudad en la tabla Clientes.

```
/* Crear un reporte que muestre el número de empleados y clientes de cada ciudad que tenga clientes en ella */
```

```
SELECT COUNT(DISTINCT e.EmployeeID) AS numEmpleados,
COUNT(DISTINCT c.CustomerID) AS numEmpresas,
e.City, c.City
FROM Employees e RIGHT JOIN Customers c ON (e.City = c.City)
GROUP BY e.City, c.City
ORDER BY numEmpleados DESC;
```

³⁰ Advertencia: valor NULL eliminado por el agregado u otra operación SET.



numEmpleados	numEmpresas	City	City
4	6	London	London
2	1	Seattle	Seattle
1	1	Kirkland	Kirkland
0	1	NULL	Kobenhavn
0	1	NULL	Köln
...			
0	1	NULL	Graz
0	1	NULL	Helsinki
0	1	NULL	I. de Margarita

Warning: Null value is eliminated by an aggregate or other SET operation.

Todos los registros de la tabla Clientes son contabilizados incluso cuando no tengan ciudades relacionadas en la tabla Empleados.

/ Crear un reporte que muestre el número de empleados y clientes de cada ciudad */*

```
SELECT COUNT(DISTINCT e.EmployeeID) AS numEmpleados,
COUNT(DISTINCT c.CustomerID) AS numEmpresas,
e.City, c.City
FROM Employees e FULL JOIN Customers c ON (e.City = c.City)
GROUP BY e.City, c.City
ORDER BY numEmpleados DESC;
```

numEmpleados	numEmpresas	City	City
4	6	London	London
2	1	Seattle	Seattle
1	1	Kirkland	Kirkland
1	0	Redmond	NULL
1	0	Tacoma	NULL
0	1	NULL	Aachen
...			
0	1	NULL	Versailles
0	1	NULL	Walla Walla
0	1	NULL	Warszawa

Warning: Null value is eliminated by an aggregate or other SET operation.

SELF JOIN

Un self join (auto join) es un join entre varias instancias de la misma tabla. Como ejemplo, la siguiente consulta relaciona a los empleados con sus gerentes, que también son empleados:

```
SELECT CONCAT(E.FirstName, ' ', E.LastName) AS emp,
CONCAT(J.FirstName, ' ', J.LastName) AS jefe
FROM Employees AS E
LEFT OUTER JOIN Employees AS J
ON E.ReportsTo = J.EmployeeID;
```

Lo que tiene de especial los self join es que es obligatorio asignar un alias a las instancias de la tabla de forma diferente; de lo contrario, terminaremos con nombres de columna duplicados, incluidos los prefijos de la tabla.



EQUI Y NONEQUI JOINS

Los términos equi y nonequi joins se refieren al tipo de operador que utiliza en el predicado de combinación. Cuando utilizamos un operador de igualdad, como en la gran mayoría de las consultas, la combinación se denomina equi join. Cuando utiliza un operador que no sea la igualdad, el join se denomina nonequi join.

Como ejemplo, la siguiente consulta usa un nonequi join con un operador menor que (<) para identificar pares únicos de empleados:

```
SELECT E1.EmployeeID, E1.FirstName, E1.LastName, E2.EmployeeID, E2.LastName, E2.FirstName
FROM Employees AS E1
INNER JOIN Employees AS E2
ON E1.EmployeeID < E2.EmployeeID;
```

Si te preguntas por qué deberíamos usar un nonequi join aquí en lugar de un cross join, es porque no deseamos obtener los “auto” pares (el mismo empleado x, x en ambos lados) y no deseamos obtener pares “espejo” (x, y así como y, x). Si deseamos producir triples únicos, simplemente agregamos otro nonequi join a una tercera instancia de la tabla (llamémosla E3), con el predicado de combinación E2.empid < E3.empid.

CONSULTAS MULTI-JOIN

Las consultas multi-join son consultas con múltiples joins (valga la redundancia). Hay una serie de consideraciones interesantes sobre estas consultas, algunas relacionadas con la optimización y otras relacionadas con el tratamiento lógico.

SINTAXIS

```
SELECT tabla1.columna, tabla2.columna, table3.columna
FROM tabla1
    [INNER] JOIN tabla2 ON (tabla1.columna=tabla2.columna)
    [INNER] JOIN tabla3 ON (tabla2.columna=tabla3.columna)
WHERE condiciones
```

Como ejemplo, la siguiente consulta une cinco tablas para devolver pares de cliente-proveedor que tuvieron actividad juntos:

```
SELECT DISTINCT C.CompanyName AS cliente, s.CompanyName AS proveedor
FROM Customers AS C
INNER JOIN Orders AS O
ON O.CustomerID = C.CustomerID
INNER JOIN [Order Details] AS DO
ON DO.OrderID = O.OrderID
INNER JOIN Products AS P
ON P.ProductID = DO.ProductID
INNER JOIN Suppliers AS S
ON S.SupplierID = P.SupplierID;
```

El nombre de la empresa del cliente se obtiene de la tabla Clientes (Customers). La tabla Clientes (Customers) se une a la tabla Ordenes (Orders) para encontrar las cabeceras de las órdenes que realizaron los clientes. El resultado se une con Detalles Orden (Order Details) para encontrar las líneas de pedido dentro de esas órdenes. Las líneas de pedido contienen los ID de los productos que se pidieron. El resultado se une a Productos (Products), donde se encuentran los ID de los proveedores de esos productos. Finalmente, el resultado se une con Proveedores (Suppliers) para obtener el nombre de la empresa proveedora.



Dado que el mismo par cliente-proveedor puede aparecer más de una vez en el resultado, la consulta tiene una cláusula DISTINCT para eliminar los duplicados. Aquí había una buena razón para usar DISTINCT. Sin embargo, tengan en cuenta que si observan un uso excesivo de DISTINCT en las consultas de join de alguien, podría indicar que no comprenden correctamente las relaciones en el modelo de datos. Al combinar tablas basándose en relaciones incorrectas, podemos terminar con duplicados, y el uso de DISTINCT en tal caso podría ser un intento de ocultarlos.

EJEMPLO

```
/* Crear un reporte que muestre OrderID, y nombre de la empresa que realizó la orden, y el nombre y apellido del empleado que la ingresó. Sólo mostrar órdenes efectuadas después del 1 de Enero de 1998 que fueron despachadas luego de la fecha requerida. Ordenar por Nombre de empresa */
```

```
SELECT o.OrderID, c.CompanyName, e.FirstName, e.LastName
FROM Orders o JOIN Employees e ON (e.EmployeeID = o.EmployeeID)
JOIN Customers c ON (c.CustomerID = o.CustomerID)
WHERE o.ShippedDate > o.RequiredDate AND o.OrderDate > '19980101'
ORDER BY c.CompanyName;
```

OrderID	CompanyName	FirstName	LastName
10924	Berglunds snabbköp	Janet	Leverling
10970	Bólido Comidas preparadas	Anne	Dodsworth
10827	Bon app'	Nancy	Davolio
10816	Great Lakes Food Market	Margaret	Peacock
10960	HILARION-Abastos	Janet	Leverling
10927	La corne d'abondance	Margaret	Peacock
10828	Rancho grande	Anne	Dodsworth
10847	Save-a-lot Markets	Margaret	Peacock

NATURAL JOIN

En Álgebra Relacional tenemos la operación que nos permite realizar un JOIN NATURAL. Este JOIN tiene características particulares. La primera es que no se especifica condición, pero que no especifiquemos condición no significa que no se aplique ninguna condición de filtro. Se aplica una condición de igualdad entre aquellos atributos que tengan el mismo nombre y tipo de datos.

SQL Server no tiene implementado el NATURAL JOIN, pero MySQL sí, con la siguiente sintaxis:

```
SELECT *
FROM tabla1 NATURAL JOIN tabla2;
```

Al igual que en el Álgebra Relacional, las columnas redundantes no forman parte del resultado.

Es importante tener en cuenta que el NATURAL JOIN puede tener algunas limitaciones, como una posible pérdida de control sobre la condición de JOIN y posibles resultados no deseados si los nombres de las columnas tienen significados ambiguos. Se recomienda revisar cuidadosamente y comprender los nombres de las columnas y los tipos de datos en las tablas antes de usar NATURAL JOIN para garantizar resultados precisos.

Esta manera de realizar JOIN no es parte del estándar; por esto y lo antedicho, no se recomienda su uso.



OPERADORES UNION, EXCEPT E INTERSECT

Los operadores UNION, EXCEPT e INTERSECT son operadores relacionales que combinan filas de los resultados de dos consultas. La forma general de utilizarlos es:

```
<consulta 1>
<operador>
<consulta 2>
[ORDER BY <lista_de_ordenamiento>];
```

El operador UNION unifica las filas de las dos entradas. El operador INTERSECT devuelve solo las filas que son comunes a ambas entradas. El operador EXCEPT devuelve las filas que aparecen en la primera entrada, pero no en la segunda.

Las consultas de entrada no pueden tener una cláusula ORDER BY porque se supone que deben devolver un resultado relacional. Sin embargo, se permite una cláusula ORDER BY contra el resultado del operador.

Los esquemas de las consultas de entrada deben ser compatibles. El número de columnas tiene que ser el mismo, y los tipos deben ser implícitamente convertibles de la que tiene la precedencia de tipo de datos más baja a la más alta. Los nombres de las columnas de resultados se definen mediante la primera consulta.

Al comparar filas entre las entradas, estos operadores utilizan implícitamente el predicado estándar DISTINCT. Según este predicado, un NULL no es distinto de otro NULL y un NULL es distinto de un valor no NULL. Por el contrario, según el operador de igualdad, comparar dos NULL da como resultado el valor lógico desconocido, y lo mismo se aplica al comparar un valor NULL con un valor no NULL. Este hecho hace que estos operadores sean sencillos e intuitivos de usar, incluso cuando son posibles NULL en los datos.

Cuando se utilizan varios operadores en una consulta sin paréntesis, INTERSECT precede a UNION y EXCEPT. Los dos últimos tienen la misma precedencia, por lo que se evalúan según el orden de aparición. Siempre podemos forzar el orden de evaluación deseado utilizando paréntesis.

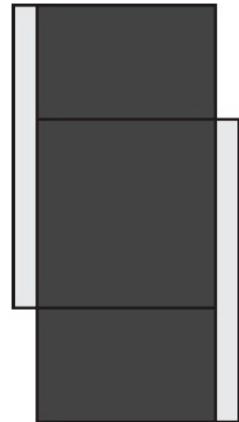
Las siguientes secciones proporcionan más detalles sobre estos operadores.



EL OPERADOR UNION

El operador UNION consolida los resultados de las dos consultas de entrada. Si una fila aparece en alguno de los conjuntos de entrada, aparecerá en el resultado de la operación UNION. T-SQL implementa el operador UNION ALL y UNION (con la opción DISTINCT implícita). En MySQL funciona de la misma manera, pero tiene la opción de agregar DISTINCT de manera explícita.

La siguiente figura ilustra al operador UNION. El área sombreada representa el resultado del operador. Las áreas no sombreadas representan el hecho de que el operador puede no incluir todos los atributos de las relaciones originales.



EL OPERADOR UNION ALL

El operador UNION ALL consolida los resultados de las dos consultas sin intentar remover duplicados del resultado final. Si la consulta 1 devuelve m filas y la consulta 2 devuelve n filas, el resultado de UNION ALL tendrá m + n filas.

Por ejemplo, en la siguiente consulta utilizaremos UNION ALL para consolidar las ubicaciones de empleados y clientes.

```
SELECT Country, Region, City FROM Employees
```

```
UNION ALL
```

```
SELECT Country, Region, City FROM Customers;
```

El resultado tiene 100 filas – 9 de empleados y 91 de clientes –

Country	Region	City
USA	WA	Seattle
USA	WA	Tacoma
USA	WA	Kirkland
USA	WA	Redmond
UK	NULL	London
UK	NULL	London
UK	NULL	London
USA	WA	Seattle
UK	NULL	London
...		
USA	WA	Kirkland
Denmark	NULL	Århus
France	NULL	Lyon
France	NULL	Reims
Germany	NULL	Stuttgart
Finland	NULL	Oulu
Brazil	SP	Resende
USA	WA	Seattle
Finland	NULL	Helsinki
Poland	NULL	Warszawa

Como UNION ALL no elimina los duplicados, el resultado es un multiset, es decir, que una misma fila puede aparecer varias veces, como de hecho podemos ver en el ejemplo.



EL OPERADOR UNION (DISTINCT)

El operador UNION (con la opción DISTINCT implícita en SQL Server y MySQL, o explícita en MySQL) consolida los resultados de las dos consultas de entrada y elimina los duplicados. Si una fila aparece en las dos consultas, sólo aparecerá una vez en el resultado final. En otras palabras, el resultado es un conjunto, y no un multiset.

Veamos el mismo ejemplo de antes, pero aplicando UNION:

```
SELECT Country, Region, City FROM Employees
UNION
SELECT Country, Region, City FROM Customers;
```

Esta vez obtenemos 71 filas en lugar de 100.

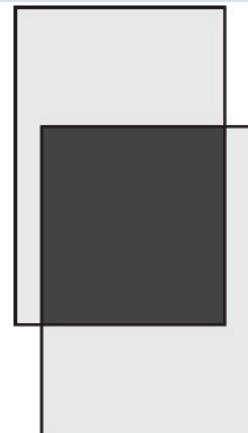
Country	Region	City
Argentina	NULL	Buenos Aires
Austria	NULL	Graz
Austria	NULL	Salzburg
Belgium	NULL	Bruxelles
Belgium	NULL	Charleroi
Brazil	RJ	Rio de Janeiro
Brazil	SP	Campinas
...		
USA	WA	Walla Walla
USA	WY	Lander
Venezuela	DF	Caracas
Venezuela	Lara	Barquisimeto
Venezuela	Nueva Esparta	I. de Margarita
Venezuela	Táchira	San Cristóbal

EL OPERADOR INTERSECT

El operador INTERSECT devuelve sólo las filas que son comunes al resultado de las consultas de entrada.

En la figura podemos verlo gráficamente.

MySQL incluyó el soporte para esta operación desde la versión 8 (8.0.31 y posteriores).



EL OPERADOR INTERSECT (DISTINCT)

El operador INTERSECT (con la opción DISTINCT implícita en SQL Server y MySQL, o explícita en MySQL) devuelve las filas que aparecen en los resultados de ambas consultas de entrada, eliminando duplicados. Si una fila aparece al menos una vez en cada resultado, formará parte del resultado final. Aunque entiendo que no tendría sentido, MySQL permite utilizar también ALL, al igual que en la UNION.

La siguiente consulta devuelve las ubicaciones en las que coinciden empleados y clientes.

```
SELECT Country, Region, City FROM Employees
INTERSECT
SELECT Country, Region, City FROM Customers;
```



Country	Region	City
UK	NULL	London
USA	WA	Kirkland
USA	WA	Seattle

EL OPERADOR EXCEPT

El operador EXCEPT implementa la diferencia de conjuntos. Opera sobre los resultados de dos consultas, y devuelve las filas que aparecen como resultado de la primera consulta, pero no en el resultado de la segunda consulta.

MySQL incluyó el soporte para esta operación desde la versión 8 (8.0.31 y posteriores).

EL OPERADOR EXCEPT (DISTINCT)

El operador EXCEPT (con la opción DISTINCT implícita en SQL Server y MySQL, o explícita en MySQL) devuelve las filas que aparecen como resultado de la primera consulta, pero no en el resultado de la segunda consulta, eliminando duplicados. En esta operación MySQL también permite el uso de ALL.

Una fila de la primera consulta terminará como parte del resultado si aparece al menos una vez en la primera consulta y no aparece en la segunda.

Noten que en las operaciones de UNION e INTERSECT el orden en que se presenten las consultas no influye en el resultado final, pero en EXCEPT sí.

Por ejemplo, la siguiente consulta devuelve las ubicaciones de empleados en las que no hay clientes.

```
SELECT Country, Region, City FROM Employees
EXCEPT
SELECT Country, Region, City FROM Customers;
```

Country	Region	City
USA	WA	Redmond
USA	WA	Tacoma

Si damos vuelta las consultas, obtenemos las ubicaciones de clientes, en la que no hay empleados, y en ese caso obtendríamos 66.



PRECEDENCIA

SQL define precedencia entre los operadores de conjuntos. El operador INTERSECT precede a los operadores UNION y EXCEPT, y UNION y EXCEPT son evaluados en orden de aparición.

Por ejemplo:

```
SELECT Country, Region, City FROM Suppliers  
EXCEPT  
SELECT Country, Region, City FROM Employees  
INTERSECT  
SELECT Country, Region, City FROM Customers;
```

Devuelve 28 filas. Pero la siguiente consulta devuelve 3:

```
(SELECT Country, Region, City FROM Suppliers  
EXCEPT  
SELECT Country, Region, City FROM Employees)  
INTERSECT  
SELECT Country, Region, City FROM Customers;
```

Country	Region	City
Canada	Québec	Montréal
France	NULL	Paris
Germany	NULL	Berlin

En el primer caso se realiza primero la operación INTERSECT, y luego EXCEPT.

En el segundo caso, mediante el uso de paréntesis, se realiza primero la operación EXCEPT, y luego INTERSECT.

REGLAS DE LOS OPERADORES DE CONJUNTOS

- Cada consulta debe tener el mismo número de columnas
- Las columnas deben estar en el mismo orden
- Los tipos de las columnas deben ser compatibles.



RUTINAS

Las rutinas son objetos programables que encapsulan código para calcular un resultado o ejecutar una actividad. Veremos tres tipos de rutinas: funciones definidas por el usuario, procedimientos almacenados (Stored Procedures) y disparadores (Triggers).

Con SQL Server, puede elegir si deseamos desarrollar una rutina con T-SQL o con código Microsoft .NET basado en la integración de CLR en el producto. En MySQL podemos utilizar SQL o bien otros lenguajes, como C o C++, que generen funciones nativas e incorporarlas al servidor. Nos limitaremos a T-SQL y SQL.

FUNCIONES DEFINIDAS POR EL USUARIO

El propósito de una función definida por el usuario (FDU) es encapsular la lógica que calcula algo, posiblemente en función de los parámetros de entrada, y devolver un resultado.

SQL Server admite FDU escalares y con valores de tabla. MySQL admite FDU escalares. Las FDU escalares devuelven un solo valor; las FDU con valores de tabla devuelven una tabla. Una ventaja de usar FDU es que podemos incorporarlas en las consultas. Las FDU escalares pueden aparecer en cualquier parte de la consulta donde pueda aparecer una expresión que devuelva un único valor (por ejemplo, en la lista SELECT). Las FDU de tabla pueden aparecer en la cláusula FROM de una consulta. Vamos a mostrar y trabajar con FDU escalares.

No se permite que los FDU tengan efectos secundarios. Esto significa que las FDU no pueden aplicar ningún cambio de esquema o datos en la base de datos. Pero otras formas de causar efectos secundarios son menos obvias.

Por ejemplo, invocar la función RAND para devolver un valor aleatorio o la función NEWID para devolver un identificador único global (GUID) tiene efectos secundarios. Cada vez que invoque la función RAND sin especificar una semilla, SQL Server genera una semilla aleatoria que se basa en la invocación anterior de RAND. Por esta razón, SQL Server necesita almacenar información internamente cada vez que invoque la función RAND. De manera similar, siempre que invoque la función NEWID, el sistema necesita reservar cierta información para tenerla en cuenta en la siguiente invocación de NEWID. Debido a que RAND y NEWID tienen efectos secundarios, no podemos usarlas en las FDU.

Por ejemplo, el siguiente código crea una FDU llamada dbo.ObtenerEdad que devuelve la edad de una persona con una fecha de nacimiento específica (argumento (@nacimiento) en una fecha de evento específica (argumento (@evento):

```
CREATE OR ALTER FUNCTION dbo.ObtenerEdad  (
@nacimiento AS DATE,
@evento AS DATE
)
RETURNS INT
AS
BEGIN
    DECLARE @edad INT;
    -- Calcular la edad restando el año de la fecha de nacimiento a la fecha del evento
    SET @edad = YEAR(@evento) - YEAR(@nacimiento)
    -- Ajustar la edad en caso de que la fecha de nacimiento aún no haya ocurrido
    IF MONTH(@nacimiento) > MONTH(@evento) OR (MONTH(@nacimiento) = MONTH(@evento) AND
    DAY(@nacimiento) > DAY(@evento))
        BEGIN
            SET @edad = @edad - 1;
        END
    RETURN @edad;
END;
```



La función calcula la edad como la diferencia, en términos de años, entre el año de nacimiento y el año del evento, menos 1 año en los casos en que el mes y el día del evento son menores que el mes y el día del nacimiento. La expresión $100 * \text{mes} + \text{día}$ es simplemente un truco para concatenar el mes y el día. Por ejemplo, para el duodécimo día del mes de febrero, la expresión da como resultado el número entero 212.

En MySQL tenemos una particularidad. Cada programa almacenado contiene un cuerpo que consta de una instrucción SQL. Esta declaración puede ser una declaración compuesta formada por varias declaraciones separadas por caracteres de punto y coma (;). Por ejemplo, el siguiente procedimiento almacenado tiene un cuerpo formado por un bloque BEGIN ... END que contiene una sentencia SET y un bucle REPEAT que a su vez contiene otra sentencia SET:

```
CREATE PROCEDURE dorepeat(p1 INT)
BEGIN
    SET @x = 0;
    REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
END;
```

Si usa el programa cliente mysql para definir un programa almacenado que contiene caracteres de punto y coma, surge un problema. Por defecto, mysql mismo reconoce el punto y coma como un delimitador de sentencia, por lo que debemos redefinir el delimitador temporalmente para que mysql pase la definición completa del programa almacenado al servidor. De otro modo, interpretaría que el programa termina en el punto y coma y esto daría error.

Para redefinir el delimitador mysql, utilizamos el comando **delimiter**. El siguiente ejemplo muestra cómo hacer esto para el procedimiento dorepeat() que se acaba de mostrar. El delimitador se cambia a // para permitir que la definición completa se pase al servidor como una declaración única y luego se restablezca a ; antes de invocar el procedimiento. Esto habilita el delimitador ; utilizado en el cuerpo del procedimiento para pasar al servidor en lugar de ser interpretado por mysql mismo.

```
mysql> delimiter //
mysql> CREATE PROCEDURE dorepeat(p1 INT)
-> BEGIN
->     SET @x = 0;
->     REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
-> END
-> //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> CALL dorepeat(1000);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x;
+-----+
| @x   |
+-----+
| 1001 |
+-----+
1 row in set (0.00 sec)
```

Entonces para crear la función ObtenerEdad en MySQL tenemos que ejecutar el siguiente código:



```

delimiter //
CREATE FUNCTION ObtenerEdad (
nacimiento DATE,
evento DATE
)
RETURNS INT DETERMINISTIC
BEGIN
    DECLARE edad INT;
    -- Calcular restando el año de la fecha de nacimiento a la fecha del evento
    SET edad = YEAR(evento)-YEAR(nacimiento);
    -- Ajustar en caso de que la fecha de nacimiento aún no haya ocurrido
    IF MONTH(nacimiento) > MONTH(evento) OR (MONTH(nacimiento) = MONTH(evento) AND
DAY(nacimiento) > DAY(evento)) THEN
        SET edad = edad - 1;
    END IF;
    RETURN edad;
END;
//
delimiter ;

```

Tengan en cuenta que una función puede tener más que sólo una cláusula **RETURN** en su cuerpo. Puede tener código con elementos de flujo, cálculos y más. Pero la función debe tener al menos una cláusula **RETURN** que devuelva un valor.

Para demostrar el uso de una FDU en una consulta, el siguiente código consulta la tabla *Employees* e invoca la función *ObtenerEdad* en la lista SELECT para calcular la edad de cada empleado hoy. Primero, en SQL Server:

```

SELECT
EmployeeID, FirstName, LastName, BirthDate,
dbo.ObtenerEdad(birthdate, SYSDATETIME()) AS Edad
FROM Employees;

```

Por ejemplo, si ejecutamos esta consulta el 11 de abril de 2023, obtendríamos el siguiente resultado:

EmployeeID	FirstName	LastName	BirthDate	Edad
1	Nancy	Davolio	1948-12-08 00:00:00.000	74
2	Andrew	Fuller	1952-02-19 00:00:00.000	71
3	Janet	Leverling	1963-08-30 00:00:00.000	59
4	Margaret	Peacock	1937-09-19 00:00:00.000	85
5	Steven	Buchanan	1955-03-04 00:00:00.000	68
6	Michael	Suyama	1963-07-02 00:00:00.000	59
7	Robert	King	1960-05-29 00:00:00.000	62
8	Laura	Callahan	1958-01-09 00:00:00.000	65
9	Anne	Dodsworth	1966-01-27 00:00:00.000	57

Y en MySQL:

```

SELECT
EmployeeID, FirstName, LastName, BirthDate,
ObtenerEdad(birthdate, CURDATE()) AS Edad
FROM Employees;

```



	EmployeeID	FirstName	LastName	BirthDate	Edad
▶	1	Nancy	Davolio	1948-12-08 00:00:00	74
	2	Andrew	Fuller	1952-02-19 00:00:00	71
	3	Janet	Leverling	1963-08-30 00:00:00	59
	4	Margaret	Peacock	1937-09-19 00:00:00	85
	5	Steven	Buchanan	1955-03-04 00:00:00	68
	6	Michael	Suyama	1963-07-02 00:00:00	59
	7	Robert	King	1960-05-29 00:00:00	62
	8	Laura	Callahan	1958-01-09 00:00:00	65
	9	Anne	Dodsworth	1966-01-27 00:00:00	57

Naturalmente, si ejecutan la consulta en su sistema, los valores que obtienen en la columna de edad dependerán de la fecha en la que ejecutan la consulta.

