Software libre

Rafael Camps Paré
Luis Alberto Casillas Santillán
Dolors Costal Costa
Marc Gibert Ginestà
Carme Martín Escofet
Oscar Pérez Mora

71Z799014MO

Bases de datos

UOC
Formación de Posgrado

David Megías Jiménez

Coordinador

Ingeniero de Informática por la UAB.
Magíster en Técnicas Avanzadas de Automatización de Procesos por la UAB.
Doctor en Informática por la UAB.
Profesor de los Estudios de Informática y Multimedia de la UOC.

Jordi Mas Hernández

Coordinador

Coordinador general de Softcatalà y desarrollador del procesador de textos libre Abiword.
Miembro fundador de Softcatalà y de la red telemática RedBBS.
En calidad de consultor, ha trabajado en empresas como Menta, Telépolis, Vodafone, Lotus, eresMas, Amena y Terra España.

Rafael Camps Paré

Autor

Profesional informático en varias empresas. Ha sido profesor universitario en la Facultad de Informática de Barcelona de la Universidad Politécnica de Cataluña. Actualmente está adscrito a la Escuela Universitaria Politécnica de Vilanova i la Geltrú.

Luis Alberto Casillas Santillán

Autor

Licenciado en Informática (1995), maestro en Sistemas (1998), doctorante en Ingeniería y Tecnología (2003), está estudiando el doctorado en la UOC. Profesor universitario desde marzo de 1995. Investigador en Inteligencia Artificial desde 1998. Consultor Universitario para cuestiones educativas.

Dolors Costal Costa

Autora

Doctora en Informática por la Universidad Politécnica de Cataluña. Profesora titular del Departamento de Lenguajes y Sistemas Informáticos de la Universidad Politécnica de Cataluña, asignada a la Facultad de Informática de Barcelona.

Marc Gibert Ginestà

Autor

Ingeniero en Informática por la Universidad Ramon Llull. Socio fundador y jefe de proyectos de Cometa Technologies, empresa dedicada a dar soluciones en tecnologías de la información, basadas en el uso de estándares y herramientas de código abierto. Profesor del Máster en Seguridad en Tecnologías de la Información en Enginyeria y Arquitectura La Salle y consultor del Master Internacional en Software Libre de la UOC.

Carme Martín Escofet

Autora

Licenciada en Informática por la Universidad Politécnica de Cataluña. Profesora de la asignatura Introducción a las bases de datos en la Facultad de Informática de Barcelona, y de la asignatura Sistemas orientados a bases de datos en la Facultad de Matemáticas y Estadística. También ha sido profesora de las asignaturas Diseño de sistemas y Gestión de sistemas informáticos en la Escuela Universitaria Politécnica de Vilanova i la Geltrú.

Oscar Pérez Mora

Autor

Ingeniero en Comunicaciones y Electrónica por la Universidad de Guadalajara (México) y Maestro en Sistemas de Información. Ha participado en diversas publicaciones e impartido cursos especializados. Miembro Fundador del Marichi del SUTUdeG y del Grupo Linux de Occidente A. C. (www.glo.org.mx). Organizador del Festival GNU/Linux y Software Libre (http://www.festivaldesoftwarelibre.org).

Primera edición: mayo 2005

Av. Tibidabo, 39-43, 08035 Barcelona
Material realizado por Eureca Media, SL

Depósito legal: B-15.562-2005
ISBN: 84-9788-269-5

Agradecimientos

Los autores agradecen a la Fundación para la Universitat Oberta de Catalunya (http://www.uoc.edu) la financiación de la primera edición de esta obra, enmarcada en el Máster Internacional en Software Libre ofrecido por la citada institución.

El autor Óscar Pérez Mora desea hacer constar el agradecimiento siguiente: "A José Pérez Arias, gracias por haber sido mi padre", en memoria de su padre fallecido durante la realización de este material.