Si observan los ejemplos, pueden notar que en el de SQL Server comienza con **CREATE OR ALTER**, esto indica que en SQL Server se puede crear o modificar la función. En cambio, en MySQL para modificar una función, hay que eliminarla primero con **DROP FUNCTION** y luego volverla a crear con las modificaciones.

PROCEDIMIENTOS ALMACENADOS (STORED PROCEDURES)

Los procedimientos almacenados son rutinas que encapsulan código. Pueden tener parámetros de entrada y salida, pueden devolver conjuntos de resultados de consultas y pueden tener efectos secundarios. No solo se pueden modificar datos a través de procedimientos almacenados, sino que también se pueden aplicar cambios de esquema a través de ellos.

En comparación con el uso de código ad-hoc, el uso de procedimientos almacenados le brinda muchos beneficios:

- **Los procedimientos almacenados encapsulan la lógica.** Si necesitamos cambiar la implementación de un procedimiento almacenado, aplicamos el cambio en un lugar usando el comando **ALTER PROC** en SQL Server o **DROP PROCEDURE** y **CREATE PROCEDURE** en MySQL, y todos los usuarios del procedimiento usarán la nueva versión desde ese punto.
- **Los procedimientos almacenados brindan un mejor control de la seguridad.** Podemos otorgar permisos a un usuario para ejecutar el procedimiento sin otorgar permisos directos al usuario para realizar las actividades subyacentes. Por ejemplo, supongamos que deseamos permitir que ciertos usuarios eliminan un cliente de la base de datos, pero no deseamos otorgarles permisos directos para eliminar filas de la tabla Clientes. Deseamos asegurarnos de que las solicitudes para eliminar un cliente se validen, por ejemplo, verificando si el cliente tiene pedidos abiertos o deudas abiertas, y es posible que también deseemos auditar las solicitudes. Al no otorgar permisos directos para eliminar filas de la tabla Clientes, sino otorgar permisos para ejecutar un procedimiento que maneja la tarea, nos aseguramos de que siempre se realicen todas las validaciones y auditorías requeridas. Además, los procedimientos almacenados con parámetros pueden ayudar a evitar la inyección de SQL, especialmente cuando reemplazan el SQL ad-hoc enviado desde la aplicación cliente.
- **Podemos incorporar todo el código de manejo de errores dentro de un procedimiento, tomando silenciosamente acciones correctivas cuando sea relevante.** Veremos el manejo de errores más adelante.
- **Los procedimientos almacenados brindan beneficios de rendimiento.** Las consultas en los procedimientos almacenados suelen estar parametrizadas y, por lo tanto, es muy probable que reutilicen planes previamente almacenados en caché, lo que genera beneficios en el rendimiento. Otro beneficio de rendimiento del uso de procedimientos almacenados es la reducción del tráfico de red. La aplicación cliente debe enviar solo el



nombre del procedimiento y sus argumentos al SGBD. El servidor procesa todo el código del procedimiento y devuelve solo la salida a la persona que llama. No hay tráfico de ida y vuelta asociado con los pasos intermedios del procedimiento.

Como ejemplo simple, el siguiente código crea un procedimiento almacenado llamado *dbo.ObtenerOrdenesCliente*. El procedimiento acepta un ID de cliente (@clienteid) y un intervalo de fechas (@desde y @hasta) como entradas. El procedimiento devuelve filas de la tabla *Orders* que representan los pedidos realizados por el cliente solicitado en el intervalo de fechas solicitado como conjunto de resultados y el número de filas afectadas como parámetro de salida (@numrows):

```
CREATE OR ALTER PROC dbo.ObtenerOrdenesCliente
@clienteid AS NCHAR(5),
@desde AS DATETIME = '19000101',
@hasta AS DATETIME = '99991231',
@numrows AS INT OUTPUT
AS
SET NOCOUNT ON;
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM Orders
WHERE CustomerID = @clienteid
AND OrderDate >= @desde
AND OrderDate < @hasta;

SET @numrows = @@ROWCOUNT;
```

Al ejecutar el procedimiento, si no especificamos un valor en el parámetro @desde, el procedimiento usará el valor predeterminado 19000101, y si no especificamos un valor en el parámetro @hasta, el procedimiento usará el valor predeterminado 99991231. Observemos el uso de la palabra clave OUTPUT para indicar que el parámetro @numrows es un parámetro de salida. El comando SET NOCOUNT ON se usa para suprimir mensajes que indican cuántas filas se vieron afectadas por las declaraciones DML, como la declaración SELECT dentro del procedimiento.

En SQL Server podemos ejecutar los stored procedures mediante el comando EXECUTE o EXEC. Con respecto a los parámetros, SQL Server brinda dos alternativas. Una es pasando los valores de manera directa y otra es especificando los nombres de los parámetros. En MySQL se ejecutan mediante CALL. Para pasar parámetros, solamente admite el modo directo, y los parámetros van entre paréntesis.

En MySQL no tenemos la alternativa de manejar parámetros opcionales, por lo que tendríamos que enviar siempre los parámetros, pero podríamos definir que en el caso de no contar con los parámetros se envíe NULL, y en el código verificamos si los parámetros tienen valor NULL y le asignamos los valores por default. Si bien esto es una manera de solucionarlo, puede no ajustarse a lo que requiere el problema, ya que en algunos casos podría ser necesario enviar NULL como parámetro a un procedure. En ese caso deberíamos tomar otro valor para indicar que use el default. En el siguiente código vemos cómo hacerlo usando NULL:



```

DELIMITER //
CREATE PROCEDURE ObtenerOrdenesCliente(IN clienteid CHAR(5), IN desde DATETIME, IN hasta
DATETIME, OUT numrows INT)
BEGIN
    DECLARE rowCount INT;
    -- Verificar si los parámetros desde y hasta son nulos, y asignar valores predeterminados
    si es necesario
    IF desde IS NULL THEN
        SET desde = '19000101';
    END IF;

    IF hasta IS NULL THEN
        SET hasta = '99991231';
    END IF;

    -- Consulta para obtener las ordenes del cliente
    SELECT OrderID, CustomerID, EmployeeID, OrderDate
    FROM Orders
    WHERE CustomerID = clienteid
        AND OrderDate >= desde
        AND OrderDate < hasta;

    -- Obtener el número de filas afectadas
    SET rowCount = ROW_COUNT();

    -- Asignar el valor del número de filas afectadas a la variable de salida
    SET numrows = rowCount;
END;
// 
DELIMITER ;

```

Aquí hay un ejemplo de ejecución del procedimiento, solicitando información sobre los pedidos realizados por el cliente con la ID de ALFKI en el año 1997. El código toma el valor del parámetro de salida @numrows en la variable local @rc y lo devuelve para mostrar cuántos las filas se vieron afectadas por la consulta:

```

DECLARE @rc AS INT;
EXEC dbo.ObtenerOrdenesCliente
@clienteid = 'ALFKI',
@desde = '19970101',
@hasta = '19980101',
@numrows = @rc OUTPUT;
SELECT @rc AS numrows;

```

El código devuelve el siguiente resultado, que muestra tres pedidos que cumplen la condición:

OrderID	CustomerID	EmployeeID	OrderDate
10643	ALFKI	6	1997-08-25 00:00:00.000
10692	ALFKI	4	1997-10-03 00:00:00.000
10702	ALFKI	4	1997-10-13 00:00:00.000

numrows

3



En cambio, en MySQL utilizamos CALL para realizar la llamada:

```
CALL ObtenerOrdenesCliente('ALFKI', '19970101', '19980101', @rc);
SELECT @rc;
```

Si volvemos a ejecutar el código y proporcionamos un ID de cliente que no existe en la tabla Orders (por ejemplo, ID de cliente AGFA). Obtenemos el siguiente resultado, que indica que no hay pedidos calificados:

OrderID	CustomerID	EmployeeID	OrderDate
-----	-----	-----	-----
numrows			

0			

Para realizar la llamada sin fechas definidas, y utilizar los valores por default, en MySQL usaremos NULL como parámetro de la siguiente manera:

```
CALL ObtenerOrdenesCliente('ALFKI', NULL, NULL, @rc);
SELECT @rc;
```

En SQL Server podemos hacerlo de la siguiente manera:

```
DECLARE @rc AS INT;
EXEC dbo.ObtenerOrdenesCliente
@clienteid = 'ALFKI',
@numrows = @rc OUTPUT;
SELECT @rc AS numrows;
```

En SQL Server también tenemos la opción de pasar los parámetros de manera directa, de manera similar a CALL. Y en el caso de querer usar los valores por default, usamos como término **DEFAULT**:

```
DECLARE @rc AS INT;
EXEC dbo.ObtenerOrdenesCliente 'ALFKI', DEFAULT, DEFAULT, @rc;
SELECT @rc AS numrows;
```

DISPARADORES (TRIGGERS)

Un disparador (trigger) es un tipo especial de procedimiento almacenado, uno que no se puede ejecutar explícitamente. En cambio, está adjunto a un evento. Cada vez que se produce el evento, el disparador se activa y el código del disparador se ejecuta. SQL Server admite la asociación de disparadores con dos tipos de eventos: eventos de manipulación de datos (activadores DML) como INSERT y eventos de definición de datos (activadores DDL) como CREATE TABLE. Veremos solamente los disparadores DML

Podemos usar disparadores para muchos propósitos, incluida la auditoría, la aplicación de reglas de integridad que no se pueden aplicar con restricciones y la aplicación de políticas.

Un disparador se considera parte de la transacción que incluye el evento que hizo que se activara el disparador. Emitir un comando ROLLBACK TRAN dentro del código del disparador provoca una reversión de todos los cambios que tuvieron lugar en el disparador, y también de todos los cambios que tuvieron lugar en la transacción asociada con el disparador.

Los disparadores en SQL Server se disparan por sentencia y no por fila modificada. En cambio, en MySQL se disparan por cada fila modificada.



DISPARADORES DML EN SQL SERVER

SQL Server admite dos tipos de disparadores DML: after y instead of. Un disparador after se activa después de que finaliza el evento al que está asociado y solo se puede definir en tablas permanentes. Un disparador instead of se dispara en lugar del evento al que está asociado y se puede definir en tablas y vistas permanentes.

En el código del disparador, podemos acceder a pseudo tablas denominadas INSERTED y DELETED que contienen las filas que se vieron afectadas por la modificación que hizo que se lanzara el disparador.

La tabla INSERTED contiene la nueva imagen de las filas afectadas en el caso de las acciones INSERT y UPDATE. La tabla DELETED contiene la imagen anterior de las filas afectadas en el caso de las acciones DELETE y UPDATE. Recuerden que las acciones INSERT, UPDATE y DELETE pueden ser invocadas por las sentencias INSERT, UPDATE y DELETE, así como por la sentencia MERGE. En el caso de disparadores instead of, las tablas INSERTED y DELETED contienen las filas que se suponía que se verían afectadas por la modificación que hizo que se disparara el disparador.

El siguiente ejemplo simple de un disparador after permite auditar las inserciones a una tabla. Ejecuten el siguiente código para crear una tabla denominada dbo.T1 en la base de datos actual y una tabla denominada dbo.T1_Audit que contenga información de auditoría para las inserciones en T1:

```
DROP TABLE IF EXISTS dbo.T1_Audit, dbo.T1;
CREATE TABLE dbo.T1
(
keycol INT NOT NULL PRIMARY KEY,
datacol VARCHAR(10) NOT NULL
);
CREATE TABLE dbo.T1_Audit
(
audit_lsn INT NOT NULL IDENTITY PRIMARY KEY,
dt DATETIME2(3) NOT NULL DEFAULT(SYSDATETIME()),
login_name sysname NOT NULL DEFAULT(ORIGINAL_LOGIN()),
keycol INT NOT NULL,
datacol VARCHAR(10) NOT NULL
);
```

En la tabla de auditoría, la columna audit_lsn tiene una propiedad de identidad y representa un número de serie del registro de auditoría. La columna dt representa la fecha y la hora de la inserción, utilizando la expresión predeterminada SYSDATETIME(). La columna login_name representa el nombre del inicio de sesión que realizó la inserción, utilizando la expresión predeterminada ORIGINAL_LOGIN().

A continuación, ejecuten el siguiente código para crear el disparador AFTER INSERT trg_T1_insert_audit en la tabla T1 para auditar las inserciones:

```
CREATE OR ALTER TRIGGER trg_T1_insert_audit ON dbo.T1 AFTER INSERT
AS
SET NOCOUNT ON;
INSERT INTO dbo.T1_Audit(keycol, datacol)
SELECT keycol, datacol FROM inserted;
GO
```

Como pueden ver, el disparador simplemente inserta en la tabla de auditoría el resultado de una consulta en la tabla INSERTED. Los valores de las columnas de la tabla de auditoría que no se enumeran explícitamente en la instrucción INSERT se generan mediante las expresiones predeterminadas descritas anteriormente. Para probar el disparador, ejecuten el siguiente código:



```
INSERT INTO dbo.T1(keycol, datacol) VALUES(10, 'a');
INSERT INTO dbo.T1(keycol, datacol) VALUES(30, 'x');
INSERT INTO dbo.T1(keycol, datacol) VALUES(20, 'g');
```

El disparador se ejecuta después de cada sentencia. A continuación, consulten la tabla de auditoría:

```
SELECT audit_lsn, dt, login_name, keycol, datacol
FROM dbo.T1_Audit;
```

Obtienen el siguiente resultado, solo con los valores dt y login_name que reflejan la fecha y la hora en que ejecutaron las inserciones y el usuario con el que iniciaron sesión para conectarse a SQL Server:

audit_lsn	dt	login_name	keycol	datacol
1	2023-04-05 16:39:44.417	sa	10	a
2	2023-04-05 16:39:44.421	sa	30	x
3	2023-04-05 16:39:44.425	sa	20	g

Cuando hayan terminado, ejecuten el siguiente código para la limpieza:

```
DROP TABLE dbo.T1_Audit, dbo.T1;
```

DISPARADORES DML EN MYSQL

Un disparador es un objeto de base de datos con nombre que está asociado con una tabla y que se activa cuando ocurre un evento particular para la tabla. Algunos usos de los disparadores son para realizar comprobaciones de valores que se insertarán en una tabla o para realizar cálculos sobre valores involucrados en una actualización.

Se define un disparador para que se active cuando una sentencia inserta, actualiza o elimina filas en la tabla asociada. Estas operaciones de fila son eventos desencadenantes. Por ejemplo, las filas se pueden insertar mediante instrucciones INSERT o LOAD DATA, y se activa un disparador de inserción para cada fila insertada. Se puede configurar un disparador para que se active antes o después del evento de activación. Por ejemplo, puede hacer que se active un disparador antes de cada fila que se inserta en una tabla o después de cada fila que se actualiza.

Importante

Los disparadores de MySQL se activan solo para los cambios realizados en las tablas mediante sentencias SQL. Esto incluye cambios en las tablas base que subyacen a las vistas actualizables. Los activadores no se activan para los cambios en las tablas realizados por las API que no transmiten sentencias SQL al servidor MySQL. Esto significa que los disparadores no se activan mediante actualizaciones realizadas con la API de NDB.

Los disparadores no se activan mediante cambios en las tablas INFORMATION_SCHEMA o performance_schema. Esas tablas son en realidad vistas y los disparadores no están permitidos en las vistas.

Para crear un disparador o eliminar un activador, se utiliza la declaración CREATE TRIGGER o DROP TRIGGER.

Aquí hay un ejemplo simple que asocia un disparador con una tabla, para que se active en operaciones INSERT. El disparador actúa como un acumulador, sumando los valores insertados en una de las columnas de la tabla.



```
mysql> CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
Query OK, 0 rows affected (0.03 sec)
```

```
mysql> CREATE TRIGGER ins_sum BEFORE INSERT ON account
FOR EACH ROW SET @sum = @sum + NEW.amount;
Query OK, 0 rows affected (0.01 sec)
```

La instrucción CREATE TRIGGER crea un disparador llamado ins_sum que está asociado con la tabla de account. También incluye cláusulas que especifican el tiempo de acción del disparador, el evento activador y qué hacer cuando se activa el activador:

- La palabra clave BEFORE indica el tiempo de acción del disparador. En este caso, el disparador se activa antes de cada fila insertada en la tabla. La otra palabra clave permitida aquí es AFTER.
- La palabra clave INSERT indica el evento desencadenante; es decir, el tipo de operación que activa el disparador. En el ejemplo, las operaciones INSERT provocan la activación del disparador. También se puede crear disparadores para operaciones DELETE y UPDATE.

La instrucción que sigue a FOR EACH ROW define el cuerpo del disparador; es decir, la sentencia a ejecutar cada vez que se activa el activador, lo que ocurre una vez por cada fila afectada por el evento activador. En el ejemplo, el cuerpo del disparador es un SET simple que acumula en una variable de usuario los valores insertados en la columna de cantidad. La declaración se refiere a la columna como NEW.cantidad, lo que significa "el valor de la columna de la cantidad que se insertará en la nueva fila".

Para usar el disparador, establezcan la variable del acumulador en cero, ejecute una declaración INSERT y luego vean qué valor tiene la variable después:

```
mysql> SET @sum = 0;
mysql> INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);
mysql> SELECT @sum AS 'Total amount inserted';
+-----+
| Total amount inserted |
+-----+
| 1852.48 |
+-----+
```

En este caso, el valor de @sum después de ejecutar la instrucción INSERT es 14,98 + 1937,50 - 100 o 1852,48.

Para eliminar el disparador, utilicen una sentencia DROP TRIGGER. Deben especificar el nombre del esquema si el disparador no está en el esquema predeterminado:

```
mysql> DROP TRIGGER test.ins_sum;
```

Si eliminamos una tabla, también se eliminarán todos los disparadores de la tabla.

Los nombres de los disparadores existen en el espacio de nombres del esquema, lo que significa que todos los disparadores deben tener nombres únicos dentro de un esquema. Los disparadores en diferentes esquemas pueden tener el mismo nombre.

Es posible definir múltiples disparadores para una tabla determinada que tengan el mismo evento de activación y tiempo de acción. Por ejemplo, puede tener dos disparadores BEFORE UPDATE para una tabla. De forma predeterminada, los disparadores que tienen el mismo evento de disparador y tiempo de acción se activan en el orden en que fueron creados. Para modificar el orden de activación, especifiquen una cláusula después de FOR EACH ROW



que indique **FOLLOW**s o **PRECEDES** y el nombre de un activador existente que también tenga el mismo evento de activación y tiempo de acción. Con **FOLLOW**s, el nuevo disparador se activa después del disparador existente. Con **PRECEDES**, el nuevo disparador se activa antes que el disparador existente.

Por ejemplo, la siguiente definición de disparador define otro disparador **BEFORE INSERT** para la tabla de cuentas:

```
mysql> CREATE TRIGGER ins_transaction BEFORE INSERT ON account
    FOR EACH ROW PRECEDES ins_sum
    SET
        @deposits = @deposits + IF(NEW.amount > 0, NEW.amount, 0),
        @withdrawals = @withdrawals + IF(NEW.amount < 0, -NEW.amount, 0);
Query OK, 0 rows affected (0.01 sec)
```

Este disparador, **ins_transaction**, es similar a **ins_sum** pero acumula depósitos y retiros por separado. Tiene una cláusula **PRECEDES** que hace que se active antes de **ins_sum**; sin esa cláusula, se activaría después de **ins_sum** porque se creó después de **ins_sum**.

Dentro del cuerpo del disparador, las palabras clave **OLD** y **NEW** permiten acceder a las columnas en las filas afectadas por un disparador. **OLD** y **NEW** son extensiones de MySQL para disparadores; no distinguen entre mayúsculas y minúsculas.

En un disparador **INSERT**, solo se puede usar **NEW.col_name**; no hay fila **OLD**. En un disparador **DELETE**, solo se puede usar **OLD.col_name**; no hay fila **NEW**. En un disparador **UPDATE**, puede usar **OLD.col_name** para hacer referencia a las columnas de una fila antes de que se actualice y **NEW.col_name** para hacer referencia a las columnas de la fila después de que se actualice.

Una columna denominada **OLD** es de solo lectura. Pueden consultarla (si tienen el privilegio **SELECT**), pero no modificarla. Pueden hacer referencia a una columna con el nombre **NEW** si tienen el privilegio **SELECT** para ella. En un disparador **BEFORE**, también pueden cambiar su valor con **SET NEW.col_name = valor** si tienen el privilegio **UPDATE** para ello. Esto significa que pueden usar un disparador para modificar los valores que se insertarán en una nueva fila o se usarán para actualizar una fila. (Dicha instrucción **SET** no tiene efecto en un disparador **DESPUÉS** porque el cambio de fila ya ocurrió).

En un disparador **BEFORE**, el valor **NEW** para una columna **AUTO_INCREMENT** es 0, no el número de secuencia que se genera automáticamente cuando se inserta realmente la nueva fila.

Mediante el uso de la construcción **BEGIN ... END**, pueden definir un disparador que ejecute varias instrucciones.

Dentro del bloque **BEGIN**, también pueden usar otra sintaxis permitida dentro de las rutinas almacenadas, como condicionales y bucles. Sin embargo, al igual que con las rutinas almacenadas, si usan el programa **mysql** para definir un disparador que ejecuta varias sentencias, es necesario redefinir el delimitador de sentencia **mysql** para que pueda usar el ; delimitador de instrucción dentro de la definición del disparador. El siguiente ejemplo ilustra estos puntos. Define un disparador de **UPDATE** que verifica el nuevo valor que se usará para actualizar cada fila y modifica el valor para que esté dentro del rango de 0 a 100. Este debe ser un disparador **BEFORE** porque el valor debe verificarse antes de que se use para actualizar la fila:



```

mysql> delimiter //
mysql> CREATE TRIGGER upd_check BEFORE UPDATE ON account
FOR EACH ROW
BEGIN
    IF NEW.amount < 0 THEN
        SET NEW.amount = 0;
    ELSEIF NEW.amount > 100 THEN
        SET NEW.amount = 100;
    END IF;
END;//
mysql> delimiter ;

```

Puede ser más fácil definir un procedimiento almacenado por separado y luego invocarlo desde el disparador usando una instrucción CALL simple. Esto también es ventajoso si desean ejecutar el mismo código desde varios disparadores.

Existen limitaciones sobre lo que puede aparecer en las declaraciones que ejecuta un disparador cuando se activa:

- El disparador no puede utilizar la instrucción CALL para invocar procedimientos almacenados que devuelven datos al cliente o que utilizan SQL dinámico. (Se permite que los procedimientos almacenados devuelvan datos al disparador a través de los parámetros OUT o INOUT).
- El disparador no puede utilizar instrucciones que explícita o implícitamente comiencen o finalicen una transacción, como START TRANSACTION, COMMIT o ROLLBACK. (Se permite ROLLBACK a SAVEPOINT porque no finaliza una transacción).

MySQL maneja los errores durante la ejecución del disparador de la siguiente manera:

- Si falla un disparador BEFORE, no se realiza la operación en la fila correspondiente.
- Un disparador BEFORE se activa por el intento de insertar o modificar la fila, independientemente de si el intento tiene éxito posteriormente.
- Un disparador AFTER se ejecuta solo si cualquier disparador BEFORE y la operación de fila se ejecutan correctamente.
- Un error durante un disparador BEFORE o AFTER da como resultado un error de toda la instrucción que provocó la invocación del desencadenador.
- En el caso de las tablas transaccionales, el fallo de una sentencia debería provocar la reversión de todos los cambios realizados por la declaración. La falla de un disparador hace que la sentencia falle, por lo que la falla del disparador también provoca la reversión. Para las tablas no transaccionales, dicha reversión no se puede realizar, por lo que, aunque la instrucción falla, los cambios realizados antes del punto del error permanecen vigentes.