Introducción

Las bases de datos son el método preferido para el almacenamiento estructurado de datos. Desde las grandes aplicaciones multiusuario, hasta los teléfonos móviles y las agendas electrónicas utilizan tecnología de bases de datos para asegurar la integridad de los datos y facilitar la labor tanto de usuarios como de los programadores que las desarrollaron.

Desde la realización del primer modelo de datos, pasando por la administración del sistema gestor, hasta llegar al desarrollo de la aplicación, los conceptos y la tecnología asociados son muchos y muy heterogéneos. Sin embargo, es imprescindible conocer los aspectos clave de cada uno de estos temas para tener éxito en cualquier proyecto que implique trabajar con bases de datos.

En este curso trataremos de dar una visión completa de los conceptos relacionados con los sistemas gestores de bases de datos. En los primeros capítulos veremos los aspectos involucrados en los motores de estos almacenes de datos, su evolución histórica y los distintos tipos de organización y abs-tracción que han ido surgiendo desde su conceptualización hasta nuestros días.

A continuación profundizaremos en el modelo llamado relacional (el más usado en nuestros días), proporcionando los métodos y herramientas que nos permitan representar necesidades de almacenamiento y consulta de datos en este modelo. En el siguiente capítulo estudiaremos a fondo el lenguaje de consultas estructurado SQL, imprescindible para trabajar con bases de datos relacionales, ya sea directamente o a través de cualquier lenguaje de programación.

El diseño de bases de datos tiene también un capítulo dedicado a aprender a modelar y representar gráficamente una base de datos, a detectar los posibles problemas de diseño antes de que éstos afecten a la aplicación, y a construir bases de datos óptimas para los distintos casos de relaciones entre entidades que formarán nuestra base de datos.

Una vez sentadas estas bases, estaremos en disposición de examinar detenidamente dos de los sistemas gestores de bases de datos de software libre más usados y populares actualmente. Así pues, aprenderemos el uso, administración y particularidades de MySQL y PostgreSQL mediante ejemplos y casos prácticos. También veremos las herramientas de consulta y administración gráficas para estos dos sistemas gestores de bases de datos, que nos permitirán mejorar nuestra productividad en el trabajo diario con ellos.

A continuación, veremos los métodos de acceso a estos sistemas gestores de bases de datos desde algunos lenguajes de programación. En cada caso, comentaremos

las mejores prácticas para cada uno, así como sus particularidades, e introduciremos algunos conceptos relacionados con la programación en bases de datos como la persistencia, tratamiento de errores, etc.

Finalmente, creemos que la mejor forma de finalizar este curso es mediante un caso de estudio completo que aborde los temas tratados en todos los capítulos, en el orden en que se producirían en la realidad de un proyecto y primando la práctica sobre la teoría.

Así pues, este curso trata de proporcionar al estudiante una visión completa de los aspectos implicados en el trabajo con bases de datos. Aunque no profundizaremos en algunos temas muy especializados como el diseño interno de un sistema gestor de bases de datos, profundizaremos técnicamente en los temas más necesarios.

Objetivos

Los objetivos que el estudiante deberá alcanzar al finalizar el curso *Bases de datos* son los siguientes:

- Comprender los diferentes modelos de bases de datos, y en concreto dominar el modelo relacional.

- Ser capaz de realizar el modelo de una base de datos relacional, a partir de la especificación de requerimientos de un proyecto, comprendiendo y aplicando los conceptos y transformaciones implicados.

- Conocer el uso y administración de dos de los gestores de bases de datos relacionales más populares en el ámbito del software libre: PostgreSQL y MySQL.

- Tener experiencia en el desarrollo de aplicaciones en conexión con bases de datos en varios lenguajes.