Los disparadores pueden contener referencias directas a tablas por nombre, como el disparador llamado testref que se muestra en este ejemplo:

```

CREATE TABLE test1(a1 INT);
CREATE TABLE test2(a2 INT);
CREATE TABLE test3(a3 INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE test4(
    a4 INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    b4 INT DEFAULT 0
);

```



```
delimiter |
CREATE TRIGGER testref BEFORE INSERT ON test1
FOR EACH ROW
BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
END;|

```

```
delimiter ;
INSERT INTO test3 (a3) VALUES
    (NULL), (NULL), (NULL), (NULL), (NULL),
    (NULL), (NULL), (NULL), (NULL);
```

```
INSERT INTO test4 (a4) VALUES
    (0), (0), (0), (0), (0), (0), (0), (0);
```

Supongan que insertan los siguientes valores en la tabla test1 como se muestra aquí:

```
mysql> INSERT INTO test1 VALUES
    (1), (3), (1), (7), (1), (8), (4), (4);
Query OK, 8 rows affected (0.01 sec)
Records: 8  Duplicates: 0  Warnings: 0
```

Como resultado, las cuatro tablas contienen los siguientes datos:

```
mysql> SELECT * FROM test1;
+----+
| a1 |
+----+
| 1 |
| 3 |
| 1 |
| 7 |
| 1 |
| 8 |
| 4 |
| 4 |
+----+
8 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM test2;
+----+
| a2 |
+----+
| 1 |
| 3 |
| 1 |
| 7 |
| 1 |
| 8 |
| 4 |
| 4 |
+----+
8 rows in set (0.00 sec)
```



```
mysql> SELECT * FROM test3;
```

a3
2
5
6
9
10

5 rows in set (0.00 sec)

```
mysql> SELECT * FROM test4;
```

a4	b4
1	3
2	0
3	1
4	2
5	0
6	0
7	1
8	1
9	0
10	0

10 rows in set (0.00 sec)



PROGRAMANDO

La mayor utilidad de las rutinas es poder realizar tareas complejas, que requieren el uso de programación. Para esto vamos a utilizar variables, elementos de flujo de control, cursores y tablas temporales.

VARIABLES

Las variables se utilizan para almacenar valores temporalmente.

Se utiliza *DECLARE* para declarar una o más variables, y se utiliza *SET* para asignar valor a una variable. Por ejemplo, en el siguiente código declaramos una variable llamada *@i* del tipo *INT* y le asignamos el valor *10*:

```
DECLARE @i AS INT;  
SET @i = 10;
```

En SQL Server se puede declarar la variable e inicializarla en la misma sentencia, como aquí:

```
DECLARE @i AS INT = 10;
```

En MySQL se puede hacer lo mismo, pero utilizando *DEFAULT*:

```
DECLARE @i INT DEFAULT 10;
```

Cuando asignamos valor a una variable escalar, este debe ser el resultado de una expresión escalar. La expresión puede ser una subconsulta escalar. Por ejemplo, en el siguiente código declaramos una variable llamada *@nombreemp* y le asignamos el resultado de una consulta escalar que devuelve el nombre completo del empleado con ID igual a 3:

```
USE Northwind;  
  
DECLARE @nombreemp AS NVARCHAR(61);  
  
SET @nombreemp = (SELECT CONCAT(FirstName, ' ', LastName)  
                  FROM Employees  
                 WHERE EmployeeID = 3);  
  
SELECT @nombreemp AS nombreemp;
```

Este código devuelve la siguiente salida:

```
nombreemp  
-----  
Janet Leverling
```

En MySQL no necesitamos utilizar *DECLARE* fuera de las rutinas.

```
USE northwind;  
  
SET @nombreemp = (SELECT CONCAT(FirstName, ' ', LastName)  
                  FROM Employees  
                 WHERE EmployeeID = 3);  
  
SELECT @nombreemp AS nombreemp;
```

La sentencia *SET* puede operar solamente sobre una variable a la vez, así que, si necesitamos asignar valores a múltiples variables, necesitamos múltiples sentencias *SET*. Esto puede implicar mucho trabajo adicional cuando



necesitamos extraer múltiples atributos de una misma fila. Por ejemplo, en el siguiente código utilizaremos dos *SET* para extraer el nombre y el apellido respectivamente, del empleado con ID igual a 3:

```
DECLARE @nombre AS NVARCHAR(20), @apellido AS NVARCHAR(40);

SET @nombre = (SELECT FirstName
               FROM Employees
              WHERE EmployeeID = 3);
SET @apellido = (SELECT LastName
                  FROM Employees
                 WHERE EmployeeID = 3);

SELECT @nombre AS nombre, @apellido AS apellido;
```

Este código devuelve la siguiente salida:

nombre	apellido
Janet	Leverling

T-SQL también soporta el uso de *SELECT* como sentencia de asignación (que no es estándar), que puede utilizarse para consultar y asignar múltiples valores obtenidos de la misma fila a múltiples variables utilizando una sola sentencia. Como en el siguiente ejemplo:

```
DECLARE @nombre AS NVARCHAR(20), @apellido AS NVARCHAR(40);

SELECT @nombre = FirstName, @apellido = LastName
FROM Employees
WHERE EmployeeID = 3;

SELECT @nombre AS nombre, @apellido AS apellido;
```

En MySQL podemos hacer lo mismo utilizando el operador de asignación :=

```
SELECT @nombre := FirstName, @apellido := LastName
FROM Employees
WHERE EmployeeID = 3;

SELECT @nombre AS nombre, @apellido AS apellido;
```

Cuando la consulta tiene como resultado una sola fila, el comportamiento del *SELECT* es predecible.

Sin embargo, notemos que, si la consulta devuelve más de una fila, la sintaxis es correcta y el código no falla. La asignación tiene lugar para cada fila resultante, y con cada fila accedida, los valores de la fila sobrescriben el valor existente de las variables. Cuando la asignación finaliza, los valores en las variables serán los de la última fila que se haya evaluado. Por ejemplo, la siguiente asignación tiene cinco filas como resultado:

```
DECLARE @nombreemp AS NVARCHAR(61);

SELECT @nombreemp = FirstName + N' ' + LastName
FROM Employees
WHERE ReportsTo = 2;

SELECT @nombreemp AS nombreemp;
```



La información del empleado que termina en la variable luego de la asignación con *SELECT* depende del orden en el que SQL Server acceda a las filas – y no podemos controlar ese orden –

Cuando corrí este código, yo obtuve la siguiente salida:

```
nombreemp
```

```
Laura Callahan
```

La sentencia *SET* es más segura que la asignación con *SELECT* porque requiere el uso de una consulta escalar para obtener datos de una tabla. Recordemos que una consulta escalar falla en tiempo de ejecución si devuelve más de un valor. Por ejemplo, el siguiente código falla:

```
DECLARE @nombreemp AS NVARCHAR(61);

SET @nombreemp = (SELECT FirstName + N' ' + LastName
                  FROM Employees
                  WHERE ReportsTo = 2);

SELECT @nombreemp AS nombreemp;
```

Como no se asignó valor a la variable, permanece en *NULL*, que es el valor por default cuando las variables no son inicializadas. Este código devuelve la siguiente salida:

```
Msg 512, Level 16, State 1, Line 3
Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=,
<, <= , >, >= or when the subquery is used as an expression.
```

```
nombreemp
```

```
NULL
```

En MySQL obtendríamos el siguiente código de error:

```
Error Code: 1242. Subquery returns more than 1 row 0.047 sec
```

ELEMENTOS DE FLUJO

Usamos elementos de flujo para controlar el flujo de nuestro código. T-SQL proporciona formas básicas de control con elementos de flujo, incluido el *IF . . ELSE* y el *WHILE*. En MySQL contamos con más estructuras, como *LOOP* y *REPEAT*, y es diferente la sintaxis del bloque *IF* y de *WHILE*.

EL ELEMENTO DE FLUJO IF...ELSE

Usamos el *IF . . ELSE* para controlar el flujo de nuestro código basado en el resultado de un predicado. Especificamos una instrucción o un bloque de instrucciones que se ejecuta si el predicado es *VERDADERO* y, opcionalmente, una instrucción o un bloque de instrucciones que se ejecuta si el predicado es *FALSO* o *DESCONOCIDO*.

Por ejemplo, el siguiente código verifica si hoy es el último día del año (en otras palabras, si el año de hoy es diferente al de mañana). Si esto es cierto, el código imprime un mensaje que dice que hoy es el último día del año; si no es cierto ("else"), el código imprime un mensaje que dice que hoy no es el último día del año:



```

IF YEAR(SYSDATETIME()) <> YEAR(DATEADD(day, 1, SYSDATETIME()))
PRINT 'Hoy es el último día del año.';
ELSE
PRINT 'Hoy no es el último día del año.';
```

En este ejemplo utilizamos *PRINT* para demostrar qué partes del código son ejecutadas y qué partes no, pero por supuesto se puede utilizar cualquier sentencia.

Tengamos en cuenta que T-SQL utiliza lógica de tres valores y que el bloque *ELSE* se activa cuando el predicado es *FALSO* o *DESCONOCIDO*. En los casos en los que tanto *FALSO* como *DESCONOCIDO* son posibles resultados del predicado (por ejemplo, cuando están implicados los valores nulos) y necesita un tratamiento diferente para cada caso, asegúrese de realizar una prueba explícita para los valores nulos con el predicado *IS NULL*.

Si el flujo que necesita controlar involucra más de dos casos, puede anidar elementos *IF...ELSE*. Por ejemplo, el siguiente código maneja los siguientes tres casos de manera diferente:

- Hoy es el último día del año.
- Hoy es el último día del mes, pero no es el último día del año.
- Hoy no es el último día del mes.

```

IF YEAR(SYSDATETIME()) <> YEAR(DATEADD(day, 1, SYSDATETIME()))
    PRINT 'Hoy es el último día del año.';
ELSE
    IF MONTH(SYSDATETIME()) <> MONTH(DATEADD(day, 1, SYSDATETIME()))
        PRINT 'Hoy es el último día del mes, pero no es el último día del año.';
    ELSE
        PRINT 'Hoy no es el último día del mes.';
```

Si necesitamos ejecutar más de una sentencia en las secciones *IF* o *ELSE*, necesitamos usar un bloque de declaración. Marcamos los límites de un bloque de instrucciones con las palabras clave *BEGIN* y *END*. Por ejemplo, el siguiente código muestra cómo ejecutar un tipo de proceso si es el primer día del mes y otro tipo de proceso si no es:

```

IF DAY(SYSDATETIME()) = 1
BEGIN
    PRINT 'Hoy es el primer día del mes.';
    PRINT 'Comenzando el proceso primer-dia-de-mes.';
    /* ... acá iría el proceso ... */
    PRINT 'Se ha finalizado el proceso primer-dia-de-mes.';
END;
ELSE
BEGIN
    PRINT 'Hoy no es el primer día del mes.';
    PRINT 'Comenzando el proceso no-primer-dia-de-mes.';
    /* ... acá iría el proceso ... */
    PRINT 'Se ha finalizado el proceso no-primer-dia-de-mes.';
```

En MySQL la sintaxis del bloque es:

```

IF SEARCH_CONDITION THEN STATEMENT_LIST
[ELSEIF SEARCH_CONDITION THEN STATEMENT_LIST] ...
[ELSE STATEMENT_LIST]
END IF
```



Cada lista_sentencia (statement_list) consta de una o más sentencias SQL; no se permite una lista de sentencias vacía.

Un bloque IF ... END IF, como todos los demás bloques de control de flujo utilizados en los programas almacenados, debe terminar con un punto y coma, como se muestra en este ejemplo:

```
DELIMITER //

CREATE FUNCTION SimpleCompare(n INT, m INT)
RETURNS VARCHAR(20)

BEGIN
DECLARE s VARCHAR(20);

IF n > m THEN SET s = '>';
ELSEIF n = m THEN SET s = '=';
ELSE SET s = '<';
END IF;

SET s = CONCAT(n, ' ', s, ' ', m);

RETURN s;
END //

DELIMITER ;
```

WHILE

T-SQL cuenta con *WHILE*, mediante el cual podemos ejecutar código en un bucle. *WHILE* ejecuta una sentencia o un bloque de sentencias repetidas veces mientras que el predicado que se especificó se evalúe como *VERDADERO*. Cuando el predicado se evalúa como *FALSO* o como *DESCONOCIDO*, el bucle termina.

T-SQL no cuenta con un elemento de flujo que permita ejecutar un número predeterminado de veces (como es el *FOR* en otros lenguajes), pero este comportamiento puede ser emulado con un *WHILE* y una variable. Por ejemplo, en el siguiente código veremos cómo emular un bucle que se ejecute 10 veces:

```
DECLARE @i AS INT = 1;
WHILE @i <= 10
BEGIN
    PRINT @i;
    SET @i = @i + 1;
END;
```

En el código declaramos una variable entera llamada *@i* que es utilizada como contador en el bucle y que se inicializa con el valor 1. El código entra en el bucle y se ejecuta mientras que la variable sea menor o igual a 10. En cada iteración, el código imprime el valor actual de *@i* y lo incrementa en 1. El código devuelve la siguiente salida, mostrando que la iteración se ejecutó 10 veces:

```
1
2
3
4
5
6
7
8
9
10
```



Si en algún punto del cuerpo del bucle deseamos salir del bucle actual y proceder a ejecutar la instrucción que aparece después del cuerpo del bucle, usamos el comando *BREAK*. Por ejemplo, el siguiente código sale del bucle si el valor de *@i* es igual a 6:

```
DECLARE @i AS INT = 1;
WHILE @i <= 10
BEGIN
    IF @i = 6 BREAK;
    PRINT @i;
    SET @i = @i + 1;
END;
```

Este código produce la siguiente salida que muestra que el bucle iteró cinco veces y terminó al comienzo de la sexta iteración:

```
1
2
3
4
5
```

Por supuesto, este código no es muy notable; si deseamos que el bucle se repita solo cinco veces, simplemente debemos especificar el predicado *@i <= 5*. Aquí solo quería demostrar el uso del comando *BREAK* con un simple ejemplo.

Si en algún punto del cuerpo del bucle desea omitir el resto de la actividad en la iteración actual y evaluar nuevamente el predicado del bucle, use el comando *CONTINUE*. Por ejemplo, el siguiente código muestra cómo omitir la actividad de la sexta iteración del bucle desde el punto donde aparece la instrucción *IF* y hasta el final del cuerpo del bucle:

```
DECLARE @i AS INT = 0;
WHILE @i < 10
BEGIN
    SET @i = @i + 1;
    IF @i = 6 CONTINUE;
    PRINT @i;
END;
```

La salida de este código muestra que el valor de *@i* fue impreso en todas las iteraciones, menos en la sexta:

```
1
2
3
4
5
7
8
9
10
```

Como otro ejemplo del uso de *WHILE*, el siguiente código crea una tabla llamada *dbo.Numeros* y la llena con 1.000 filas con los valores desde 1 hasta 1.000 en la columna *n*:



```

SET NOCOUNT ON;
DROP TABLE IF EXISTS dbo.Numeros;
CREATE TABLE dbo.Numeros(n INT NOT NULL PRIMARY KEY);
GO

DECLARE @i AS INT = 1;
WHILE @i <= 1000
BEGIN
    INSERT INTO dbo.Numeros(n)
        VALUES(@i);
    SET @i = @i + 1;
END;

```

En MySQL la sintaxis de WHILE es:

```
[BEGIN_LABEL:] WHILE SEARCH_CONDITION DO
    STATEMENT_LIST
END WHILE [END_LABEL]
```

La lista de sentencias dentro de una sentencia WHILE se repite siempre que la expresión condición_búsqueda (search_condition) sea verdadera. lista_sentencias (statement_list) consta de una o más sentencias SQL, cada una terminada por un delimitador de sentencia de punto y coma (;).

Una instrucción WHILE se puede etiquetar.

Ejemplo:

```

CREATE PROCEDURE dowhile()
BEGIN
    DECLARE v1 INT DEFAULT 5;

    WHILE v1 > 0 DO
        ...
        SET v1 = v1 - 1;
    END WHILE;
END;
```

REPEAT (MYSQL)

```
[BEGIN_LABEL:] REPEAT
    STATEMENT_LIST
UNTIL SEARCH_CONDITION
END REPEAT [END_LABEL]
```

La lista de sentencias (statement_list) dentro de una sentencia REPEAT se repite hasta que la expresión condición_búsqueda (search_condition) sea verdadera. Por lo tanto, un REPEAT siempre ingresa al bucle al menos una vez. lista_sentencia (statement_list) consta de una o más sentencias, cada una terminada por un delimitador de sentencia de punto y coma (;).

Una instrucción REPEAT se puede etiquetar.

Ejemplo:



```
mysql> delimiter //
mysql> CREATE PROCEDURE dorepeat(p1 INT)
BEGIN
    SET @x = 0;
    REPEAT
        SET @x = @x + 1;
    UNTIL @x > p1 END REPEAT;
END
//
Query OK, 0 rows affected (0.00 sec)

mysql> CALL dorepeat(1000)//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x//
+-----+
| @x   |
+-----+
| 1001 |
+-----+
1 row in set (0.00 sec)
```

LOOP (MYSQL)

```
[BEGIN_LABEL:] LOOP
    STATEMENT_LIST
END LOOP [END_LABEL]
```

LOOP implementa una construcción de bucle simple, que permite la ejecución repetida de la lista de sentencias, que consta de una o más sentencias, cada una terminada con un delimitador de sentencia de punto y coma (;). Las sentencias dentro del bucle se repiten hasta que se termina el bucle. Por lo general, esto se logra con una sentencia LEAVE. Dentro de una función almacenada o stored procedure, también se puede usar RETURN, que sale de la función por completo.

Omitir incluir una declaración de terminación de bucle da como resultado un bucle infinito.

Una instrucción LOOP se puede etiquetar.

Ejemplo:

```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
    label1: LOOP
        SET p1 = p1 + 1;
        IF p1 < 10 THEN
            ITERATE label1;
        END IF;
        LEAVE label1;
    END LOOP label1;
    SET @x = p1;
END;
```



TRANSACCIONES Y CONCURRENCIA

Veremos a continuación las transacciones y sus propiedades y veremos cómo Microsoft SQL Server maneja a los usuarios que intentan acceder a los mismos datos al mismo tiempo. Explicaremos cómo SQL Server usa bloqueos para aislar datos inconsistentes, cómo podemos solucionar situaciones de bloqueo y cómo podemos controlar el nivel de coherencia cuando consultamos datos con niveles de aislamiento. También veremos interbloqueos y formas de mitigar su ocurrencia.

Nos enfocaremos en los aspectos de concurrencia de la representación de datos tradicional en tablas basadas en disco.

TRANSACCIONES

Una transacción es una unidad de trabajo que puede incluir múltiples actividades que consultan y modifican datos y que también pueden cambiar la definición de datos.

Podemos definir los límites de las transacciones de forma explícita o implícita. El comienzo de una transacción se define explícitamente con una instrucción **BEGIN TRAN** (o **BEGIN TRANSACTION**). El final de una transacción se define explícitamente con una declaración **COMMIT TRAN** si deseamos confirmarla y con una declaración **ROLLBACK TRAN** (o **ROLLBACK TRANSACTION**) si deseamos deshacer sus cambios. Aquí hay un ejemplo de cómo marcar los límites de una transacción con dos instrucciones **INSERT**:

```
BEGIN TRAN;  
INSERT INTO dbo.T1(keycol, col1, col2) VALUES(4, 101, 'C');  
INSERT INTO dbo.T2(keycol, col1, col2) VALUES(4, 201, 'X');  
COMMIT TRAN;
```

Si no marcamos explícitamente los límites de una transacción, de forma predeterminada, SQL Server trata cada instrucción individual como una transacción; en otras palabras, de forma predeterminada, SQL Server inicia automáticamente una transacción antes de que comience cada sentencia y confirma la transacción al final de cada sentencia. Este modo se conoce como modo de confirmación automática (auto-commit).

Podemos cambiar la forma en que SQL Server maneja las transacciones implícitas para que no se confirme automáticamente activando una opción de sesión llamada **IMPLICIT_TRANSACTIONS**. Esta opción está desactivada de forma predeterminada. Cuando esta opción está activada, no tenemos que especificar la declaración **BEGIN TRAN** para marcar el comienzo de una transacción, pero debemos marcar el final de la transacción con una declaración **COMMIT TRAN** o **ROLLBACK TRAN**.

Después de que una transacción se confirma o revierte, a menos que abramos otra transacción explícita, la siguiente declaración ejecutada implícitamente inicia otra transacción.

Recordemos que las transacciones tienen cuatro propiedades: **Atomicidad**, **Consistencia**, **aislamiento (Isolation)** y **Durabilidad**, abreviadas con el acrónimo **ACID**:

- **Atomicidad:** Una transacción es una unidad atómica de trabajo. O se realizan todos los cambios en la transacción o no se realiza ninguno. Si el sistema falla antes de que se complete una transacción (antes de que la instrucción de confirmación se registre en el registro de transacciones), al reiniciar, SQL Server deshace los cambios realizados. Además, si se encuentran errores durante la transacción y el error se considera lo suficientemente grave, como que el grupo de archivos de destino está lleno cuando intenta insertar datos, SQL Server revierte automáticamente la transacción. Algunos errores, como las infracciones de clave principal y los tiempos de espera de vencimiento de bloqueo, no se consideran lo suficientemente graves como para justificar una



reversión automática de la transacción. Si deseamos que todos los errores cancelen la ejecución y provoquen la reversión de cualquier transacción abierta, podemos habilitar una opción de sesión llamada **XACT_ABORT**. Podemos usar el código de manejo de errores para capturar dichos errores y aplicar algún curso de acción (por ejemplo, registrar el error y revertir la transacción).

- **Consistencia:** el término consistencia se refiere al estado de los datos a los que el sistema de administración de bases de datos relacionales (RDBMS) le da acceso a medida que las transacciones simultáneas los modifican y consultan. Como probablemente podemos imaginar, la consistencia es un término subjetivo, que depende de las necesidades de cada aplicación. La sección "Niveles de aislamiento" más adelante explica el nivel de coherencia que proporciona SQL Server de forma predeterminada y cómo podemos controlarlo si el comportamiento predeterminado no es adecuado para la aplicación. La consistencia también se refiere al hecho de que la base de datos debe adherirse a todas las reglas de integridad que se han definido dentro de ella mediante restricciones (como claves primarias, restricciones únicas y claves foráneas). La transacción hace que la base de datos pase de un estado coherente a otro.
- **Aislamiento (Isolation):** el aislamiento garantiza que las transacciones solo accedan a datos consistentes. Cada uno controla lo que significa la consistencia para sus transacciones a través de un mecanismo llamado niveles de aislamiento. Con las tablas basadas en disco, SQL Server admite dos modelos diferentes para manejar el aislamiento: uno basado únicamente en el bloqueo y otro basado en una combinación de bloqueo y control de versiones de filas. Para simplificar, nos referiremos a este último sólo como control de versiones de filas. El modelo basado en el bloqueo es el predeterminado en instalaciones en caja. En este modelo, los lectores requieren bloqueos compartidos. Si el estado actual de los datos es inconsistente, los lectores se bloquean hasta que el estado de los datos sea consistente. El modelo basado en el control de versiones de filas es el predeterminado en Azure SQL Database. En este modelo, los lectores no toman bloqueos compartidos y no necesitan esperar. Si el estado actual de los datos es inconsistente, el lector obtiene un estado consistente anterior. La sección "Niveles de aislamiento" más adelante proporciona más detalles sobre ambas formas de manejar el aislamiento.
- **Durabilidad:** La propiedad de durabilidad significa que una vez que el motor de la base de datos reconoce una instrucción de confirmación, se garantiza que los cambios de la transacción serán duraderos, o en otras palabras, persistentes en la base de datos. Se reconoce una confirmación devolviendo el control a la aplicación y ejecutando la siguiente línea de código. El mecanismo que garantiza la propiedad de durabilidad en SQL Server depende de la arquitectura de recuperación que esté en vigor. Existe la arquitectura más tradicional utilizada antes de SQL Server 2019. Hay una arquitectura más nueva que está disponible en SQL Server 2019 o posterior si habilitamos una función llamada Recuperación acelerada de datos (**ADR**) para admitir un proceso de recuperación más rápido. La arquitectura tradicional es más simple e involucra menos componentes. Los cambios de datos siempre se escriben en el registro de transacciones de la base de datos en el disco antes de que se escriban en la parte de datos de la base de datos en el disco. Una vez que la instrucción de confirmación se asienta en el registro de transacciones en el disco, la transacción se considera duradera incluso si el cambio aún no se ha realizado en la porción de datos en el disco. Cuando el sistema se inicia, ya sea normalmente o después de una falla del sistema, SQL Server ejecuta un proceso de recuperación en cada base de datos que implica analizar el registro, luego aplicar una fase de rehacer y luego aplicar una fase de deshacer. La fase de rehacer implica avanzar (reproducir) todos los cambios de cualquier transacción cuya instrucción de confirmación se escriba en el registro, pero cuyos cambios aún no hayan llegado a la parte de datos. La fase de deshacer implica revertir (deshacer) los cambios de cualquier transacción cuya instrucción de confirmación no se asentó en el registro. Si habilita ADR, la arquitectura de recuperación es más sofisticada, involucrando componentes adicionales y un proceso más sofisticado. Puede encontrar detalles sobre ADR en la documentación del producto aquí: <https://learn.microsoft.com/en-us/sql/relational-databases/accelerated-databaserecovery-concepts>. Ya sea que se base en la arquitectura de recuperación tradicional o en la más



nueva, obtendremos la misma garantía de durabilidad de la transacción tras el reconocimiento de su instrucción de confirmación, sólo que con una mecánica diferente detrás de bambalinas.