Contenidos

4. Consultas
5. Administración de MySQL
6. Clientes gráficos

Módulo didáctico 6

Bases de datos en PostgreSQL

Marc Gibert Ginestà; Oscar Pérez Mora

1. Características de PostgreSQL
2. Introducción a la orientación a objetos
3. Acceso a un servidor PostgreSQL
4. Creación y manipulación de tablas
5. Manipulación de datos
6. Funciones y disparadores
7. Administración de PostgreSQL
8. Cliente gráfico: pgAdmin3

Módulo didáctico 7

Desarrollo de aplicaciones en conexión con bases de datos

Marc Gibert Ginestà

1. Conexión y uso de bases de datos en lenguaje PHP
2. Conexión y uso de bases de datos en lenguaje Java

Módulo didáctico 8

Caso de estudio

Marc Gibert Ginestà

1. Presentación del caso de estudio
2. El modelo relacional y el álgebra relacional
3. El lenguaje SQL
4. Introducción al diseño de bases de datos
5. Bases de datos en MySQL
6. Bases de datos en PostgreSQL
7. Desarrollo de aplicaciones en conexión con bases de datos

Apéndice

GNU Free Documentation License

Introducción a las bases de datos

Rafael Camps Paré

UOC
Formación de Posgrado

Índice

Introducción a las bases de datos

Introducción

Empezaremos esta unidad didáctica viendo cuáles son los **objetivos de los sistemas de gestión de las bases de datos (SGBD)** y, a continuación, daremos una visión general de la **arquitectura**, el **funcionamiento** y el **entorno** de estos sistemas.

Objetivos

En los materiales didácticos de esta unidad encontraréis las herramientas para adquirir una visión global del mundo de las BD y de los SGBD, así como para alcanzar los siguientes objetivos:

1. Conocer a grandes rasgos la evolución de los SGBD desde los años sesenta hasta la actualidad.

2. Distinguir los principales objetivos de los SGBD actuales y contrastarlos con los sistemas de ficheros tradicionales.

3. Saber explicar mediante ejemplos los problemas que intenta resolver el concepto de *transacción*.

4. Relacionar la idea de flexibilidad en los cambios con la independencia lógica y física de los datos, así como con la arquitectura de tres niveles.

5. Distinguir los principales modelos de BD.

6. Conocer a grandes rasgos el funcionamiento de un SGBD.

7. Saber relacionar los diferentes tipos de lenguajes con los diferentes tipos de usuarios.

1. Concepto y origen de las BD y de los SGBD

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

Aplicaciones informáticas de los años sesenta

La emisión de facturas, el control de pedidos pendientes de servir, el mantenimiento del fichero de productos o la nómina del personal eran algunas de las aplicaciones informáticas habituales en los años sesenta.

Cada aplicación (una o varias cadenas de programas) utilizaba ficheros de movimientos para actualizar (creando una copia nueva) y/o para consultar uno o dos ficheros maestros o, excepcionalmente, más de dos. Cada programa trataba como máximo un fichero 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 fichero nuevo con todos los datos necesarios (algo que provocaba redundancia) para evitar que los programas tuviesen que leer muchos ficheros.

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 ficheros *on-line* y de forma simultánea. Más adelante fue surgiendo la necesidad de hacer las actualizaciones también *on-line*.

Integración de aplicaciones

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

A medida que se integraban las aplicaciones, se tuvieron que interrelacionar sus ficheros y fue necesario eliminar la redundancia. El nuevo conjunto de ficheros 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 ficheros 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 ficheros 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 ficheros** 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 ficheros 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 BD (SGBD)**.

Con todo lo que hemos dicho hasta ahora, podríamos definir el término *BD*; una **base de datos de un SI** es la representación integrada de los conjuntos de entidades instancia correspondientes a las diferentes entidades tipo del SI 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 ficheros tradicionales y las BD

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

1) Entidades tipos:

- Ficheros: tienen registros de una sola entidad tipo.
- BD: tienen datos de varias entidades tipo.

2) Interrelaciones:

- Ficheros: el sistema no interrelaciona ficheros.
- BD: el sistema tiene previstas herramientas para interrelacionar entidades.

3) Redundancia:

- Ficheros: se crean ficheros a la medida de cada aplicación, con todos los datos necesarios aunque algunos sean redundantes respecto de otros ficheros.
- BD: todas las aplicaciones trabajan con la misma BD y la integración de los datos es básica, de modo que se evita la redundancia.

4) Usuarios

- Ficheros: 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.