El siguiente ejemplo actualiza el nombre de contacto del cliente con ID 'ALFKI' a 'Juan Pérez' y la ciudad de envío de todos los pedidos del cliente con ID 'ALFKI' a 'Méjico'. La transacción se inicia con la instrucción BEGIN TRANSACTION, se confirma con la instrucción COMMIT y se deshace con la instrucción ROLLBACK

```
BEGIN TRANSACTION;

UPDATE Customers
SET ContactName = 'Juan Pérez'
WHERE CustomerID = 'ALFKI';

UPDATE Orders
SET ShipCity = 'Méjico'
WHERE CustomerID = 'ALFKI';

COMMIT;
```

En MySQL la sintaxis para el manejo de transacciones es:

```
START TRANSACTION
[TRANSACTION_CHARACTERISTIC [, TRANSACTION_CHARACTERISTIC] ...]

TRANSACTION_CHARACTERISTIC: {
    WITH CONSISTENT SNAPSHOT
    | READ WRITE
    | READ ONLY
}

BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET autocommit = {0 | 1}
```

Estas declaraciones proporcionan control sobre el uso de transacciones:

- START TRANSACTION o BEGIN iniciar una nueva transacción.
- COMMIT confirma la transacción actual, haciendo que sus cambios sean permanentes.
- ROLLBACK revierte la transacción actual, cancelando sus cambios.
- SET autocommit deshabilita o habilita el modo de autocommit predeterminado para la sesión actual.

De forma predeterminada, MySQL se ejecuta con el modo de confirmación automática habilitado. Esto significa que, cuando no está dentro de una transacción, cada sentencia es atómica, como si estuviera rodeada por START TRANSACTION y COMMIT. No puede usar ROLLBACK para deshacer el efecto; sin embargo, si ocurre un error durante la ejecución de la sentencia, la sentencia se revierte.

CURSORES

Una consulta que no utiliza *ORDER BY* devuelve un conjunto (o un multiset), mientras que una consulta con *ORDER BY* devuelve lo que el SQL estándar llama *cursor* –una estructura no relacional con el orden garantizado entre sus filas.

SQL y T-SQL también admiten un objeto llamado *cursor* que podemos utilizar para procesar las filas resultantes de una consulta de a una a la vez y en el orden solicitado. Esto contrasta con el uso de consultas basadas en conjuntos: consultas normales sin un cursor para las cuales manipula el conjunto o multiset y no puede confiar en el orden.



Quiero enfatizar que la elección predeterminada debe ser utilizar consultas basadas en conjuntos; solo cuando tengamos una razón convincente para hacer lo contrario, debemos considerar el uso de cursos. Esta recomendación se basa en varios factores, incluidos los siguientes:

- En primer lugar, cuando utilizamos cursos, vamos en contra del modelo relacional, que se basa en la teoría de conjuntos.
- La manipulación registro por registro realizada por el cursor tiene sobrecarga. Un cierto costo adicional está asociado con cada manipulación de registros por el cursor en comparación con la manipulación basada en conjuntos. Dada una consulta basada en un conjunto y un código de cursor que realiza un procesamiento físico similar detrás de las escenas, el código del cursor suele ser mucho más lento que el código basado en el conjunto.
- Con los cursos, escribimos soluciones imperativas; en otras palabras, somos responsables de definir cómo procesar los datos (declarar el cursor, abrirlo, recorrer los registros del cursor, cerrar el cursor y desasignar el cursor). Con las soluciones basadas en conjuntos, escribimos un código declarativo donde se enfoca principalmente en los aspectos lógicos de la solución; en otras palabras, en qué obtener en lugar de como obtenerlo. Por lo tanto, las soluciones de cursor tienden a ser más largas, menos legibles y más difíciles de mantener que las soluciones basadas en conjuntos.

Para la mayoría de las personas, no es sencillo pensar en términos relativos inmediatamente cuando comienzan a aprender SQL. Es más intuitivo para la mayoría de las personas pensar en términos de cursos: procesar un registro a la vez en un cierto orden. Como resultado, los cursos se utilizan ampliamente y, en la mayoría de los casos, se utilizan incorrectamente; es decir, se utilizan incluso cuando existen soluciones mucho mejores basadas en conjuntos. Hagamos un esfuerzo consciente para adoptar el estado mental basado en conjuntos y para pensar verdaderamente en términos de conjuntos. Puede llevar tiempo, en algunos casos años, pero mientras trabajemos con un lenguaje basado en el modelo relacional, esa es la manera correcta de pensar.

Cada regla tiene excepciones. Un ejemplo es cuando necesitamos aplicar una determinada tarea a cada fila de alguna tabla o vista. Por ejemplo, puede que necesite ejecutar alguna tarea administrativa para cada índice o tabla en su base de datos. En tal caso, tiene sentido usar un cursor para iterar a través de los nombres de índice o tabla uno a la vez y ejecutar la tarea relevante para cada uno de ellos.

Otro ejemplo de cuándo debería considerar los cursos es cuando su solución basada en conjuntos funciona mal y agotamos esfuerzos de ajuste utilizando el enfoque basado en conjuntos. Como se mencionó, las soluciones basadas en conjuntos tienden a ser mucho más rápidas, pero en algunos casos excepcionales, la solución de cursor es más rápida. Un ejemplo de este tipo es el cálculo de agregados en ejecución utilizando el código T-SQL que es compatible con versiones legacy de SQL Server que no admiten la opción de marco en las funciones de ventana. Las soluciones relativas para ejecutar agregados utilizando uniones o subconsultas son extremadamente lentas. Una solución iterativa, como una basada en un cursor, suele ser la óptima. Si no hay restricciones de compatibilidad, utilizar una solución relacional con funciones de ventana es la forma óptima de calcular los totales acumulados.

Trabajar con un cursor generalmente implica los siguientes pasos:

1. Declare el cursor basado en una consulta.
2. Abra el cursor.
3. Obtenga los valores de los atributos del primer registro de cursor en variables.
4. Mientras no hayamos llegado al final del cursor (mientras que el valor de una función llamada `@@FETCH_STATUS` es 0), recorremos los registros del cursor; en cada iteración del bucle, realizamos el



procesamiento necesario para la fila actual y luego obtenemos los valores de los atributos de la siguiente fila en las variables.

5. Cierre el cursor.
6. Desasignar el cursor.

El siguiente ejemplo con código de cursor recorremos la tabla Products y mostramos el nombre y precio formateado:

```

DECLARE
    @nombre VARCHAR(MAX),
    @precio MONEY;

DECLARE cursor_productos CURSOR
FOR SELECT
    ProductName,
    UnitPrice
FROM
    Products;

OPEN cursor_productos;

FETCH NEXT FROM cursor_productos INTO
    @nombre,
    @precio;

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CONCAT(@nombre, ' ', FORMAT(@precio, 'C', 'es-ar')) ;
    FETCH NEXT FROM cursor_productos INTO
        @nombre,
        @precio;
END;

CLOSE cursor_productos;

DEALLOCATE cursor_productos;

```

MySQL admite cursores dentro de stored procedures. Los cursores tienen estas propiedades:

- Insensible: el servidor puede o no hacer una copia de su tabla de resultados
- Solo lectura: no actualizable
- No desplazable: solo se puede recorrer en una dirección y no se pueden saltar filas

Las declaraciones de cursor deben aparecer antes de las declaraciones de controlador y después de las declaraciones de variables y condiciones.

Ejemplo:



```

CREATE PROCEDURE curdemo()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE a CHAR(16);
    DECLARE b, c INT;
    DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
    DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    OPEN cur1;
    OPEN cur2;

    read_loop: LOOP
        FETCH cur1 INTO a, b;
        FETCH cur2 INTO c;
        IF done THEN
            LEAVE read_loop;
        END IF;
        IF b < c THEN
            INSERT INTO test.t3 VALUES (a,b);
        ELSE
            INSERT INTO test.t3 VALUES (a,c);
        END IF;
    END LOOP;

    CLOSE cur1;
    CLOSE cur2;
END;

```

TABLAS TEMPORALES

Cuando necesitamos almacenar temporalmente datos en tablas, en ciertos casos podemos preferir no trabajar con tablas permanentes.

Otro caso en el que solemos usar tablas temporales es cuando no tenemos permisos para crear tablas permanentes en una base de datos de usuarios.

SQL Server admite tres tipos de tablas temporales con las que puede ser más conveniente trabajar que las tablas permanentes en tales casos: tablas temporales locales, tablas temporales globales y variables de tabla. Las siguientes secciones describen los tres tipos y demuestran su uso con ejemplos de código.

TABLAS TEMPORALES LOCALES

Para crear una tabla temporal local, la nombramos con único numeral como prefijo, como #T1. Los tres tipos de tablas temporales se crean en la base de datos *tempdb*.

Una tabla temporal local solo es visible para la sesión que la creó, en el nivel de creación y en todos los niveles internos de la pila de llamadas (procedimientos internos, activadores y lotes dinámicos). SQL Server destruye automáticamente una tabla temporal local cuando el nivel de creación en la pila de llamadas queda fuera del alcance. Por ejemplo, supongamos que un procedimiento almacenado llamado *Proc1* llama a un procedimiento llamado *Proc2*, que a su vez llama a un procedimiento llamado *Proc3*, que a su vez llama a un procedimiento llamado *Proc4*. *Proc2* crea una tabla temporal llamada #T1 antes de llamar a *Proc3*. La tabla #T1 es visible para *Proc2*, *Proc3* y *Proc4*, pero no para *Proc1*, y SQL Server la destruye automáticamente cuando termina *Proc2*. Si la tabla temporal se crea en un lote ad hoc en el nivel de anidación más externo de la sesión (en otras palabras, cuando el valor de la función @@NESTLEVEL es 0),



también es visible para todos los lotes posteriores y se destruye por SQL Server automáticamente solo cuando la sesión de creación se desconecta.

Podría preguntarse cómo SQL Server evita conflictos de nombres cuando dos sesiones crean tablas temporales locales con el mismo nombre. SQL Server agrega internamente un sufijo al nombre de la tabla que lo hace único en *tempdb*. Como desarrollador, no debería importarte: te refieres a la tabla utilizando el nombre que proporcionaste sin el sufijo interno, y solo tu sesión tiene acceso a tu tabla.

Un escenario obvio para el cual las tablas temporales locales son útiles es cuando tiene un proceso que necesita almacenar resultados intermedios temporalmente, como durante un ciclo, y luego consultar los datos.

Otro escenario es cuando necesita acceder al resultado de un procesamiento costoso varias veces. Por ejemplo, supongamos que necesitamos unir las tablas *Orders* y *Order Details*, sumar las cantidades de pedido por año de pedido y unir dos instancias de los datos agregados para comparar la cantidad total de cada año con el año anterior. Las tablas *Orders* y *Order Details* en la base de datos muestra son muy pequeñas, pero en situaciones reales, estas tablas pueden tener millones de filas. Tiene sentido hacer todo el trabajo costoso solo una vez (almacenar el resultado en una tabla temporal local) y luego unir dos instancias de la tabla temporal, especialmente porque el resultado del trabajo costoso es un conjunto pequeño con solo una fila por año de orden.

El siguiente código ilustra este escenario utilizando una tabla temporal local:

```
DROP TABLE IF EXISTS #MiTotalOrdenesPorAnio;
GO

CREATE TABLE #MiTotalOrdenesPorAnio (
ordenanio INT NOT NULL PRIMARY KEY,
cantidad INT NOT NULL
);

INSERT INTO #MiTotalOrdenesPorAnio(ordenanio, cantidad)
SELECT YEAR(O.OrderDate) AS ordenanio, SUM(DO.Quantity) AS cantidad
FROM Orders AS O
INNER JOIN [Order Details] AS DO ON DO.OrderID = O.OrderID
GROUP BY YEAR(OrderDate);

SELECT Act.ordenanio, Act.cantidad AS cantidadanioact, Prv.cantidad AS cantidadaniopriv
FROM #MiTotalOrdenesPorAnio AS Act
LEFT OUTER JOIN #MiTotalOrdenesPorAnio AS Prv ON Act.ordenanio = Prv.ordenanio + 1;
```

Este código genera la siguiente salida:

ordenanio	cantidadanioact	cantidadaniopriv
1996	9581	NULL
1997	25489	9581
1998	16247	25489

Para verificar que la tabla local temporal es visible solo en la sesión que la crea, intenten acceder a la tabla desde otra sesión:

```
SELECT ordenanio, cantidad
FROM #MiTotalOrdenesPorAnio;
```



Obtendremos el siguiente error:

```
Msg 208, Level 16, State 0, Line 1
Invalid object name '#MiTotalOrdenesPorAnio'.
```

Al finalizar, vuelvan a la sesión original y eliminén la tabla temporal:

```
DROP TABLE IF EXISTS #MiTotalOrdenesPorAnio;
```

Generalmente se recomienda limpiar los recursos tan pronto como haya terminado de trabajar con ellos.

TABLAS TEMPORALES GLOBALES

Cuando creamos una tabla temporal global, es visible para todas las demás sesiones. SQL Server destruye automáticamente las tablas temporales globales cuando la sesión de creación se desconecta y no hay referencias activas a la tabla. Creamos una tabla temporal global al nombrarla con dos numerales como un prefijo, como ##T1.

Las tablas temporales globales son útiles cuando desea compartir datos temporales con todos. No se requieren permisos especiales, y todos tienen acceso total a DDL y DML. Por supuesto, el hecho de que todos tengan acceso completo significa que cualquiera puede cambiar o incluso eliminar la tabla, así que considere las alternativas con cuidado.

Por ejemplo, el siguiente código crea una tabla temporal global llamada ##Globales con columnas llamadas id y val:

```
CREATE TABLE ##Globales (
    id sysname NOT NULL PRIMARY KEY,
    val SQL_VARIANT NOT NULL
);
```

La tabla en este ejemplo está diseñada para imitar variables globales, que no son compatibles con T-SQL. La columna id es de un tipo de datos sysname (el tipo que SQL Server utiliza internamente para representar identificadores), y la columna val es de un tipo de datos SQL_VARIANT (un tipo genérico que puede almacenar dentro de él un valor de casi cualquier tipo de datos).

Cualquiera puede insertar filas en la tabla. Por ejemplo, ejecute el siguiente código para insertar una fila que represente una variable llamada i e inicialícela con el valor entero 10:

```
INSERT INTO ##Globales(id, val) VALUES(N'i', CAST(10 AS INT));
```

Cualquiera puede modificar y consultar datos de la tabla. Por ejemplo, correr el siguiente código desde cualquier sesión para consultar el valor actual de la variable i:

```
SELECT val FROM ##Globales WHERE id = N'i';
```

Este código devuelve la siguiente salida:

```
val
```

```
-----
```

```
10
```



Tengan en cuenta que tan pronto como la sesión que creó la tabla temporal global se desconecta y no hay referencias activas a la tabla, SQL Server destruye la tabla automáticamente.

Si desea que se cree una tabla temporal global cada vez que se inicie SQL Server y no quiere que SQL Server intente destruirlo automáticamente, debe crear la tabla a partir de un procedimiento almacenado que esté marcado como un procedimiento de inicio. (Para más detalles, puede ver “[sp_procoption](#)” en los libros en línea de SQL Server en la siguiente dirección: [sp_procoption \(Transact-SQL\) - SQL Server | Microsoft Docs](#)).

Corra el siguiente código desde cualquier sesión para destruir explícitamente la tabla temporal global:

```
DROP TABLE IF EXISTS ##Globales;
```

VARIABLES DE TABLA

Las variables de tabla son similares a las tablas temporales locales en algunos aspectos y diferentes en otros. Declaramos las variables de tabla de manera muy similar a como declaramos otras variables, utilizando la instrucción **DECLARE**.

Al igual que con las tablas temporales locales, las variables de tabla tienen una presencia física como una tabla en la base de datos *tempdb*, contrariamente a la idea errónea de que existen solo en la memoria. Al igual que las tablas temporales locales, las variables de la tabla son visibles solo para la sesión de creación, pero como son variables, tienen un alcance más limitado: solo el lote actual. Las variables de tabla no son visibles para los lotes internos en la pila de llamadas ni para los lotes posteriores en la sesión.

Si una transacción explícita se revierte, los cambios realizados en las tablas temporales de esa transacción también se retrotraen; sin embargo, los cambios realizados en las variables de tabla por las declaraciones que se completaron en la transacción no se retrotraen. Sólo se deshacen los cambios realizados por la declaración activa que falló o que se terminó antes de completarse.

Las tablas temporales y las variables de tabla también tienen diferencias de optimización, pero esos temas están fuera del alcance de la materia. Por ahora, solo diremos que, en términos de rendimiento, generalmente tiene más sentido usar variables de tabla con pequeños volúmenes de datos (solo unas pocas filas) y usar tablas temporales locales en el resto de los casos.

Por ejemplo, el siguiente código utiliza una variable de tabla en lugar de una tabla temporal local para comparar las cantidades totales de pedidos de cada año de pedido con el año anterior:



```

DECLARE @MiTotalOrdenesPorAnio TABLE (
    anioorden INT NOT NULL PRIMARY KEY,
    cantidad    INT NOT NULL);

INSERT INTO @MiTotalOrdenesPorAnio(anioorden, cantidad)

SELECT
    YEAR(O.OrderDate) AS orderyear, SUM(DO.Quantity) AS cantidad
    FROM Orders AS O
    INNER JOIN [Order Details] AS DO ON DO.OrderID = O.OrderID
    GROUP BY YEAR(OrderDate);

SELECT Act.anioorden, Act.cantidad AS actaniocant, Prv.cantidad AS prvaniocant
    FROM @MiTotalOrdenesPorAnio AS Act
    LEFT OUTER JOIN @MiTotalOrdenesPorAnio AS Prv
    ON Act.anioorden = Prv.anioorden + 1;

```

Este código devuelve la siguiente salida:

anioorden	actaniocant	prvaniocant
1996	9581	NULL
1997	25489	9581
1998	16247	25489

TABLAS TEMPORALES EN MYSQL

Puede utilizar la palabra clave TEMPORARY al crear una tabla. Una tabla TEMPORAL solo es visible dentro de la sesión actual y se elimina automáticamente cuando se cierra la sesión. Esto significa que dos sesiones diferentes pueden usar el mismo nombre de tabla temporal sin entrar en conflicto entre sí o con una tabla no TEMPORAL existente del mismo nombre. (La tabla existente está oculta hasta que se elimina la tabla temporal).

InnoDB no admite tablas temporales comprimidas. Cuando innodb_strict_mode está habilitado (el valor predeterminado), CREATE TEMPORARY TABLE devuelve un error si se especifica ROW_FORMAT=COMPRESSED o KEY_BLOCK_SIZE. Si innodb_strict_mode está deshabilitado, se emiten advertencias y la tabla temporal se crea utilizando un formato de fila no comprimido. La opción innodb_file_per_table no afecta la creación de tablas temporales de InnoDB.

Las tablas TEMPORALES tienen una relación muy flexible con las bases de datos (esquemas). Al descartar una base de datos, no se eliminan automáticamente las tablas TEMPORALES creadas dentro de esa base de datos.

Para crear una tabla temporal, debe tener el privilegio CREATE TEMPORARY TABLES. Después de que una sesión haya creado una tabla temporal, el servidor no realiza más comprobaciones de privilegios en la tabla. La sesión de creación puede realizar cualquier operación en la tabla, como DROP TABLE, INSERT, UPDATE o SELECT.

No puede usar CREATE TEMPORARY TABLE... LIKE para crear una tabla vacía basada en la definición de una tabla que reside en el espacio de tabla mysql, el espacio de tabla del sistema InnoDB (innodb_system) o un espacio de tabla general. La definición de tablespace para una tabla de este tipo incluye un atributo TABLESPACE que define el tablespace donde reside la tabla, y los tablespaces antes mencionados no admiten tablas temporales. Para crear una tabla temporal basada en la definición de dicha tabla, use esta sintaxis en su lugar:

```
CREATE TEMPORARY TABLE new_tbl SELECT * FROM orig_tbl LIMIT 0;
```



MANEJO DE ERRORES EN SQL SERVER

SQL Server proporciona herramientas para manejar errores en el código T-SQL. La principal herramienta utilizada para el manejo de errores es una construcción llamada TRY...CATCH. SQL Server también proporciona un conjunto de funciones que podemos invocar para obtener información sobre el error. Comenzaremos con un ejemplo básico que demuestra el uso de TRY...CATCH, seguido de un ejemplo más detallado que demuestra el uso de las funciones de error.

Trabajamos con la construcción TRY...CATCH colocando el código T-SQL habitual en un bloque TRY (entre las palabras clave BEGIN TRY y END TRY) y colocando todo el código de manejo de errores en el bloque CATCH adyacente (entre las palabras clave BEGIN CATCH y END CATCH). Si el bloque TRY no tiene ningún error, el bloque CATCH simplemente se omite. Si el bloque TRY tiene un error, el control pasa al bloque CATCH correspondiente. Tengan en cuenta que si un bloque TRY...CATCH captura y maneja un error, en lo que respecta al usuario que hace la llamada, no hubo error.

Ejecuten el siguiente código para demostrar un caso sin error en el bloque TRY:

```
BEGIN TRY
PRINT 10/2;
PRINT 'No error';
END TRY
BEGIN CATCH
PRINT 'Error';
END CATCH;
```

Todo el código en el bloque TRY se completó con éxito; por lo tanto, se omitió el bloque CATCH. Este código genera el siguiente resultado:

```
5
No error
```

A continuación, ejecuten un código similar, pero esta vez dividan por cero. Se produce un error:

```
BEGIN TRY
PRINT 10/0;
PRINT 'No error';
END TRY
BEGIN CATCH
PRINT 'Error';
END CATCH;
```

Cuando ocurrió el error de división por cero en la primera instrucción PRINT en el bloque TRY, el control se pasó al bloque CATCH correspondiente. No se ejecutó la segunda instrucción PRINT en el bloque TRY. Por lo tanto, este código genera el siguiente resultado:

```
Error
```

Por lo general, el manejo de errores implica algún trabajo en el bloque CATCH que investiga la causa del error y toma un curso de acción. SQL Server brinda información sobre el error a través de un conjunto de funciones. La función ERROR_NUMBER devuelve un número entero con el número del error.

El bloque CATCH generalmente incluye código de flujo que inspecciona el número de error para determinar qué curso de acción tomar. La función ERROR_MESSAGE devuelve el texto del mensaje de error. Para obtener la lista de números



y mensajes de error, pueden consultar la vista de catálogo sys.messages. Las funciones ERROR_SEVERITY y ERROR_STATE devuelven la gravedad y el estado del error.

La función ERROR_LINE devuelve el número de línea en el código donde ocurrió el error. Finalmente, la función ERROR_PROCEDURE devuelve el nombre del procedimiento en el que ocurrió el error y devuelve NULL si el error no ocurrió dentro de un procedimiento.

Para demostrar un ejemplo más detallado de manejo de errores, incluido el uso de las funciones de error, primero ejecuten el siguiente código, que crea una tabla llamada dbo.Employees en la base de datos actual (no utilicen Northwind, ya borrarán la tabla Employees):

```
DROP TABLE IF EXISTS dbo.Employees;
CREATE TABLE dbo.Employees
(
empid INT NOT NULL,
empname VARCHAR(25) NOT NULL,
mgrid INT NULL,
CONSTRAINT PK_Employees PRIMARY KEY(empid),
CONSTRAINT CHK_Employees_empid CHECK(empid > 0),
CONSTRAINT FK_Employees_Employees
FOREIGN KEY(mgrid) REFERENCES dbo.Employees(empid)
);
```

El siguiente código inserta una nueva fila en la tabla Empleados en un bloque TRY y, si ocurre un error, muestra cómo identificar el error al inspeccionar la función ERROR_NUMBER en el bloque CATCH. El código utiliza el control de flujo para identificar y manejar los errores que desea tratar en el bloque CATCH y, de lo contrario, vuelve a generar el error.