2. Evolución de los SGBD

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

2.1. 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: un gran ordenador 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 al ordenador 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.

El *Data Base / Data Comunications*

IBM denominaba *Data Base/ Data Comunications* (DB/DC) el *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.

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 BD 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 BD. La preocupación básica era maximizar el rendimiento: el tiempo de respuesta y las transacciones por segundo.

2.2. Los años ochenta: SGBD relacionales

Los **ordenadores *minis***, en primer lugar, y después los **ordenadores *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 relacionales*** supone un avance importante para facilitar la programación de aplicaciones con BD y para conseguir que los programas sean independientes de los aspectos físicos de la BD.

*** Oracle aparece en el año 1980.**

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.

Los ordenadores personales

Durante los años ochenta aparecen y se extienden muy rápidamente los ordenadores 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*. Notad 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.

2.3. 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 prerrelacionales.

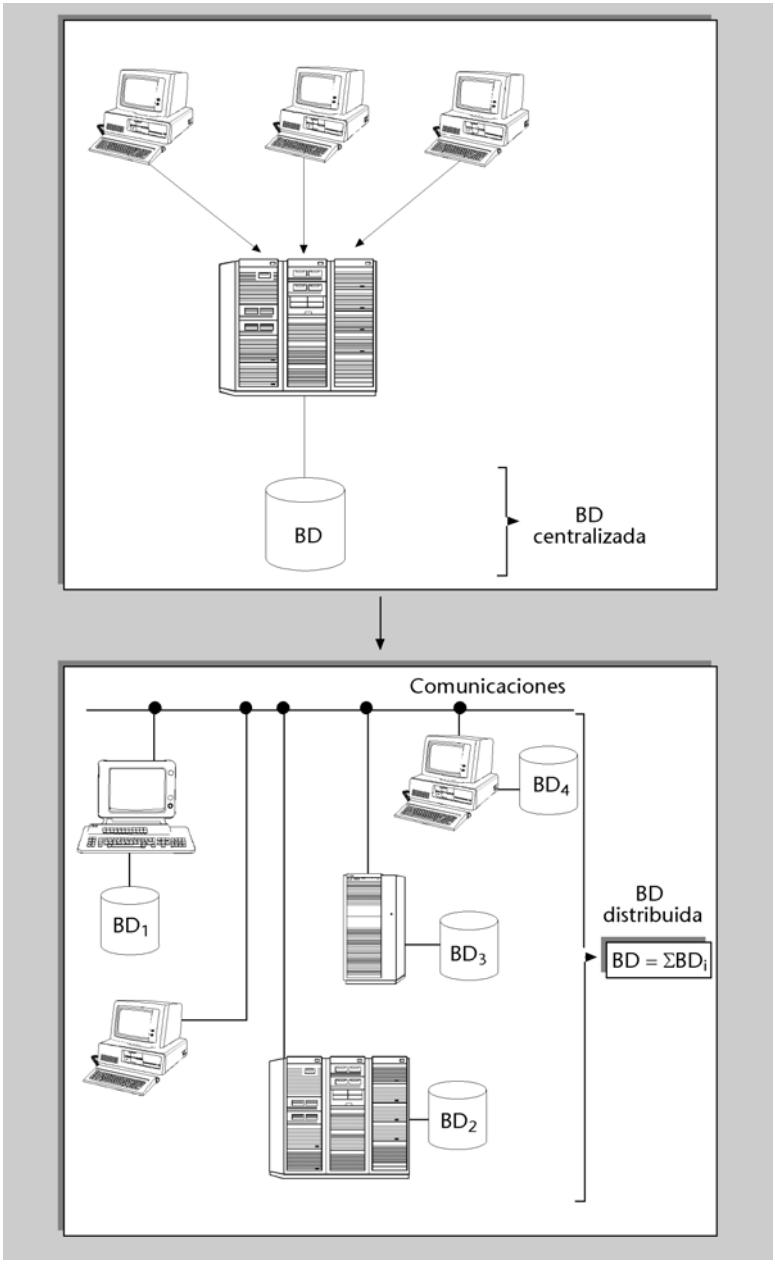
A finales de los ochenta y principios de los noventa, las empresas se han encontrado con el hecho de que sus departamentos han ido comprando ordenadores departamentales y personales, y han ido haciendo aplicaciones con BD. El resultado ha sido que en el seno de la empresa hay numerosas BD y varios SGBD de diferentes tipos o proveedores. Este fenómeno de **multiplicación de las BD 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 BD diferentes, junto con la facilidad que dan las redes para la intercomunicación entre ordenadores, ha conducido a los SGBD actuales, que permiten que un programa pueda trabajar con diferentes BD 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 BD son soportadas por una misma marca de SGBD, es decir, cuando hay homogeneidad. Sin em-