El código también imprime los valores de las otras funciones de error simplemente para mostrar qué información está disponible cuando ocurre un error:



```

BEGIN TRY
    INSERT INTO dbo.Employees(empid, empname, mgrid)
    VALUES(1, 'Emp1', NULL);
    -- Intentar tambien con empid = 0, 'A', NULL
END TRY
BEGIN CATCH
    IF ERROR_NUMBER() = 2627
    BEGIN
        PRINT ' Manejo de violacion de PK...';
    END;
    ELSE IF ERROR_NUMBER() = 547
    BEGIN
        PRINT ' Manejo de violacion de restriccion CHECK/FK...';
    END;
    ELSE IF ERROR_NUMBER() = 515
    BEGIN
        PRINT ' Manejo de violacion NULL...';
    END;
    ELSE IF ERROR_NUMBER() = 245
    BEGIN
        PRINT ' Manejo de error de conversion...';
    END;
    ELSE
    BEGIN
        PRINT 'Volver a lanzar el error...';
        THROW;
    END;

    PRINT ' Error Number : ' + CAST(ERROR_NUMBER() AS VARCHAR(10));
    PRINT ' Error Message : ' + ERROR_MESSAGE();
    PRINT ' Error Severity: ' + CAST(ERROR_SEVERITY() AS VARCHAR(10));
    PRINT ' Error State : ' + CAST(ERROR_STATE() AS VARCHAR(10));
    PRINT ' Error Line : ' + CAST(ERROR_LINE() AS VARCHAR(10));
    PRINT ' Error Proc : ' + COALESCE(ERROR_PROCEDURE(), 'No dentro de un proc');
END CATCH;

```

Cuando ejecutan este código por primera vez, la nueva fila se inserta correctamente en la tabla Empleados y, por lo tanto, se omite el bloque CATCH. Obtienen el siguiente resultado:

(1 rows affected)

Cuando ejecutan el mismo código por segunda vez, la declaración INSERT falla, el control pasa al bloque CATCH y se identifica un error de violación de clave primaria. Obtienen el siguiente resultado:

```

(0 rows affected)
Manejo de violacion de PK...
Error Number : 2627
Error Message : Violation of PRIMARY KEY constraint 'PK_Employees'. Cannot insert duplicate key in object 'dbo.Employees'. The duplicate key value is (1).
Error Severity: 14
Error State : 1
Error Line : 2
Error Proc : No dentro de un proc

```

Para ver otros errores, ejecuten el código con los valores 0, 'A' y NULL como ID de empleado.



Aquí, con fines de demostración, utilizamos instrucciones PRINT como acciones cuando se identificó un error. Por supuesto, el manejo de errores generalmente involucra más que simplemente imprimir un mensaje que indica que se identificó el error.

Tengan en cuenta que pueden crear un procedimiento almacenado que encapsule un código de manejo de errores reutilizable como este:

```

CREATE OR ALTER PROC dbo.ErrInsertHandler
AS
SET NOCOUNT ON;
    IF ERROR_NUMBER() = 2627
    BEGIN
        PRINT ' Manejo de violacion de PK...';
    END;
    ELSE IF ERROR_NUMBER() = 547
    BEGIN
        PRINT ' Manejo de violacion de restriccion CHECK/FK...';
    END;
    ELSE IF ERROR_NUMBER() = 515
    BEGIN
        PRINT ' Manejo de violacion NULL...';
    END;
    ELSE IF ERROR_NUMBER() = 245
    BEGIN
        PRINT ' Manejo de error de conversion...';
    END;

    PRINT ' Error Number : ' + CAST(ERROR_NUMBER() AS VARCHAR(10));
    PRINT ' Error Message : ' + ERROR_MESSAGE();
    PRINT ' Error Severity: ' + CAST(ERROR_SEVERITY() AS VARCHAR(10));
    PRINT ' Error State : ' + CAST(ERROR_STATE() AS VARCHAR(10));
    PRINT ' Error Line : ' + CAST(ERROR_LINE() AS VARCHAR(10));
    PRINT ' Error Proc : ' + COALESCE(ERROR_PROCEDURE(), 'No dentro de un proc');

GO

```

En el bloque CATCH, verificamos si el número de error es uno de los que deseamos tratar localmente. Si es así, simplemente ejecutamos el procedimiento almacenado; de lo contrario, volvemos a arrojar el error:

```

BEGIN TRY
INSERT INTO dbo.Employees(empid, empname, mgrid)
VALUES(1, 'Emp1', NULL);
END TRY
BEGIN CATCH
    IF ERROR_NUMBER() IN (2627, 547, 515, 245)
        EXEC dbo.ErrInsertHandler;
    ELSE
        THROW;
END CATCH;

```

De esta manera, podemos mantener el código de manejo de errores reutilizable en un solo lugar.



MANEJO DE ERRORES EN MYSQL

En MySQL no existe el bloque TRY CATCH. Para el manejo de errores se utilizan los handlers.

```
DECLARE handler_action HANDLER
FOR condition_value [, condition_value] ...
statement

handler_action: {
    CONTINUE
    EXIT
    UNDO
}

condition_value: {
    mysql_error_code
    SQLSTATE [VALUE] sqlstate_value
    condition_name
    SQLWARNING
    NOT FOUND
    SQLEXCEPTION
}
```

La declaración DECLARE ... HANDLER especifica un controlador que se ocupa de una o más condiciones. Si ocurre una de estas condiciones, se ejecuta la instrucción especificada. puede ser una declaración simple como SET var_name = valor, o una declaración compuesta escrita usando BEGIN y END.

Las declaraciones de controlador deben aparecer después de las declaraciones de variables o condiciones.

El valor handler_action indica qué acción toma el controlador después de la ejecución de la declaración del controlador:

- CONTINUE: continúa la ejecución del programa actual.
- EXIT: la ejecución finaliza para la declaración compuesta BEGIN ... END en la que se declara el controlador. Esto es cierto incluso si la condición ocurre en un bloque interno.
- UNDO: no compatible.

condition_value para DECLARE... HANDLER indica la condición específica o la clase de condiciones que activa el controlador. Puede tomar las siguientes formas:

- mysql_error_code: un literal entero que indica un código de error de MySQL, como 1051 para especificar "tabla desconocida":

```
DECLARE CONTINUE HANDLER FOR 1051
BEGIN
    -- body of handler
END;
```

No utilice el código de error 0 de MySQL porque indica éxito en lugar de una condición de error. Para obtener una lista de los códigos de error de MySQL, consulte Referencia de mensajes de error del servidor.

- SQLSTATE [VALOR] sqlstate_value: un literal de cadena de 5 caracteres que indica un valor de SQLSTATE, como '42S01' para especificar "tabla desconocida":

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'
BEGIN
    -- body of handler
END;
```



No utilice valores de SQLSTATE que empiecen por '00' porque indican éxito en lugar de una condición de error. Para obtener una lista de valores de SQLSTATE, consulte Referencia de mensajes de error del servidor.

- condition_name: Un nombre de condición previamente especificado con DECLARE ... CONDITION. Un nombre de condición se puede asociar con un código de error de MySQL o un valor de SQLSTATE.
- SQLWARNING: abreviatura de la clase de valores SQLSTATE que comienzan con '01'.

```
DECLARE CONTINUE HANDLER FOR SQLWARNING
BEGIN
    -- body of handler
END;
```

- NOT FOUND: abreviatura de la clase de valores SQLSTATE que comienzan con '02'. Esto es relevante dentro del contexto de los cursos y se usa para controlar lo que sucede cuando un cursor llega al final de un conjunto de datos. Si no hay más filas disponibles, se produce una condición Sin datos con el valor de SQLSTATE '02000'. Para detectar esta condición, puede configurar un controlador para ella o para una condición NOT FOUND.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND
BEGIN
    -- body of handler
END;
```

- SQLEXCEPTION: abreviatura de la clase de valores SQLSTATE que no comienzan con '00', '01' o '02'.

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
BEGIN
    -- body of handler
END;
```

Si se produce una condición para la que no se ha declarado ningún controlador, la acción que se emprende depende de la clase de condición:

- Para las condiciones SQLEXCEPTION, el stored procedure finaliza en la instrucción que generó la condición, como si hubiera un controlador EXIT. Si el programa fue llamado por otro stored procedure, el programa que llama maneja la condición usando las reglas de selección de manejadores aplicadas a sus propios manejadores.
- Para las condiciones de SQLWARNING, el programa continúa ejecutándose, como si hubiera un controlador CONTINUE.
- Para las condiciones NOT FOUND, si la condición se generó normalmente, la acción es CONTINUE. Si fue planteada por SIGNAL o RESIGNAL, la acción es EXIT.

El siguiente ejemplo utiliza un controlador para SQLSTATE '23000', que se produce por un error de clave duplicada:



```

mysql> CREATE TABLE test.t (s1 INT, PRIMARY KEY (s1));
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter //
mysql> CREATE PROCEDURE handlertdemo ()
BEGIN
    DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1;
    SET @x = 1;
    INSERT INTO test.t VALUES (1);
    SET @x = 2;
    INSERT INTO test.t VALUES (1);
    SET @x = 3;
END;
//
Query OK, 0 rows affected (0.00 sec)

mysql> CALL handlertdemo()//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x//
+-----+
| @x   |
+-----+
| 3    |
+-----+
1 row in set (0.00 sec)

```

Observe que @x es 3 después de que se ejecuta el procedimiento, lo que muestra que la ejecución continuó hasta el final del procedimiento después de que ocurrió el error. Si la declaración DECLARE ... HANDLER no hubiera estado presente, MySQL habría tomado la acción predeterminada (EXIT) después de que fallara el segundo INSERT debido a la restricción PRIMARY KEY, y SELECT @x habría devuelto 2.

Para ignorar una condición, declare un controlador CONTINUE para ella y asócielo con un bloque vacío. Por ejemplo:

```
DECLARE CONTINUE HANDLER FOR SQLWARNING BEGIN END;
```

El alcance de una etiqueta de bloque no incluye el código de los controladores declarados dentro del bloque. Por lo tanto, la declaración asociada con un controlador no puede usar ITERATE o LEAVE para hacer referencia a las etiquetas de los bloques que encierran la declaración del controlador. Considere el siguiente ejemplo, donde el bloque REPEAT tiene una etiqueta de reintento:

```

CREATE PROCEDURE p ()
BEGIN
    DECLARE i INT DEFAULT 3;
    retry:
    REPEAT
        BEGIN
            DECLARE CONTINUE HANDLER FOR SQLWARNING
            BEGIN
                ITERATE retry;      # illegal
            END;
            IF i < 0 THEN
                LEAVE retry;      # legal
            END IF;
            SET i = i - 1;
        END;
    UNTIL FALSE END REPEAT;
END;

```



La etiqueta de retry está dentro del alcance de la declaración IF dentro del bloque. No está dentro del alcance del controlador CONTINUE, por lo que la referencia allí no es válida y genera un error:

```
ERROR 1308 (42000): LEAVE with no matching label: retry
```

Para evitar referencias a etiquetas externas en controladores, utilice una de estas estrategias:

- Para salir del bloque, use un controlador EXIT. Si no se requiere limpieza de bloques, el cuerpo del controlador BEGIN ... END puede estar vacío:

```
DECLARE EXIT HANDLER FOR SQLWARNING BEGIN END;
```

De lo contrario, coloque las declaraciones de limpieza en el cuerpo del controlador:

```
DECLARE EXIT HANDLER FOR SQLWARNING
BEGIN
    block cleanup statements
END;
```

- Para continuar con la ejecución, establezca una variable de estado en un controlador CONTINUE que se pueda verificar en el bloque adjunto para determinar si se invocó el controlador. El siguiente ejemplo usa la variable done para este propósito:

```
CREATE PROCEDURE p ()
BEGIN
    DECLARE i INT DEFAULT 3;
    DECLARE done INT DEFAULT FALSE;
    retry:
    REPEAT
        BEGIN
            DECLARE CONTINUE HANDLER FOR SQLWARNING
            BEGIN
                SET done = TRUE;
            END;
            IF done OR i < 0 THEN
                LEAVE retry;
            END IF;
            SET i = i - 1;
        END;
        UNTIL FALSE END REPEAT;
    END;
```



ANEXO OPERADORES Y FUNCIONES

CONCATENACIÓN DE CADENAS (OPERADOR + Y FUNCIÓN CONCAT)

T-SQL proporciona el operador de signo más (+) y la función CONCAT y CONCAT_WS para concatenar cadenas. En MySQL no podemos utilizar el operador signo más (+) para realizar concatenaciones, porque queda reservado a operaciones aritméticas.

SQL estándar dicta que una concatenación con un NULL debe producir un valor NULL. Este es el comportamiento predeterminado de T-SQL con respecto al operador (+), pero en T-SQL la función CONCAT() convierte de manera implícita los NULL en cadenas vacías. En cambio en MySQL la función CONCAT() funciona como dicta el estándar.

Para tratar un NULL como una cadena vacía -o con mayor precisión, para sustituir un NULL por una cadena vacía- podemos utilizar funciones, como COALESCE. Esta función acepta una lista de valores de entrada y devuelve la primera que no es NULL.

Por ejemplo, para obtener el nombre completo de los empleados:

```
SELECT EmployeeID, FirstName + ' ' + LastName AS NombreCompleto
FROM Employees;
```

SQL Server	
EmployeeID	NombreCompleto
1	Nancy Davolio
2	Andrew Fuller
3	Janet Leverling
4	Margaret Peacock
5	Steven Buchanan
6	Michael Suyama
7	Robert King
8	Laura Callahan
9	Anne Dodsworth

MySQL	
EmployeeID	NombreCompleto
1	0.0
2	0.0
3	0.0
4	0.0
5	0.0
6	0.0
7	0.0
8	0.0
9	0.0

Para lograr el mismo resultado en MySQL tenemos que usar CONCAT(), y en SQL Server podemos usar la misma consulta:

```
SELECT EmployeeID, CONCAT(FirstName, ' ', LastName) AS NombreCompleto
FROM Employees;
```

Concatenar una cadena con NULL da como resultado NULL, como podemos ver en el siguiente ejemplo:

SQL Server	
CustomerID	Country, Region, City, Country + ',' + Region + ',' + City AS Ubicacion
1	United States, California, San Francisco, United States, California, San Francisco

MySQL	
CustomerID	Country, Region, City, CONCAT(Country, ',', Region, ',', City) AS Ubicacion
1	United States, California, San Francisco, United States, California, San Francisco



CustomerID	Country	Region	City	Ubicacion
ALFKI	Germany	NULL	Berlin	NULL
ANATR	Mexico	NULL	México D.F.	NULL
ANTON	Mexico	NULL	México D.F.	NULL
AROUT	UK	NULL	London	NULL
...				
WELLI	Brazil	SP	Resende	Brazil,SP,Resende
WHITC	USA	WA	Seattle	USA,WA,Seattle
WILMK	Finland	NULL	Helsinki	NULL
WOLZA	Poland	NULL	Warszawa	NULL

Para trabajar en la concatenación con atributos que pueden tener valor NULL se puede utilizar la función COALESCE, mediante la cual se termina convirtiendo a NULL en una cadena vacía.

SQL Server

```
SELECT CustomerID, Country, Region, City,
Country + COALESCE( ',' + Region, '') + ','
+ City AS Ubicacion
FROM Customers;
```

MySQL

```
SELECT CustomerID, Country, Region, City,
CONCAT(Country, COALESCE( CONCAT(',',
Region), '' ), ',', City) AS Ubicacion
FROM Customers;
```

CustomerID	Country	Region	City	Ubicacion
ALFKI	Germany	NULL	Berlin	Germany,Berlin
ANATR	Mexico	NULL	México D.F.	Mexico,México D.F.
ANTON	Mexico	NULL	México D.F.	Mexico,México D.F.
AROUT	UK	NULL	London	UK,London
...				
WELLI	Brazil	SP	Resende	Brazil,SP,Resende
WHITC	USA	WA	Seattle	USA,WA,Seattle
WILMK	Finland	NULL	Helsinki	Finland,Helsinki
WOLZA	Poland	NULL	Warszawa	Poland,Warszawa

La función CONCAT tiene la ventaja de tratar de manera directa a NULL como cadena vacía en T-SQL.

```
SELECT CustomerID, Country, Region, City,
CONCAT(Country, ',' , Region, ',' , City) AS Ubicacion
FROM Customers;
```

CustomerID	Country	Region	City	Ubicacion
ALFKI	Germany	NULL	Berlin	Germany,Berlin
ANATR	Mexico	NULL	México D.F.	Mexico,México D.F.
ANTON	Mexico	NULL	México D.F.	Mexico,México D.F.
AROUT	UK	NULL	London	UK,London
...				
WELLI	Brazil	SP	Resende	Brazil,SP,Resende
WHITC	USA	WA	Seattle	USA,WA,Seattle
WILMK	Finland	NULL	Helsinki	Finland,Helsinki
WOLZA	Poland	NULL	Warszawa	Poland,Warszawa



FUNCIONES DE CADENA

A continuación, enumeraremos algunas de las funciones de cadena más usadas.

FUNCIÓN ASCII

Devuelve el valor ASCII del carácter que se le pase como parámetro. Si se le pasa una cadena, devuelve el valor ASCII del primer carácter.

ASCII(cadena)

```
SELECT ASCII('A')
```

65

FUNCIÓN CHAR

Convierte el entero que recibe como parámetro en el carácter correspondiente a su valor ASCII. En MySQL se devuelve una cadena binaria, a menos que se declare el charset a utilizar mediante USING.

SQL Server

CHAR(entero)

```
SELECT CHAR(65)
```

A

MySQL

CHAR(entero, ... [USING charset_name])

```
SELECT CHAR(65 USING ascii)
```

A

FUNCIÓN CHARINDEX

Devuelve la posición de la primera ocurrencia de una subcadena dentro de una cadena. La sintaxis es:

CHARINDEX(subcadena, cadena[, pos_desde])

Si no se especifica pos_desde busca desde el primer carácter. Si no encuentra la subcadena devuelve 0.

```
SELECT CHARINDEX(' ', 'Universidad Nacional de José C. Paz');
```

12

MySQL no tiene implementada la función CHARINDEX, pero se puede obtener un resultado similar con la función LOCATE.

LOCATE(subcadena, cadena [, pos_desde])

```
SELECT LOCATE(' ', 'Universidad Nacional de José C. Paz');
```

```
LOCATE(' ', 'Universidad Nacional de José C. Paz')|
```

-----+
12|



FUNCIÓN CONCAT

Esta función devuelve una cadena resultante de la concatenación, o unión, de dos o más valores de cadena de un extremo a otro.

CONCAT(cadena1, cadena2 [, cadena])

```
SELECT CONCAT('Universidad', 'Nacional', 'Jose C. Paz')
```

```
-----  
UniversidadNacionalJose C. Paz
```

FUNCIÓN CONCAT_WS

Esta función devuelve una cadena resultante de la concatenación, o unión, de dos o más valores de cadena de un extremo a otro. Es similar a la función CONCAT, pero tiene un primer parámetro que permite indicar una expresión para utilizar como separador.

CONCAT_WS(separador, cadena1, cadena2 [, cadena])

```
SELECT CONCAT_WS(' - ', 'Universidad', 'Nacional', 'Jose C. Paz')
```

```
-----  
Universidad - Nacional - Jose C. Paz
```

FUNCIÓN FORMAT (SQL SERVER)

Devuelve una cadena formateada con el formato y la referencia cultural opcional especificados en SQL Server 2016. Use la función FORMAT para aplicar formato específico de la configuración regional de los valores de fecha/hora y de número como cadenas. Para las conversiones de tipos de datos generales, use CAST o CONVERT. Más información en <http://go.microsoft.com/fwlink/?LinkId=211776>

FORMAT(entrada , formato_cadena, referencia_cultural)

```
SELECT FORMAT(1759, '00000000');
```

```
-----  
000001759
```

FUNCIÓN FORMAT (MYSQL)

En MySQL, la función le da formato al número X, en un formato del estilo #,###,###.##, redondeado a D decimales, y lo devuelve como cadena. Mediante el tercer parámetro, que es opcional, se puede definir la configuración regional a aplicar con respecto a separadores de miles y decimales.

FORMAT(X , D [, Locale])

```
SELECT FORMAT(12332.2,2,'es_ES');
```

```
FORMAT(12332.2,2,'es_ES')|
```

```
-----+|
```

```
12.332,20 |
```



FUNCIONES LEFT Y RIGHT

Estas funciones devuelven la cantidad de caracteres solicitadas desde la izquierda o la derecha de la cadena. La sintaxis es:

LEFT(cadena, n), RIGHT(cadena, n)

```
SELECT LEFT('Universidad', 5)
```

```
-----  
Unive
```

```
SELECT RIGHT('Universidad', 5)
```

```
-----  
sidad
```

FUNCIONES LEN Y DATALENGTH (SQL SERVER)

Ambas funciones devuelven la longitud de las cadenas, pero la diferencia está en que LEN devuelve en todas las oportunidades la cantidad de caracteres que forman la cadena, pero DATALENGTH devuelve la cantidad de bytes. En caracteres regulares ambos números coinciden, pero cuando trabajamos con caracteres Unicode (que se almacenan en doble byte) los resultados varían.

LEN (cadena), DATALENGTH (cadena)

```
SELECT LEN(N'abcde');  
SELECT LEN('abcde');
```

```
-----  
5
```

```
SELECT DATALENGTH('abcde');
```

```
-----  
5
```

```
SELECT DATALENGTH(N'abcde');
```

```
-----  
10
```

Otra diferencia entre ambas funciones es que LEN omite los blancos al final de la cadena.



FUNCIÓN LENGTH (MYSQL)

En MySQL no contamos con la función LEN y DATALENGTH, pero tenemos disponible la función LENGTH y la familia de funciones BIT_LENGTH(), CHAR_LENGTH(), OCTET_LENGTH(). La diferencia es que en MySQL la función LENGTH() devuelve la longitud en bytes (como DATALENGTH en T-SQL), y CHAR_LENGTH() devuelve la longitud en caracteres (como LEN en T-SQL).

LENGTH (cadena), CHAR_LENGTH (cadena)

```
SELECT LENGTH ('abcde');
LENGTH ('abcde')|
-----+
      5|
```

```
SELECT CHAR_LENGTH ('abcde');
CHAR_LENGTH ('abcde')|
-----+
      5|
```

FUNCIÓN LOWER

Devuelve la cadena de entrada toda en minúsculas.

LOWER(cadena)

```
SELECT LOWER('Universidad Nacional de José C. Paz');
-----+
universidad nacional de josé c. paz
```

FUNCIÓN LTRIM

Devuelve la cadena de entrada sin los espacios en blanco delanteros.

LTRIM(string)

```
SELECT LTRIM(' abc ');
-----+
abc
```

FUNCIÓN PATINDEX (SQL SERVER)

Devuelve la posición de la primera ocurrencia de un patrón dentro de una cadena. La sintaxis es:

PATINDEX(patron, cadena)

Si no encuentra el patrón devuelve 0.

```
SELECT PATINDEX('%[0-9]%', 'abcd123efgh');
-----+
5
```



En MySQL no está la función PATINDEX, pero tiene disponible la función REGEXP_INSTR, que permite realizar la búsqueda mediante expresiones regulares, que son más completas y potentes que los patrones.

REGEXP_INSTR(cadena, patron[, pos[, ocurrencia[, return_option [,match_type]]]])

```
SELECT REGEXP_INSTR('abcd123efgh', '[0-9]');
```

```
REGEXP_INSTR('abcd123efgh', '[0-9]')
-----+
      5|
```

FUNCIÓN REPLACE

Reemplaza todas las ocurrencias de una subcadena por otra. La sintaxis es:

REPLACE(cadena, subcadena1, subcadena2)

```
SELECT REPLACE('1-a 2-b', '-', ':');
```

```
-----+
1:a 2:b
```

FUNCIÓN REPLICATE/REPEAT

Replica una cadena el número de veces solicitado. La sintaxis es:

SQL Server

REPLICATE(cadena, n)

```
SELECT REPLICATE('abc', 3);
```

```
-----
abcabcabc
```

MySQL

REPEAT(cadena, n)

```
SELECT REPEAT('abc', 3);
```

```
REPEAT('abc', 3)
-----+
abcabcabc
```

FUNCIÓN REVERSE

Devuelve una cadena en orden invertido.

REVERSE (cadena)

```
SELECT REVERSE(1234) AS Invertido ;
```

```
Invertido
```

```
-----
4321
```



FUNCIÓN RTRIM

Devuelve la cadena de entrada sin los espacios en blanco traseros.

RTRIM (cadena)

```
SELECT RTRIM(' abc ');
```

```
-----  
abc
```

FUNCIÓN SPACE

Devuelve una cadena formada por tantos espacios como especifica el parámetro.

SPACE (entero)

```
SELECT SPACE(5)
```

```
-----
```

FUNCIÓN STR (SQL SERVER)

Devuelve datos de cadena convertidos a partir de datos numéricos. Los datos de cadena están justificados a la derecha, con una longitud y precisión decimaladas en los parámetros.

STR (expresión_numérica [, Longitud [, decimales]])

Expresión_numérica, es una expresión numérica aproximada (float) con punto decimal

Longitud, es la longitud total, incluye punto decimal, signo, dígitos y espacios. El valor por defecto es 10

Decimal, es la cantidad de números a la derecha del punto decimal. Debe ser igual o menor a 16.

```
SELECT STR(123.45, 6, 1);
```

```
-----  
123.5
```

En MySQL podemos utilizar la función CAST, con un ROUND anidado de ser necesario para lograr el mismo comportamiento.



FUNCIÓN STRING_AGG (SQL SERVER)

Concatena los valores de las expresiones de cadena y coloca valores de separador entre ellos. El separador no se agrega al final de la cadena.

```
STRING_AGG ( expresión , separador ) [ <cLauseLa_orden> ]  
<cLauseLa_orden> ::= WITHIN GROUP ( ORDER BY <expresión_orden> [ ASC | DESC ] )
```

```
SELECT STRING_AGG(FirstName, ',') WITHIN GROUP (ORDER BY FirstName ASC) AS csv  
FROM Employees;
```

CSV

```
-----  
Andrew,Anne,Janet,Laura,Margaret,Michael,Nancy,Robert,Steven
```

En MySQL podemos obtener un resultado similar con la función GROUP_CONCAT()

```
GROUP_CONCAT ([DISTINCT] expresión [, expresión] [ <cLauseLa_orden> ] [SEPARATOR str])  
<cLauseLa_orden> ::= ORDER BY <expresión_orden> [ ASC | DESC ]
```

```
SELECT GROUP_CONCAT(FirstName ORDER BY FirstName ASC) AS csv  
FROM Employees;
```

CSV

```
-----  
Andrew,Anne,Janet,Laura,Margaret,Michael,Nancy,Robert,Steven
```

FUNCIÓN STRING_SPLIT (SQL SERVER)

Es una función de tabla (como resultado devuelve una tabla), que separa los valores de una cadena, según el separador especificado, y devuelve una fila por cada valor.

```
SELECT value FROM STRING_SPLIT(cadena, separador);
```

```
SELECT CAST(value AS INT) AS misvalores  
FROM STRING_SPLIT('10248,10249,10250', ',') AS S;  
  
misvalores  
-----  
10248  
10249  
10250
```



FUNCIÓN STUFF / INSERT

Remueve una subcadena de una cadena e inserta una nueva subcadena en su lugar.

SQL Server

```
STUFF(cadena, pos, Longitud_a_borrar,
       cadena_a_insertar)
```

```
SELECT STUFF('xyz', 2, 1, 'abc');

-----
xabcz
```

MySQL

```
INSERT(cadena, pos, Longitud_a_borrar,
       cadena_a_insertar)
```

```
SELECT INSERT('xyz', 2, 1, 'abc');

INSERT('xyz', 2, 1, 'abc')|
-----
+-----+
|      |
xabcz
```

FUNCIÓN SUBSTRING

Extrae una subcadena de una cadena. La sintaxis es:

```
SUBSTRING(cadena, comienzo, Longitud)
```

```
SELECT SUBSTRING('abcdefg', 2, 3);

-----
bcd
```

FUNCIÓN TRANSLATE (SQL SERVER)

Devuelve la cadena proporcionada como primer argumento después de que algunos caracteres especificados en el segundo argumento se traduzcan a un conjunto de caracteres de destino especificado en el tercer argumento.

```
TRANSLATE ( cadena, caracteres, traducciones )
```

```
SELECT TRANSLATE('2*[3+4]/{7-2}', '[]{}', '()());

-----
2*(3+4)/(7-2)
```

En MySQL se puede lograr un resultado similar encadenando funciones REPLACE, pero hay que manejarlas con cuidado si hay coincidencias entre los caracteres originales y sus traducciones.



FUNCIÓN TRIM

Elimina el carácter de espacio char (32) u otros caracteres especificados del principio y final de una cadena.

TRIM ([caracteres FROM] cadena)

```
SELECT TRIM(' abc ')
-----
abc

SELECT TRIM( '.,,! ' FROM      #      prueba      .') AS Resultado;
Resultado
-----
#      prueba
```

FUNCIÓN UPPER

Devuelve la cadena de entrada toda en mayúsculas.

UPPER(cadena)

```
SELECT UPPER('Universidad Nacional de José C. Paz');
-----
UNIVERSIDAD NACIONAL DE JOSÉ C. PAZ
```



ANEXO: TRABAJANDO CON DATOS DE FECHA Y HORA EN SQL SERVER

Trabajar con datos de fecha y hora en SQL no es trivial. Hay muchos desafíos en esta área, como expresar literales de una manera neutral al lenguaje y trabajar con la fecha y la hora por separado.

LITERALES

Cuando necesitamos especificar un literal (constante) de un tipo de datos de fecha y hora, debemos considerar varias cosas. En primer lugar, aunque podría sonar un poco extraño, SQL no proporciona los medios para expresar una fecha y hora literal. En su lugar, podemos especificar un literal de un tipo diferente que se puede convertir, explícita o implícitamente, en un tipo de datos de fecha y hora. Es una práctica recomendada utilizar cadenas de caracteres para expresar valores de fecha y hora, como se muestra en el ejemplo siguiente:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate  
FROM Orders  
WHERE OrderDate = '19960212';
```

SQL reconoce el literal '19960212' como un literal de cadena de caracteres y no como una fecha y hora literal, pero debido a que la expresión implica operandos de dos tipos diferentes, un operando necesita convertirse implícitamente al tipo del otro. Normalmente, la conversión implícita entre tipos se basa en lo que se llama precedencia de tipo de datos. SQL define la precedencia entre los tipos de datos y suele convertir implícitamente el operando que tiene una prioridad de tipo de datos inferior a la que tiene mayor precedencia. En este ejemplo, el literal de cadena de caracteres se convierte al tipo de datos de la columna (DATETIME) porque las cadenas de caracteres se consideran inferiores en términos de precedencia de tipo de datos con respecto a los tipos de datos de fecha y hora.

El punto que estoy tratando de hacer es que, en el ejemplo anterior, la conversión implícita tiene lugar detrás de las escenas. La conversión se puede hacer de manera explícita, como podemos ver en el siguiente ejemplo:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate  
FROM Orders  
WHERE OrderDate = CAST('19960212' AS date);
```

Esta consulta es equivalente a la anterior, con la diferencia que la primera hace la conversión de manera implícita y esta lo hace de manera explícita.

Tengan en cuenta que algunos formatos de cadena de caracteres de literales de fecha y hora son dependientes del idioma, lo que significa que, al convertirlos a un tipo de datos de fecha y hora, SQL podría interpretar el valor de forma diferente en función de la configuración de idioma en vigor en la sesión. Cada inicio de sesión definido por el administrador de la base de datos tiene un lenguaje predeterminado asociado a él y, a menos que se cambie explícitamente, ese idioma se convierte en el idioma efectivo en la sesión. Se puede sobrescribir el idioma predeterminado en la sesión mediante el comando SET LANGUAGE en SQL Server pero generalmente no se recomienda porque algunos aspectos del código pueden basarse en el idioma predeterminado del usuario.

El lenguaje efectivo en la sesión establece varios ajustes relacionados con el idioma detrás de escena. Entre ellos se encuentra uno llamado DATEFORMAT, que determina cómo SQL Server interpreta los literales que ingresa cuando se convierten de un tipo de cadena de caracteres a un tipo de fecha y hora. El ajuste DATEFORMAT se expresa como una combinación de los caracteres d, m, yy. Por ejemplo, la configuración de idioma us_english establece el DATEFORMAT en mdy, mientras que el idioma inglés británico establece el DATEFORMAT en dmy. Puede anular la configuración DATEFORMAT en su sesión mediante el comando SET DATEFORMAT, pero como se mencionó antes, generalmente no se recomienda cambiar la configuración relacionada con el idioma.



Consideremos, por ejemplo, el literal '02/12/2016'. SQL Server puede interpretar la fecha como 12 de febrero de 2016 o 2 de diciembre de 2016 cuando convierta este literal a uno de los siguientes tipos: DATETIME, SMALLDATETIME, DATE, DATETIME2 o DATETIMEOFFSET. El parámetro LANGUAGE / DATEFORMAT efectivo es el factor determinante. Para demostrar diferentes interpretaciones de la misma cadena de caracteres literal, ejecute el siguiente código:

```
SET LANGUAGE British;
SELECT CAST('02/12/2016' AS date);
SET LANGUAGE us_english;
SELECT CAST('02/12/2016' AS date);
Changed language setting to British.
```

2016-12-02

Changed language setting to us_english.

2016-02-12

Tenga en cuenta que el ajuste LANGUAGE / DATEFORMAT sólo afecta a la forma en que se interpretan los valores introducidos. Estos ajustes no tienen impacto en el formato utilizado en la salida para propósitos de presentación. El formato de salida se determina por la interfaz de base de datos utilizada por la herramienta de cliente (como ODBC) y no por la configuración LANGUAGE / DATEFORMAT. Por ejemplo, OLE DB y ODBC presentan valores de fecha en el formato 'AAAA-MM-DD'.

Debido a que el código que escribe puede ser utilizado por usuarios internacionales con diferentes configuraciones de idioma para sus inicios de sesión, la comprensión de que algunos formatos de literales dependen del idioma es crucial. Es muy recomendable que exprese sus literales de una manera neutral al lenguaje. Los formatos de lenguaje neutro siempre son interpretados por SQL Server de la misma manera y no se ven afectados por la configuración relacionada con el idioma.

En la siguiente tabla podemos ver los literales neutrales para cada tipo:

Tipo de Dato	Precisión	Formato recomendado de entrada y ejemplo
DATETIME	'AAAMMDD hh:mm:ss.nnn' 'AAAA-MM-DDThh:mm:ss.nnn' 'AAAAMMDD'	'20160212 12:30:15.123' '2016-02-12T12:30:15.123' '20160212'
SMALLDATETIME	'AAAAMMDD hh:mm' 'AAAA-MM-DDThh:mm' 'AAAAMMDD'	'20160212 12:30' '2016-02-12T12:30' '20160212'
DATE	'AAAAMMDD' 'AAAA-MM-DD'	'20160212' '2016-02-12'
DATETIME2	'AAAMMDD hh:mm:ss.nnnnnnnn' 'AAAA-MM-DD hh:mm:ss.nnnnnnnn' 'AAAA-MM-DDThh:mm:ss.nnnnnnnn' 'AAAAMMDD' 'AAAA-MM-DD'	'20160212 12:30:15.1234567' '2016-02-12 12:30:15.1234567' '2016-02-12T12:30:15.1234567' '20160212' '2016-02-12'
DATETIMEOFFSET	'AAAMMDD hh:mm:ss.nnnnnnnn [+ -]hh:mm' 'AAAA-MM-DD hh:mm:ss.nnnnnnnn [+ -]hh:mm' 'AAAAMMDD' 'AAAA-MM-DD'	'20160212 12:30:15.1234567 +02:00' '2016-02-12 12:30:15.1234567 +02:00' '20160212' '2016-02-12'
TIME	'hh:mm:ss.nnnnnnnn'	'12:30:15.1234567'

Los tipos que incluyen fecha y hora, cuando no se establece la hora en el literal, SQL Server asume como hora la medianoche. Si no se establece la diferencia con UTC, se asume 00:00.



Los literales 'YYYY-MM-DD' y 'YYYY-MM-DD hh:mm...' son dependientes del lenguaje cuando son convertidos a DATETIME y SMALLDATETIME, pero son neutrales cuando son convertidos a DATE, DATETIME2 y DATETIMEOFFSET.

Por lo tanto, es recomendable usar la forma 'YYYYMMDD', ya que siempre es neutral.

Si a pesar de las recomendaciones, se quiere usar un formato dependiente del idioma para expresar literales, hay dos opciones disponibles. Una es usar la función CONVERT para convertir explícitamente el literal de cadena de caracteres al tipo de datos solicitado y, en el tercer argumento, especificar un número que represente el estilo que usó. Los libros en pantalla de SQL Server tienen una tabla con todos los números de estilo y los formatos que representan. Por ejemplo, si desea especificar el literal '02/12/2016' con el formato MM/DD/YYYY, utilice el número de estilo 101, como se muestra aquí:

```
SELECT CONVERT(DATE, '02/12/2016', 101);
```

```
-----  
2016-02-12
```

El literal se interpreta como 12 de febrero 2016, independientemente de la configuración de idioma que está en vigente.

Si desea utilizar el formato DD/MM/YYYY, utilice el número de estilo 103:

```
SELECT CONVERT(DATE, '02/12/2016', 103);
```

```
-----  
2016-12-02
```

Esta vez, el literal se interpreta como el 2 de diciembre de 2016.

Otra opción es utilizar la función PARSE. Al usar esta función, puede analizar un valor como un tipo solicitado e indicar la cultura. Por ejemplo, lo siguiente es el equivalente de usar CONVERT con estilo 101 (inglés de EE. UU.):

```
SELECT PARSE('02/12/2016' AS date USING 'en-US');
```

```
-----  
2016-02-12
```

Y el siguiente es equivalente a usar CONVERT con el estilo 103 (inglés británico):

```
SELECT PARSE('02/12/2016' AS date USING 'en-GB');
```

```
-----  
2016-12-02
```

TRABAJANDO CON FECHA Y HORA POR SEPARADO

Si necesitamos trabajar sólo con fechas o sólo horas, se recomienda utilizar los tipos de datos DATE y TIME, respectivamente. Seguir este consejo puede convertirse en un desafío si necesita restringirse a usar sólo los tipos heredados DATETIME y SMALLDATETIME por razones tales como la compatibilidad con sistemas antiguos. El reto es que los tipos heredados contienen los componentes de fecha y hora. La mejor práctica en tal caso dice que cuando se quiere trabajar sólo con fechas, se almacena la fecha con un valor de medianoche en la parte de tiempo. Cuando desea trabajar sólo con tiempos, almacena la hora con la fecha base 1 de enero de 1900.



Para demostrar el trabajo con fecha y hora por separado, utilizaré una tabla llamada Orders2, que tiene una columna llamada OrderDate de un tipo de datos DATETIME. Ejecute el código siguiente para crear la tabla Orders2 copiando datos de la tabla Orders y casteando la columna OrderDate de origen, que es de un tipo DATE, a DATETIME:

```
DROP TABLE IF EXISTS Orders2;
SELECT OrderID, CustomerID, EmployeeID, CAST(OrderDate AS datetime) AS OrderDate
INTO Orders2
FROM Orders;
```

Como se mencionó, la columna OrderDate en la tabla Orders2 es de un tipo de datos DATETIME, pero como sólo el componente fecha es realmente relevante, todos los valores contienen la medianoche como hora. Cuando necesita filtrar sólo pedidos de una fecha determinada, no tiene que utilizar un filtro de rango. En su lugar, puede utilizar el operador de igualdad de la siguiente manera:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM Orders2
WHERE OrderDate = '19960212';
```

Cuando SQL Server convierte un literal de cadena de caracteres que sólo tiene una fecha a DATETIME, asume por default de hora la medianoche. Dado que todos los valores de la columna de fecha de pedido contienen la medianoche en el componente de tiempo, se devolverán todos los pedidos realizados en la fecha solicitada.

Si la parte de horas tuviera un valor distinto a la medianoche, la consulta se puede hacer filtrando por rangos:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM Orders2
WHERE OrderDate >= '19960212'
AND OrderDate < '19960213';
```

Si desea trabajar sólo con horas utilizando los tipos legados, puede almacenar todos los valores con la fecha base del 1 de enero de 1900. Cuando SQL Server convierte un literal de cadena de caracteres que contiene sólo un componente de tiempo a DATETIME o SMALLDATETIME, SQL Server Supone que la fecha es la fecha base. Por ejemplo, ejecute el código siguiente:

```
SELECT CAST('12:30:15.123' AS DATETIME);
```

```
-----
1900-01-01 12:30:15.123
```

FILTRANDO RANGOS DE FECHAS

Cuando se necesita filtrar un rango de fechas, como un año entero o un mes completo, parece natural utilizar funciones como YEAR y MONTH. Por ejemplo, la siguiente consulta devuelve todos los pedidos realizados en el año 1996:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM Orders
WHERE YEAR(OrderDate) = 1996;
```

Sin embargo, debemos tener en cuenta que, en la mayoría de los casos, cuando se aplica la manipulación en la columna filtrada, SQL Server no puede utilizar un índice de manera eficiente. Esto es probablemente difícil de entender sin conocimientos previos sobre los índices y el rendimiento, que aún no hemos visto. Por ahora, basta con tener presente



este punto general: Para tener el potencial de usar un índice de manera eficiente, no debe manipular la columna filtrada. Para lograr esto, puede corregir el predicado de filtro de la última consulta se la siguiente manera:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM Orders
WHERE OrderDate >= '19960101' AND OrderDate < '19970101';
```

Del mismo modo, si queremos filtrar las órdenes emitidas en un mes en particular:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM Orders
WHERE YEAR(OrderDate) = 1996 AND MONTH(OrderDate) = 7;
```

Puede escribirse con rangos de la siguiente manera:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM Orders
WHERE OrderDate >= '19960701' AND OrderDate < '19960801';
```

FUNCIONES DE FECHA Y HORA

OBTENER LA FECHA Y HORA ACTUAL

Las siguientes funciones niládicas (sin parámetros) devuelven los valores actuales de fecha y hora en el sistema donde reside la instancia de SQL Server: GETDATE, CURRENT_TIMESTAMP, GETUTCDATE, SYSDATETIME, SYSUTCDATETIME y SYSDATETIMEOFFSET.

Función	Tipo Devuelto	Descripción
GETDATE	DATETIME	Fecha y hora actual
CURRENT_TIMESTAMP	DATETIME	Igual a GETDATE pero cumple SQL ANSI
GETUTCDATE	DATETIME	Fecha y hora actual en UTC
SYSDATETIME	DATETIME2	Fecha y hora actual
SYSUTCDATETIME	DATETIME2	Fecha y hora actual en UTC
SYSDATETIMEOFFSET	DATETIMEOFFSET	Fecha y hora actual incluyendo zona

Tenga en cuenta que debe especificar paréntesis vacíos con todas las funciones que se deben especificar sin parámetros, excepto la función estándar CURRENT_TIMESTAMP. Además, debido a que CURRENT_TIMESTAMP y GETDATE devuelven lo mismo, pero sólo el primero es estándar, se recomienda que utilice el anterior.

El siguiente código muestra el uso de las funciones actuales de fecha y hora:

```
SELECT
GETDATE() AS [GETDATE],
CURRENT_TIMESTAMP AS [CURRENT_TIMESTAMP],
GETUTCDATE() AS [GETUTCDATE],
SYSDATETIME() AS [SYSDATETIME],
SYSUTCDATETIME() AS [SYSUTCDATETIME],
SYSDATETIMEOFFSET() AS [SYSDATETIMEOFFSET];
```



Como probablemente habrán notado, ninguna de las funciones devuelve sólo la fecha del sistema actual o sólo la hora actual del sistema. Sin embargo, se pueden obtener fácilmente mediante la conversión de CURRENT_TIMESTAMP o SYSDATETIME a DATE o TIME de esta manera:

```
SELECT
CAST(SYSDATETIME() AS DATE) AS [current_date],
CAST(SYSDATETIME() AS TIME) AS [current_time];
```

FUNCIONES CAST, CONVERT Y PARSE Y SUS CONTRAPARTES TRY_

Las funciones CAST, CONVERT y PARSE se utilizan para convertir un valor de entrada en algún tipo de destino. Si la conversión tiene éxito, las funciones devuelven el valor convertido, de lo contrario, provocan la falla de la consulta. Las tres funciones tienen contrapartidas llamadas TRY_CAST, TRY_CONVERT y TRY_PARSE, respectivamente. Cada versión con el prefijo TRY_ acepta la misma entrada que su contraparte y aplica la misma conversión. La diferencia es que, si la entrada no es convertible al tipo de destino, la función devuelve un NULL en lugar de fallar la consulta. Las funciones TRY_ se incorporaron en SQL Server 2012.

$$\begin{aligned} & \text{CAST}(\text{valor AS tipo de datos}) \\ & \text{TRY_CAST}(\text{valor AS tipo de datos}) \\ & \text{CONVERT (tipo de datos, valor [, estilo de número])} \\ & \text{TRY_CONVERT (tipo de datos, valor [, estilo de número])} \\ & \text{PARSE (valor AS tipo de datos [USING cultura])} \\ & \text{TRY_PARSE (valor AS tipo de datos [USING cultura])} \end{aligned}$$

Las tres funciones de base convierten el valor de entrada al tipo de dato de destino especificado. En algunos casos, CONVERT tiene un tercer argumento con el que puede especificar el estilo de la conversión.

Por ejemplo, cuando se convierte de una cadena de caracteres a uno de los tipos de datos de fecha y hora (o al revés), el número de estilo indica el formato de la cadena. Por ejemplo, el estilo 101 indica 'MM/DD/YYYY', y el estilo 103 indica 'DD/MM/YYYY'. Puede encontrar la lista completa de números de estilo y sus significados en los Libros en pantalla de SQL Server en "CAST y CONVERT". De forma similar, cuando corresponda, la función PARSE admite la indicación de una cultura (por ejemplo, 'en-US' Y 'en-GB' para el inglés británico).

Como se mencionó anteriormente, cuando se está convirtiendo de una cadena de caracteres a uno de los tipos de datos de fecha y hora, algunos de los formatos de cadena son dependientes del idioma. Se recomienda utilizar uno de los formatos de idioma neutro o utilizar la función CONVERT y especificando explícitamente el número de estilo. De esta manera, su código se interpreta de la misma manera independientemente del idioma del login que lo ejecuta.

Tenga en cuenta que CAST es estándar y CONVERT y PARSE no lo son, por lo que a menos que necesite utilizar el número de estilo o cultura, se recomienda utilizar la función CAST.

FUNCIÓN DATEPART

Devuelve un integer representa la unidad deseada de la fecha

$$\text{DATEPART(unid, fecha_val)}$$


Los valores válidos para el argumento *unid* incluyen year, quarter, month, dayofyear, day, week, weekday, hour, minute, second, millisecond, microsecond, nanosecond, TZoffset, e ISO_WEEK. Como se mencionó, puede utilizar abreviaturas para las partes de fecha y hora, como yy en lugar de year, mm en lugar de month, dd en lugar de day y así sucesivamente.

```
SELECT DATEPART(month, '20160212');

-----
2
```

FUNCIONES YEAR, MONTH Y DAY

Las funciones YEAR, MONTH y DAY son abreviaturas para la función DATEPART que devuelve la representación entera de las partes del año, mes y día de un valor de fecha y hora de entrada.

YEAR(fecha_val)

MONTH(fecha_val)

DAY(fecha_val)

```
SELECT
DAY('20160212') AS eldia,
MONTH('20160212') AS elmes,
YEAR('20160212') AS elanio;

eldia      elmes      elanio
-----      -----
12          2          2016
```

FUNCIÓN DATENAME

La función DATENAME devuelve una cadena de caracteres que representa un valor en una unidad determinada de un valor de fecha y hora.

DATENAME(fecha_val, unid)

Esta función es similar a DATEPART y, de hecho, tiene las mismas opciones para la entrada de unid. Sin embargo, cuando es relevante, devuelve el nombre de la parte solicitada en lugar del número.

```
SELECT DATENAME(month, '20160212');

-----
February
```

Recordemos que DATEPART devolvió el entero 2 para esta entrada. DATENAME devuelve el nombre del mes, que depende del idioma. Si el idioma de la sesión es uno de los idiomas en inglés (como el inglés de los Estados Unidos o inglés británico), se recupera el valor 'February'. Si el idioma de la sesión es el italiano, se recupera el valor 'febbraio'. Si se solicita una parte que no tiene nombre y sólo un valor numérico (como año), la función DATENAME devuelve su valor numérico como una cadena de caracteres.



```
SELECT DATENAME(year, '20160212');
```

```
-----  
2016
```

FUNCIONES FROMPARTS

Las funciones FROMPARTS aceptan entradas enteras que representan partes de un valor de fecha y hora y construyen un valor del tipo solicitado desde esas partes.

DATEFROMPARTS (year, month, day)

DATETIME2FROMPARTS (year, month, day, hour, minute, seconds, fractions, precision)

DATETIMEFROMPARTS (year, month, day, hour, minute, seconds, milliseconds)

DATETIMEOFFSETFROMPARTS (year, month, day, hour, minute, seconds, fractions, hour_offset, minute_offset, precision)

SMALLDATETIMEFROMPARTS (year, month, day, hour, minute)

TIMEFROMPARTS (hour, minute, seconds, fractions, precision)

Estas funciones facilitan a los programadores la construcción de valores de fecha y hora de los diferentes componentes y también simplifican la migración de código de otros entornos que ya soportan funciones similares.

```
SELECT  
DATEFROMPARTS(2016, 02, 12),  
DATETIME2FROMPARTS(2016, 02, 12, 13, 30, 5, 1, 7),  
DATETIMEFROMPARTS(2016, 02, 12, 13, 30, 5, 997),  
DATETIMEOFFSETFROMPARTS(2016, 02, 12, 13, 30, 5, 1, -8, 0, 7),  
SMALLDATETIMEFROMPARTS(2016, 02, 12, 13, 30),  
TIMEFROMPARTS(13, 30, 5, 1, 7);
```

```
-----  
2016-02-12 2016-02-12 13:30:05.0000001 2016-02-12 13:30:05.997 2016-02-12 13:30:05.0000001 -08:00 2016-02-12 13:30:00 13:30:05.0000001
```

FUNCIÓN SWITCHOFFSET

Ajusta una fecha y hora de tipo DATETIMEOFFSET a una zona horaria particular. La sintaxis es:

SWITCHOFFSET(valor_datetimeoffset, UTC_Offset)

```
SELECT SWITCHOFFSET(SYSDATETIMEOFFSET(), '-05:00');
```

```
-----  
2017-04-08 19:29:00.8186278 -05:00
```

```
SELECT SWITCHOFFSET(SYSDATETIMEOFFSET(), '+00:00');
```

```
-----  
2017-04-09 00:29:00.8186278 +00:00
```



FUNCIÓN TODATETIMEOFFSET

Para setear la zona horaria a una fecha y hora. La sintaxis es:

TODATETIMEOFFSET(valor_fecha_y_hora_Local, UTC_Offset)

Esta función se diferencia de SWITCHOFFSET en que su primera entrada es un valor local de fecha y hora sin un componente de desplazamiento. Esta función simplemente fusiona el valor de fecha y hora de entrada con el desplazamiento especificado para crear un nuevo valor de tiempo de salida de fecha.

Por lo general, utilizará esta función al migrar datos no compatibles con offset para datos con detección de offset. Imagine que tiene una tabla que contiene valores de fecha y hora locales en un atributo llamado dt de un tipo de datos DATETIME2 o DATETIME y que mantiene el desplazamiento en un atributo denominado offset.

A continuación, decide combinar los dos a un atributo que reconoce offset denominado dto. Cambia la tabla y agrega el nuevo atributo. A continuación, actualiza el resultado de la expresión TODATETIMEOFFSET (dt, theoffset). A continuación, puede eliminar los dos atributos existentes dt y theoffset.

FUNCIÓN AT TIME ZONE

Acepta un valor de fecha y hora de entrada y lo convierte en un valor de hora de salida que corresponde a la zona horaria objetivo especificada. Esta función se introdujo en SQL Server 2016. La sintaxis es la siguiente:

fecha_val AT TIME ZONE zona_horaria

La entrada fecha_val puede ser de los siguientes tipos de datos: DATETIME, SMALLDATETIME, DATETIME2 y DATETIMEOFFSET. La entrada zona_horaria puede ser cualquiera de los nombres de zona horaria de Windows admitidos, tal como aparecen en la columna de nombre en la vista sys.time_zone_info.

Utilice la consulta siguiente para ver las zonas horarias disponibles, su desplazamiento actual de UTC y si es actualmente hora de verano (DST):

```
SELECT name, current_utc_offset, is_currently_dst
FROM sys.time_zone_info;
```

Respecto a fecha_val: cuando se utiliza cualquiera de los tres tipos no-datetimeoffset (DATETIME, SMALLDATETIME y DATETIME2), la función AT TIME ZONE asume que el valor de entrada ya está en la zona horaria objetivo. Como resultado, se comporta de forma similar a la función TODATETIMEOFFSET, excepto que el desplazamiento no es necesariamente fijo. Depende de si se aplica DST. Tome como ejemplo la zona horaria de la hora estándar del Pacífico. Cuando no es DST, la diferencia de UTC es -08: 00; Cuando es DST, el desplazamiento es -07: 00. El código siguiente demuestra el uso de esta función con las entradas no-datetimeoffset:

```
SELECT
CAST('20160212 12:00:00.000000' AS datetime2)
AT TIME ZONE 'Pacific Standard Time' AS val1,
CAST('20160812 12:00:00.000000' AS datetime2)
AT TIME ZONE 'Pacific Standard Time' AS val2;

val1           val2
-----
2016-02-12 12:00:00.000000 -08:00 2016-08-12 12:00:00.000000 -07:00
```



El primer valor ocurre cuando DST no se aplica; Por lo tanto, el desplazamiento -08:00 se asume. El segundo valor ocurre durante el horario de verano; Por lo tanto, se supone offset -07:00. Aquí no hay ambigüedad.

Hay dos casos difíciles: al cambiar a DST y al cambiar de DST. Por ejemplo, en Pacific Standard Time, al cambiar a DST el reloj se avanza por una hora, por lo que hay una hora que no existe. Si especifica un tiempo no existente durante esa hora, se supondrá el desplazamiento antes del cambio (-08: 00). Al cambiar de DST, el reloj retrocede por una hora, así que hay una hora que se repite. Si se especifica una hora durante esa hora, comenzando en la parte inferior de la hora, se supone que no es DST (es decir, se utiliza el desplazamiento -08: 00).

Cuando la entrada fecha_val es un valor datetimeoffset, la función AT TIME ZONE se comporta de forma más similar a la función SWITCHOFFSET. De nuevo, sin embargo, el desplazamiento de destino no es necesariamente fijo. T-SQL utiliza las reglas de conversión de zona horaria de Windows para aplicar la conversión. El código siguiente demuestra el uso de la función con entradas datetimeoffset:

```
SELECT
CAST('20160212 12:00:00.000000 -05:00' AS datetimeoffset)
AT TIME ZONE 'Pacific Standard Time' AS val1,
CAST('20160812 12:00:00.000000 -04:00' AS datetimeoffset)
AT TIME ZONE 'Pacific Standard Time' AS val2;
```

Los valores de entrada reflejan la zona horaria Hora estándar del este. Ambos tienen el mediodía en el componente de hora. El primer valor ocurre cuando DST no se aplica (offset es -05:00), y el segundo ocurre cuando DST se aplica (es decir, el desplazamiento es -04:00). Ambos valores se convierten a la zona horaria Hora estándar del Pacífico (la compensación -08:00 cuando DST no se aplica y -07:00 cuando lo hace). En ambos casos, el tiempo necesita retroceder en tres horas a las 9:00 AM. Obtendrá la salida siguiente:

val1	val2
2016-02-12 09:00:00.000000 -08:00	2016-08-12 09:00:00.000000 -07:00

FUNCIÓN DATEADD

Permite agregar una cantidad específica de unidades (especificadas en el primer parámetro) a una fecha y hora

DATEADD(unid, n, fecha_val)

Los valores válidos para *unid* incluyen year, quarter, month, dayofyear, day, week, weekday, hour, minute, second, millisecond, microsecond, y nanosecond. También puede especificar *unid* en forma abreviada, como yy en lugar de year. Consulte los Libros en pantalla de SQL Server para obtener más información.

El tipo devuelto para un parámetro de fecha y hora es del mismo tipo que el tipo del parámetro de entrada. Si a esta función se le da una cadena literal como entrada, la salida es DATETIME.

```
SELECT DATEADD(year, 1, '20160212');
-----
2017-02-12 00:00:00.000
```



FUNCIONES DATEDIFF Y DATEDIFF_BIG

Devuelven la diferencia entre dos valores de fecha y hora en términos de una unidad de fecha especificada. El primero devuelve un valor tipificado como INT (un entero de 4 bytes), y el segundo devuelve un valor escrito como BIGINT (un entero de 8 bytes). La función DATEDIFF_BIG se introdujo en SQL Server 2016.

DATEDIFF(unid, fecha_val1, fecha_val2), DATEDIFF_BIG(unid, fecha_val1, fecha_val2)

```
SELECT DATEDIFF(day, '20150212', '20160212');
-----
365
SELECT DATEDIFF_BIG(millisecond, '00010101', '20160212');
-----
63590832000000
```

Si necesitamos calcular el comienzo del día que corresponde a un valor de fecha y hora de entrada, simplemente podemos castear el valor de entrada al tipo DATE y luego mostrar el resultado al tipo de destino. Pero con un uso un poco más sofisticado de las funciones DATEADD y DATEDIFF, podemos calcular el comienzo o el final de diferentes unidades (día, mes, trimestre, año) que corresponden al valor de entrada.

```
SELECT
DATEADD(day,DATEDIFF(day, '19000101', SYSDATETIME()), '19000101');
```

Esto se logra utilizando primero la función DATEDIFF para calcular la diferencia en términos de días enteros entre una fecha de anclaje a medianoche ('19000101' en este caso) y la fecha y hora actuales (llamar a esa diferencia diff). A continuación, la función DATEADD se utiliza para añadir días diff al ancla. Y así se obtiene la fecha actual del sistema a la medianoche.

FUNCIÓN ISDATE

Acepta una cadena de caracteres como entrada y devuelve 1 si es convertible en un tipo de datos de fecha y hora y 0 si no lo es.

ISDATE(string)

```
SELECT ISDATE('20160212');
-----
1
SELECT ISDATE('20160230');
-----
0
```



FUNCIÓN EOMONTH

La función EOMONTH acepta un valor de fecha y hora de entrada y devuelve la fecha respectiva del fin de mes como un valor de tipo DATE. La función también admite un segundo argumento opcional que indica cuántos meses añadir (o restar, si es negativo).

EOMONTH(entrada [, meses_a_agregar])

```
SELECT EOMONTH(SYSDATETIME());  
-----  
2021-05-31
```



ANEXO: TRABAJANDO CON DATOS DE FECHA Y HORA EN MYSQL³¹

Los valores de fecha y hora se pueden representar en varios formatos, como cadenas entrecomilladas o como números, según el tipo exacto del valor y otros factores. Por ejemplo, en contextos donde MySQL espera una fecha, interpreta cualquiera de '2015-07-21', '20150721' y 20150721 como una fecha.

Este anexo describe los formatos aceptables para los literales de fecha y hora.

LITERALES ESTÁNDAR DE FECHA Y HORA DE SQL Y ODBC

El SQL estándar requiere que se especifiquen literales temporales mediante una palabra clave de tipo y una cadena. El espacio entre la palabra clave y la cadena es opcional.

```
DATE 'str'  
TIME 'str'  
TIMESTAMP 'str'
```

MySQL reconoce pero, a diferencia del SQL estándar, no requiere la palabra clave de tipo. Las aplicaciones que deben cumplir con los estándares deben incluir la palabra clave de tipo para los literales temporales.

MySQL también reconoce la sintaxis ODBC correspondiente a la sintaxis SQL estándar:

```
{ d 'str' }  
{ t 'str' }  
{ ts 'str' }
```

MySQL utiliza las palabras clave de tipo y las construcciones ODBC para producir valores DATE, TIME y DATETIME, respectivamente, incluida una parte final de segundos fraccionarios si se especifica. La sintaxis de TIMESTAMP produce un valor DATETIME en MySQL porque DATETIME tiene un rango que se corresponde más con el tipo estándar de SQL TIMESTAMP, que tiene un rango de años de 0001 a 9999. (El rango de años de MySQL TIMESTAMP es de 1970 a 2038).

LITERALES DE CADENA Y NUMÉRICOS EN CONTEXTO DE FECHA Y HORA

MySQL reconoce valores de FECHA en estos formatos:

- Como una cadena en formato 'YYYY-MM-DD' o 'YY-MM-DD'. Se permite una sintaxis "relajada", pero está en desuso: se puede usar cualquier carácter de puntuación como delimitador entre las partes de la fecha. Por ejemplo, '2012-12-31', '2012/12/31', '2012^12^31' y '2012@12@31' son equivalentes. A partir de MySQL 8.0.29, el uso de cualquier carácter que no sea el guión (-) como delimitador genera una advertencia, como se muestra aquí:

```
mysql> SELECT DATE '2012@12@31';  
+-----+  
| DATE '2012@12@31' |  
+-----+  
| 2012-12-31 |  
+-----+  
1 row in set, 1 warning (0.00 sec)  
  
mysql> SHOW WARNINGS\G  
***** 1. row *****  
Level: Warning  
Code: 4095
```

³¹ <https://dev.mysql.com/doc/refman/8.0/en/date-and-time-literals.html>



```
Message: Delimiter '@' in position 4 in datetime value '2012@12@31' at row 1 is
deprecated. Prefer the standard '-'.
1 row in set (0.00 sec)
```

- Como una cadena sin delimitadores en formato 'YYYYMMDD' o 'YYMMDD', siempre que la cadena tenga sentido como una fecha. Por ejemplo, '20070523' y '070523' se interpretan como '2007-05-23', pero '071332' es ilegal (tiene partes de mes y día sin sentido) y se convierte en '0000-00-00'.
- Como un número en formato AAAAMMDD o AAAAMMDD, siempre que el número tenga sentido como una fecha. Por ejemplo, 19830905 y 830905 se interpretan como '1983-09-05'.

MySQL reconoce los valores DATETIME y TIMESTAMP en estos formatos:

- Como una cadena en formato 'YYYY-MM-DD hh:mm:ss' o 'YY-MM-DD hh:mm:ss'. MySQL también permite una sintaxis "relajada" aquí, aunque esto está en desuso: cualquier carácter de puntuación puede usarse como delimitador entre las partes de la fecha o la hora. Por ejemplo, '2012-12-31 11:30:45', '2012^12^31 11+30+45', '2012/12/31 11*30*45' y '2012@12@31 11 ^30^45' son equivalentes. A partir de MySQL 8.0.29, el uso de caracteres como delimitadores en tales valores, que no sean el guión (-) para la parte de la fecha y los dos puntos (:) para la parte de la hora, genera una advertencia, como se muestra aquí:

```
mysql> SELECT TIMESTAMP '2012^12^31 11*30*45';
+-----+
| TIMESTAMP '2012^12^31 11*30*45' |
+-----+
| 2012-12-31 11:30:45 |
+-----+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
***** 1. row *****
  Level: Warning
    Code: 4095
Message: Delimiter '^' in position 4 in datetime value '2012^12^31 11*30*45' at
row 1 is deprecated. Prefer the standard '-'.
1 row in set (0.00 sec)
```

El único delimitador reconocido entre una parte de fecha y hora y una parte de segundos fraccionarios es el punto decimal.

Las partes de fecha y hora se pueden separar por T en lugar de un espacio. Por ejemplo, '2012-12-31 11:30:45' '2012-12-31T11:30:45' son equivalentes.

Anteriormente, MySQL admitía números arbitrarios de caracteres de espacio en blanco iniciales y finales en los valores de fecha y hora, así como entre las partes de fecha y hora de los valores DATETIME y TIMESTAMP. En MySQL 8.0.29 y versiones posteriores, este comportamiento está obsoleto y la presencia de un exceso de espacios en blanco activa una advertencia, como se muestra aquí:

```
mysql> SELECT TIMESTAMP '2012-12-31    11-30-45';
+-----+
| TIMESTAMP '2012-12-31    11-30-45' |
+-----+
| 2012-12-31 11:30:45 |
+-----+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
***** 1. row *****

```



```

Level: Warning
Code: 4096
Message: Delimiter '' in position 11 in datetime value '2012-12-31 11:30:45'
at row 1 is superfluous and is deprecated. Please remove.
1 row in set (0.00 sec)

```

También a partir de MySQL 8.0.29, se genera una advertencia cuando se utilizan caracteres de espacio en blanco que no sean el carácter de espacio, como este:

```

mysql> SELECT TIMESTAMP '2021-06-06
      '> 11:15:25';
+-----+
| TIMESTAMP'2021-06-06
11:15:25'           |
+-----+
| 2021-06-06 11:15:25 |
+-----+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
***** 1. row *****
Level: Warning
Code: 4095
Message: Delimiter '\n' in position 10 in datetime value '2021-06-06
11:15:25' at row 1 is deprecated. Prefer the standard ''.
1 row in set (0.00 sec)

```

Solo se genera una advertencia de este tipo por valor temporal, aunque pueden existir varios problemas con los delimitadores, los espacios en blanco o ambos, como se muestra en la siguiente serie de declaraciones:

```

mysql> SELECT TIMESTAMP '2012!-12-31 11:30:45';
+-----+
| TIMESTAMP'2012!-12-31 11:30:45' |
+-----+
| 2012-12-31 11:30:45           |
+-----+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
***** 1. row *****
Level: Warning
Code: 4095
Message: Delimiter '!' in position 4 in datetime value '2012!-12-31 11:30:45'
at row 1 is deprecated. Prefer the standard '-'.
1 row in set (0.00 sec)

mysql> SELECT TIMESTAMP '2012-12-31 11:30:45';
+-----+
| TIMESTAMP'2012-12-31 11:30:45' |
+-----+
| 2012-12-31 11:30:45           |
+-----+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
***** 1. row *****
Level: Warning
Code: 4096
Message: Delimiter '' in position 11 in datetime value '2012-12-31 11:30:45'
at row 1 is superfluous and is deprecated. Please remove.

```



1 row in set (0.00 sec)

```
mysql> SELECT TIMESTAMP '2012-12-31 11:30:45';
+-----+
| TIMESTAMP '2012-12-31 11:30:45' |
+-----+
| 2012-12-31 11:30:45           |
+-----+
1 row in set (0.00 sec)
```

- Como una cadena sin delimitadores en formato 'AAAAMMDDhhmmss' o 'AAMMDDhhmmss', siempre que la cadena tenga sentido como una fecha. Por ejemplo, '20070523091528' y '070523091528' se interpretan como '2007-05-23 09:15:28', pero '071122129015' es ilegal (tiene una parte de minutos sin sentido) y se convierte en '0000-00-00 00:00:00'.
- Como un número en formato AAAAMMDDhhmmss o AAAAMMDDhhmmss, siempre que el número tenga sentido como una fecha. Por ejemplo, 19830905132800 y 830905132800 se interpretan como '1983-09-05 13:28:00'.

Un valor DATETIME o TIMESTAMP puede incluir una parte final de segundos fraccionarios con una precisión de hasta microsegundos (6 dígitos). La parte fraccionaria siempre debe estar separada del resto del tiempo por un punto decimal; no se reconoce ningún otro delimitador de fracciones de segundo.

Las fechas que contienen valores de año de dos dígitos son ambiguas porque se desconoce el siglo. MySQL interpreta los valores de año de dos dígitos usando estas reglas:

- Los valores de año en el rango 70-99 se convierten en 1970-1999.
- Los valores de año en el rango 00-69 se convierten en 2000-2069.

Para valores especificados como cadenas que incluyen delimitadores de parte de fecha, no es necesario especificar dos dígitos para valores de mes o día que sean menores que 10. '2015-6-9' es lo mismo que '2015-06-09'. De manera similar, para valores especificados como cadenas que incluyen delimitadores de partes de tiempo, no es necesario especificar dos dígitos para valores de hora, minuto o segundo que sean menores que 10. '2015-10-30 1:2:3' es lo mismo que '2015-10-30 01:02:03'.

Los valores especificados como números deben tener 6, 8, 12 o 14 dígitos. Si un número tiene 8 o 14 dígitos, se supone que está en formato AAAAMMDD o AAAAMMDDhhmmss y que el año viene dado por los primeros 4 dígitos. Si el número tiene 6 o 12 dígitos, se supone que está en formato AAMMDD o AAMMDDhhmmss y que el año viene dado por los 2 primeros dígitos. Los números que no tienen una de estas longitudes se interpretan como rellenos con ceros a la izquierda hasta la longitud más cercana.

Los valores especificados como cadenas no delimitadas se interpretan según su longitud. Para una cadena de 8 o 14 caracteres, se supone que el año viene dado por los primeros 4 caracteres. De lo contrario, se supone que el año viene dado por los 2 primeros caracteres. La cadena se interpreta de izquierda a derecha para encontrar los valores de año, mes, día, hora, minuto y segundo, para tantas partes como estén presentes en la cadena. Esto significa que no debe usar cadenas que tengan menos de 6 caracteres. Por ejemplo, si especifica '9903' pensando que representa marzo de 1999, MySQL lo convierte al valor de fecha "cero". Esto ocurre porque los valores de año y mes son 99 y 03, pero falta completamente la parte del día. Sin embargo, puede especificar explícitamente un valor de cero para representar las partes del mes o del día que faltan. Por ejemplo, para insertar el valor '1999-03-00', use '990300'.

MySQL reconoce valores TIME en estos formatos:



- Como una cadena en formato 'D hh:mm:ss'. También puede utilizar una de las siguientes sintaxis "relajadas": 'hh:mm:ss', 'hh:mm', 'D hh:mm', 'D hh' o 'ss'. Aquí D representa días y puede tener un valor de 0 a 34.
- Como una cadena sin delimitadores en formato 'hhmmss', siempre que tenga sentido como tiempo. Por ejemplo, '101112' se entiende como '10:11:12', pero '109712' es ilegal (tiene una parte de minutos sin sentido) y se convierte en '00:00:00'.
- Como número en formato hhmmss, siempre que tenga sentido como hora. Por ejemplo, 101112 se entiende como '10:11:12'. También se entienden los siguientes formatos alternativos: ss, mmss o hhmmss.

Una parte de segundos fraccionarios finales se reconoce en los formatos de tiempo 'D hh:mm:ss.fracción', 'hh:mm:ss.fracción', 'hhmmss.fracción' y hhmmss.fracción, donde fracción es la parte fraccionaria en Precisión de hasta microsegundos (6 dígitos). La parte fraccionaria siempre debe estar separada del resto del tiempo por un punto decimal; no se reconoce ningún otro delimitador de fracciones de segundo.

Para valores TIME especificados como cadenas que incluyen un delimitador de parte de tiempo, no es necesario especificar dos dígitos para valores de horas, minutos o segundos que sean menores que 10. '8:3:2' es lo mismo que '08:03:02'.

A partir de MySQL 8.0.19, puede especificar un desplazamiento de zona horaria al insertar valores TIMESTAMP y DATETIME en una tabla. El desplazamiento se agrega a la parte de la hora de un literal de fecha y hora, sin espacios intermedios, y usa el mismo formato que se usa para configurar la variable del sistema time_zone, con las siguientes excepciones:

- Para valores de hora inferiores a 10, se requiere un cero inicial.
- Se rechaza el valor '-00:00'.
- No se pueden utilizar nombres de zonas horarias como 'EET' y 'Asia/Shanghai'; 'SYSTEM' tampoco se puede utilizar en este contexto.

El valor insertado no debe tener un cero para la parte del mes, la parte del día o ambas partes. Esto se aplica a partir de MySQL 8.0.22, independientemente de la configuración del modo SQL del servidor.

Este ejemplo ilustra la inserción de valores de fecha y hora con desplazamientos de zona horaria en las columnas TIMESTAMP y DATETIME usando diferentes configuraciones de zona horaria y luego recuperándolos:

```
mysql> CREATE TABLE ts (
->     id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
->     col TIMESTAMP NOT NULL
-> ) AUTO_INCREMENT = 1;

mysql> CREATE TABLE dt (
->     id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
->     col DATETIME NOT NULL
-> ) AUTO_INCREMENT = 1;

mysql> SET @@time_zone = 'SYSTEM';

mysql> INSERT INTO ts (col) VALUES ('2020-01-01 10:10:10'),
->     ('2020-01-01 10:10:10+05:30'), ('2020-01-01 10:10:10-08:00');

mysql> SET @@time_zone = '+00:00';

mysql> INSERT INTO ts (col) VALUES ('2020-01-01 10:10:10'),
->     ('2020-01-01 10:10:10+05:30'), ('2020-01-01 10:10:10-08:00');
```



```

mysql> SET @@time_zone = 'SYSTEM';

mysql> INSERT INTO dt (col) VALUES ('2020-01-01 10:10:10'),
->      ('2020-01-01 10:10:10+05:30'), ('2020-01-01 10:10:10-08:00');

mysql> SET @@time_zone = '+00:00';

mysql> INSERT INTO dt (col) VALUES ('2020-01-01 10:10:10'),
->      ('2020-01-01 10:10:10+05:30'), ('2020-01-01 10:10:10-08:00');

mysql> SET @@time_zone = 'SYSTEM';

mysql> SELECT @@system_time_zone;
+-----+
| @@system_time_zone |
+-----+
| EST                |
+-----+


mysql> SELECT col, UNIX_TIMESTAMP(col) FROM dt ORDER BY id;
+-----+-----+
| col           | UNIX_TIMESTAMP(col) |
+-----+-----+
| 2020-01-01 10:10:10 | 1577891410 |
| 2019-12-31 23:40:10 | 1577853610 |
| 2020-01-01 13:10:10 | 1577902210 |
| 2020-01-01 10:10:10 | 1577891410 |
| 2020-01-01 04:40:10 | 1577871610 |
| 2020-01-01 18:10:10 | 1577920210 |
+-----+-----+


mysql> SELECT col, UNIX_TIMESTAMP(col) FROM ts ORDER BY id;
+-----+-----+
| col           | UNIX_TIMESTAMP(col) |
+-----+-----+
| 2020-01-01 10:10:10 | 1577891410 |
| 2019-12-31 23:40:10 | 1577853610 |
| 2020-01-01 13:10:10 | 1577902210 |
| 2020-01-01 05:10:10 | 1577873410 |
| 2019-12-31 23:40:10 | 1577853610 |
| 2020-01-01 13:10:10 | 1577902210 |
+-----+-----+

```

El desplazamiento no se muestra al seleccionar un valor de fecha y hora, incluso si se utilizó uno al insertarlo.

El rango de valores de compensación admitidos es de -13:59 a +14:00, inclusive.

Los literales de fecha y hora que incluyen compensaciones de zona horaria se aceptan como valores de parámetro mediante declaraciones preparadas.



FUNCIONES DE FECHA Y HORA

Esta sección describe las funciones que se pueden usar para manipular valores temporales.

Nombre	Descripción
ADDDATE()	Agrega valores de tiempo (intervalos) a un valor de fecha
ADDTIME()	Agrega tiempo
CONVERT_TZ()	Convierte de una zona horaria a otra
CURDATE()	Devuelve la fecha actual
CURRENT_DATE(), CURRENT_DATE	Sinónimos para CURDATE()
CURRENT_TIME(), CURRENT_TIME	Sinónimos para CURTIME()
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP	Sinónimos para NOW()
CURTIME()	Devuelve la hora actual
DATE()	Extrae la parte de la fecha de una expresión de fecha o fecha y hora
DATE_ADD()	Agrega valores de tiempo (intervalos) a un valor de fecha
DATE_FORMAT()	Da formato a la fecha como se especifica
DATE_SUB()	Resta un valor de tiempo (intervalo) de una fecha
DATEDIFF()	Resta dos fechas
DAY()	Sinónimo para DAYOFMONTH()
DAYNAME()	Devuelve el nombre del día de la semana
DAYOFMONTH()	Devuelve el día del mes (0-31)
DAYOFWEEK()	Devuelve el índice del día de la semana del argumento.
DAYOFYEAR()	Devuelve el día del año (1-366)
EXTRACT()	Extrae parte de una fecha
FROM_DAYS()	Convierte un número de día en una fecha
FROM_UNIXTIME()	Formatear timestamp de Unix como una fecha
GET_FORMAT()	Devuelve una cadena de formato de fecha
HOUR()	Extrae la hora
LAST_DAY	Devuelve el último día del mes para el argumento.
LOCALTIME(), LOCALTIME	Sinónimo para NOW()
LOCALTIMESTAMP, LOCALTIMESTAMP()	Sinónimo para NOW()
MAKEDATE()	Crea una fecha a partir del año y el día del año.
MAKETIME()	Crear tiempo a partir de hora, minuto, segundo
MICROSECOND()	Devuelve los microsegundos del argumento.
MINUTE()	Devuelve el minuto del argumento.
MONTH()	Devuelve el mes a partir de la fecha pasada

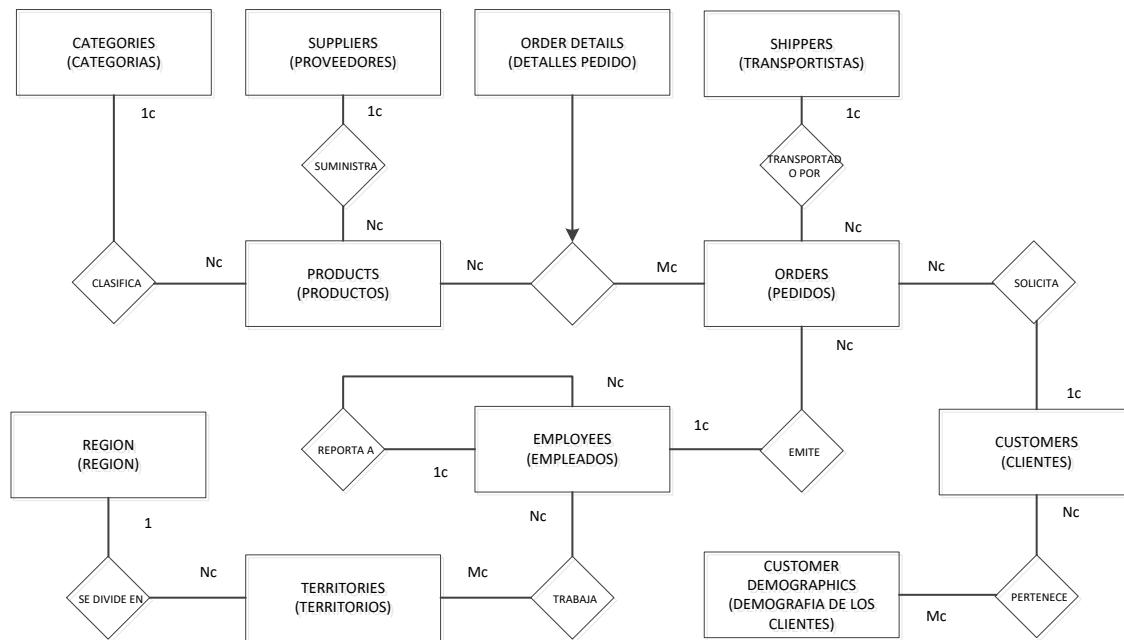


Nombre	Descripción
MONTHNAME()	Devuelve el nombre del mes.
NOW()	Devuelve la fecha y hora actual
PERIOD_ADD()	Agrega un período a un año-mes
PERIOD_DIFF()	Devuelve el número de meses entre periodos
QUARTER()	Devuelve el trimestre de un argumento de fecha
SEC_TO_TIME()	Convierte segundos al formato 'hh:mm:ss'
SECOND()	Devuelve el segundo (0-59)
STR_TO_DATE()	Convertir una cadena en una fecha
SUBDATE()	Sinónimo para DATE_SUB() cuando es invocado con tres argumentos
SUBTIME()	Resta tiempos
SYSDATE()	Devuelve la hora a la que se ejecuta la función.
TIME()	Extrae la porción de tiempo de la expresión pasada
TIME_FORMAT()	Formatea como tiempo
TIME_TO_SEC()	Devuelve el argumento convertido a segundos.
TIMEDIFF()	Resta tiempo
TIMESTAMP()	Con un solo argumento, esta función devuelve la expresión de fecha o fecha y hora; con dos argumentos, la suma de los argumentos
TIMESTAMPADD()	Agrega un intervalo a una expresión de fecha y hora
TIMESTAMPDIFF()	Resta un intervalo de una expresión de fecha y hora
TO_DAYS()	Devuelve el argumento de fecha convertido a días.
TO_SECONDS()	Devuelve el argumento de fecha o fecha y hora convertido a segundos desde el año 0
UNIX_TIMESTAMP()	Devuelve un timestamp de Unix
UTC_DATE()	Devuelve la fecha UTC actual
UTC_TIME()	Devuelve la hora UTC actual
UTC_TIMESTAMP()	Devuelve la fecha y hora UTC actual
WEEK()	Devuelve el número de semana
WEEKDAY()	Devuelve el índice del día de la semana
WEEKOFYEAR()	Devuelve la semana natural de la fecha (1-53)
YEAR()	Devuelve el año
YEARWEEK()	Devuelve el año y la semana.



ANEXO: BASE NORTHWIND

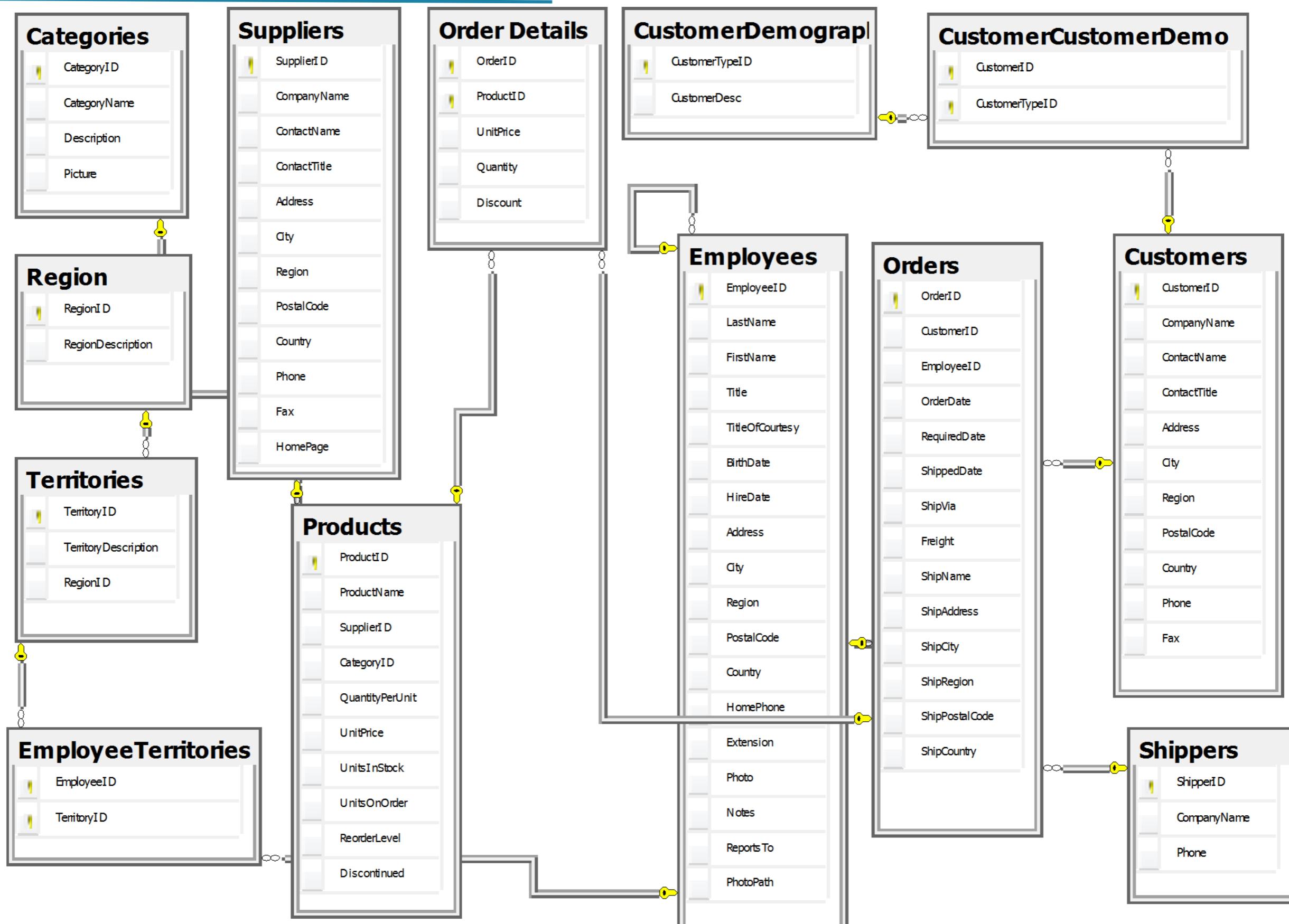
DER

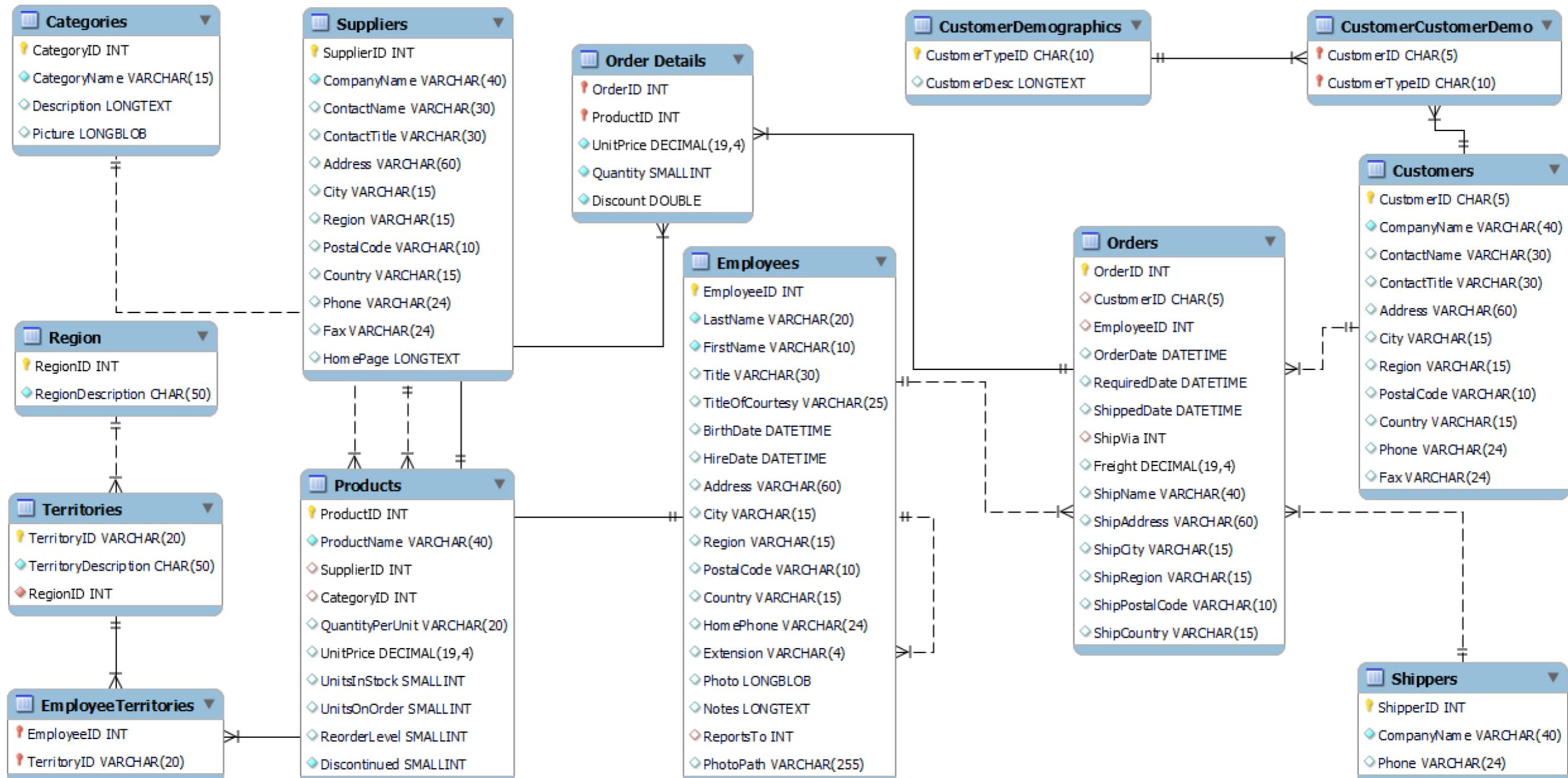


DICCIONARIO DE DATOS

CATEGORIES	=	@CategoryID + CategoryName + Description + Picture
CUSTOMERDEMOGRAPHICS	=	@CustomerTypeID + CustomerDesc
CUSTOMERS	=	@CustomerID + CompanyName + ContactName + ContactTitle + Address + City + Region + PostalCode + Country + Phone + Fax
EMPLOYEES	=	@EmployeeID + LastName + FirstName + Title + TitleOfCourtesy + BirthDate + HireDate + Address + City + Region + PostalCode + Country + HomePhone + Extension + Photo + Notes + Photopath
ORDER DETAILS	=	UnitPrice + Quantity + Discount
ORDERS	=	@OrderID + OrderDate + RequiredDate + ShippedDate + Freight + ShipName + ShipAddress + ShipCity + ShipRegion + ShipPostalCode + ShipCountry
PRODUCTS	=	@ProductID + ProductName + QuantityPerUnit + UnitPrice + UnitsInStock + UnitsOnOrder + ReorderLevel + Discontinued
REGION	=	@RegionID + RegionDescription
SHIPPERS	=	@ShipperID + CompanyName + Phone
SUPPLIERS	=	@SupplierID + CompanyName + ContactName + ContactTitle + Address + City + Region + PostalCode + Country + Phone + Fax + HomePage
TERRITORIES	=	@TerritoryID + TerritoryDescription







2 ER Base Northwind - MySQL Workbench



CONTENIDO

Introducción a las Bases de Datos	2
Concepto y origen de las Bases de Datos y de los SGBD	2
Los archivos tradicionales y las Bases de Datos.....	3
Evolución de los SGBD	3
Los años sesenta y setenta: sistemas centralizados	3
Los años ochenta: SGBD relacionales	4
Los años noventa: distribución, C/S y 4GL.....	4
Tendencias actuales.....	7
Objetivos y servicios de los SGBD.....	8
Consultas no predefinidas y complejas	8
Flexibilidad e independencia	9
Problemas de la redundancia	10
Integridad de los datos	11
Concurrencia de usuarios	12
Seguridad	13
Otros objetivos	14
Arquitectura de los SGBD	14
Esquemas y niveles	14
Independencia de los datos	17
Flujo de datos y de control	19
Modelos de Bases de Datos	20
Evolución de los modelos de Base de Datos.....	21
Lenguajes y usuarios.....	22
Administración de Bases de Datos	24
Resumen.....	26
Modelo Entidad Relación.....	27
Objetos básicos del modelo E-R	27
Entidades y conjunto de entidades.....	27
Relaciones y conjunto de relaciones.....	27
Cardinalidades de mapeo	28
Claves primarias.....	30
Metodología Estructurada	31
Esquema de Datos.....	31
Objetivo:	31



Herramientas:	32
Técnicas:	32
Diagrama de Entidad Relación (DER).....	34
Introducción.....	34
Convenciones.....	34
Misceláneas	38
Normalización en la construcción del Esquema de Datos.....	38
Ejemplo	39
Anexo A: Técnica de Preguntas y Respuestas.....	40
Ejemplo:.....	40
Diccionario de Datos	40
Diccionario de Datos	41
Anexo B: Ciclo de Vida Centrado en los Datos.....	43
El modelo relacional	44
Introducción	44
Objetivos	44
Introducción al modelo relacional.....	44
Estructura de los datos.....	45
Visión informal de una relación	45
Visión formal de una relación	46
Diferencias entre relaciones y ficheros.....	49
Clave candidata, clave primaria y clave alternativa de las relaciones	50
Claves foráneas de las relaciones	51
Creación de las relaciones de una base de datos	54
Operaciones del modelo relacional.....	54
Reglas de integridad	55
Regla de integridad de unicidad de la clave primaria	56
Regla de integridad de entidad de la clave primaria	57
Regla de integridad referencial.....	58
Referencia incorrecta.....	59
Regla de integridad de dominio.....	66
Esquema Lógico de Base de Datos	68
Objetivo:	68
Herramientas	68
Técnicas	68



Construcción del Esquema Lógico de Base de Datos.....	69
Diccionario de Datos	71
Normalización	72
Breve reseña histórica	72
Definición:	72
Teoría de la normalización	72
Conservación de la información	72
Conservación de las dependencias	73
Dependencias	73
Definición de DF:.....	74
Descriptores equivalentes:	74
Objetivos de la normalización:	74
Ejemplo.....	75
1FN – 1º Forma Normal:.....	76
2FN – 2º Forma Normal	77
3FN– 3º Forma Normal	78
El álgebra relacional.....	79
Introducción	79
Objetivos	79
Introducción al Álgebra Relacional.....	80
Operaciones conjuntistas	83
Operaciones específicamente relacionales	87
Secuencias de operaciones del álgebra relacional	92
Extensiones: combinaciones externas	93
Operación división.....	96
Resumen.....	97
SQL.....	98
Historia de Microsoft SQL Server.....	99
SQL Server 2005	100
SQL Server 2008	100
SQL Server 2008 R2	100
SQL Server 2012	100
SQL Server 2014	101
SQL Server 2016	101
SQL Server 2017	101



SQL Server 2019	101
SQL Server 2022	101
Ediciones Principales.....	101
Tipos de Sistemas de Bases de Datos	102
Arquitectura de SQL Server	103
Alternativas ABC.....	103
Instancias de SQL Server	104
Bases de Datos	104
Esquemas y objetos.....	106
MySQL.....	106
¿Qué es MySQL?.....	106
MySQL es un sistema de gestión de bases de datos.....	106
Las bases de datos MySQL son relacionales.....	107
El software MySQL es de código abierto.	107
El servidor de base de datos MySQL es muy rápido, confiable, escalable y fácil de usar.....	107
MySQL Server funciona en sistemas cliente/servidor o integrados.	107
Hay disponible una gran cantidad de software MySQL contribuido.	108
Las características principales de MySQL	108
Internos y Portabilidad	108
Tipos de datos.....	109
Declaraciones y funciones	109
Seguridad	109
Escalabilidad y límites	109
Conectividad	109
Localización.....	110
Clientes y Herramientas.....	110
Esquemas	111
Categorías de las instrucciones SQL.....	111
Restricciones (Constraints)	111
Null y la lógica de tres valores	112
Tipos de datos en SQL.....	112
Categorías de Tipos de Datos	113
Números Exactos.....	113
Enteros.....	113
Tipos de Punto Fijo	114



Números aproximados	114
Fecha y Hora.....	114
Cadenas de caracteres	115
Cadenas de caracteres Unicode	115
Cadenas Binarias	115
Otros Tipos de Datos	116
Aclaraciones básicas: Comentarios.....	116
Espacios en blanco y el punto y coma	116
Sensibilidad a Mayúsculas y minúsculas	116
DDL.....	117
CREATE TABLE	117
DROP TABLE.....	123
ALTER TABLE	123
Para agregar una columna	127
Para eliminar una columna	127
Para modificar una columna	127
DML.....	128
Seleccionando todas las columnas y todas las filas.....	128
Sintaxis	128
Ejemplo	128
Seleccionando columnas específicas.....	128
Sintaxis	128
Ejemplo	129
Ordenando registros	129
Ordenando por una columna - Sintaxis	129
Ejemplo	129
Ordenar por varias columnas - Sintaxis	130
Ejemplo	130
Ordenar por posición de la columna - Sintaxis	130
Ejemplo	130
Orden Ascendente y Descendente.....	131
Sintaxis	131
Ejemplo	131
La cláusula WHERE y los símbolos operadores	131
Sintaxis	131



Verificando por igualdad - Ejemplo	132
Verificando por no igualdad – Ejemplo.....	132
Verificando por mayor o menor que	132
Verificando por NULL.....	132
Ejemplo	133
Combinando WHERE y ORDER BY	133
Ejemplo	133
La cláusula WHERE y las palabras operadores	133
Ejemplos	134
El operador LIKE	134
Ejemplos	135
El operador NOT	135
Verificando múltiples condiciones	136
AND	136
OR	136
Orden de evaluación	137
Campos calculados	137
Concatenación	137
Sintaxis.....	138
Ejemplo	138
Cálculos matemáticos.....	139
Alias	139
Funciones de agregación y agrupamiento.....	140
Ejemplos	141
Agrupando Datos.....	141
Having	142
Ejemplo	142
Orden de las cláusulas.....	142
Ejemplo	142
Reglas de Agrupamiento	142
Seleccionando registros sin repetición.....	143
Ejemplo	143
Ejemplo	143
Funciones de manipulación incluidas.....	143
Ejemplo	144



Funciones de manipulación de cadenas.....	145
Ejemplo	145
Funciones de Fechas.....	146
Ejemplo	146
Subconsultas.....	148
Clasificación de las subconsultas.....	149
Subconsultas Autocontenido.....	149
Subconsultas Correlacionadas	151
EXISTS	152
JOINS (Juntar)	153
CROSS JOIN	153
Sintaxis.....	154
INNER JOIN	154
Sintaxis.....	154
OUTER JOIN	156
Sintaxis.....	156
Ejemplo	157
SELF JOIN	158
EQUI Y NONEQUI JOINS.....	159
CONSULTAS MULTI-JOIN	159
Sintaxis.....	159
Ejemplo	160
Operadores UNION, EXCEPT e INTERSECT.....	161
El operador UNION.....	162
El operador UNION ALL.....	162
El operador UNION (DISTINCT)	163
El operador INTERSECT.....	163
El operador INTERSECT (DISTINCT)	163
El operador EXCEPT	164
El operador EXCEPT (DISTINCT)	164
Precedencia	165
Reglas de LOS OPERADORES DE CONJUNTOS	165
Rutinas	166
Funciones definidas por el usuario.....	166
Procedimientos almacenados (Stored Procedures)	169



Disparadores (Triggers)	172
Disparadores DML en SQL Server	173
Disparadores DML en MySQL	174
Programando.....	180
Variables	180
Elementos de Flujo	182
Transacciones y Conurrencia	188
Cursos	190
Tablas Temporales	193
Manejo de errores en SQL Server	198
Manejo de errores en MySQL.....	202
Anexo Operadores y funciones.....	206
Concatenación de cadenas (operador + y función CONCAT)	206
Funciones de Cadena	208
ANEXO: Trabajando con datos de fecha y hora en SQL SERVER.....	217
Literales	217
Trabajando con fecha y hora por separado.....	219
Filtrando rangos de fechas	220
Funciones de Fecha y Hora.....	221
Obtener la fecha y hora actual	221
Funciones CAST, CONVERT y PARSE y sus contrapartes TRY_	222
Función DATEPART	222
Funciones YEAR, MONTH y DAY	223
Función DATENAME.....	223
Funciones FROMPARTS.....	224
Función SWITCHOFFSET.....	224
Función TODATETIMEOFFSET	225
Función AT TIME ZONE	225
Función DATEADD.....	226
Funciones DATEDIFF y DATEDIFF_BIG	227
Función ISDATE	227
Función EOMONTH	228
ANEXO: Trabajando con datos de fecha y hora en MySQL.....	229
Literales estándar de fecha y hora de SQL y ODBC	229
Literales de cadena y numéricos en contexto de fecha y hora	229



Funciones de Fecha y Hora.....	235
ANEXO: BASE Northwind	237
DER	237
Diccionario de Datos.....	237