bargo, 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 BD.

Figura 1



Además de esta distribución "impuesta", al querer tratar de forma integrada distintas BD preexistentes, también se puede hacer una distribución "deseada", diseñando una BD 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 BD distribuida puede ser más alta, porque si queda fuera de servicio uno de los sistemas, los de-

más 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) Coste. Una BD distribuida puede reducir el coste. 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 coste total de las comunicaciones se puede reducir haciendo que un usuario tenga más cerca los datos que utiliza con mayor frecuencia; por ejemplo, en un ordenador de su propia oficina o, incluso, en su ordenador 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.

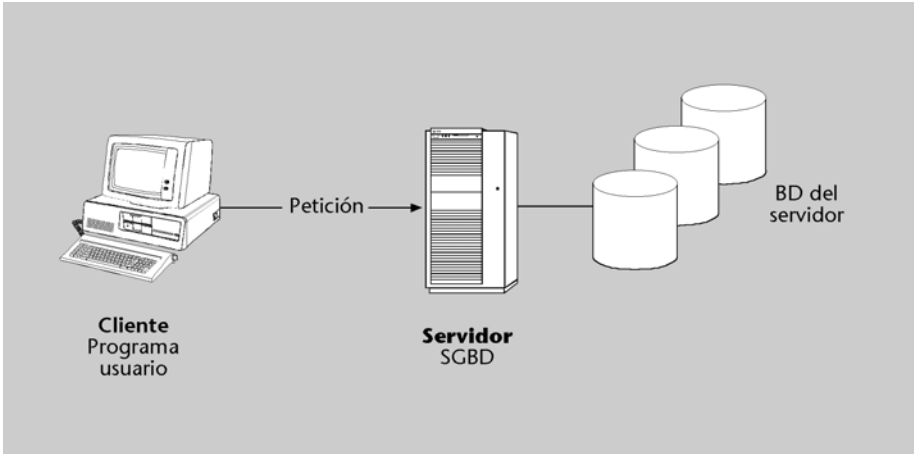
Por ejemplo, un programa de aplicación que un usuario ejecuta en su PC (que está conectado a una red) pide ciertos datos de una BD que reside en un equipo UNIX donde, a su vez, se ejecuta el SGBD relacional que la gestiona. El programa de aplicación es el cliente y el SGBD es el servidor.

Otros servicios

Notad que el servicio que da un servidor de un sistema C/S no tiene por qué estar relacionado con las BD; 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.

Figura 2



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 BD, 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 BD. 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.

2.4. Tendencias actuales

Hoy día, los SGBD relacionales 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 relacionales de los años ochenta y noventa son muy limitados. La **incorporación de tecnologías multimedia** –imagen y sonido– en los SI hace necesario que los SGBD relacionales acepten atributos de estos tipos.

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.

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 librerías de clases.

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

La rápida adopción de la **web** a los SI hace que los SGBD incorporen recursos para ser servidores de páginas web, como por ejemplo la inclusión de SQL en guiones HTML, SQL incorporado en Java, etc. Notad 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 BD 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 BD de distintas aplicaciones, las empresas han ido acumulando gran cantidad de datos de todo tipo. Si estos datos se analizan convenientemente pueden dar información valiosa*.

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

Por lo tanto, se trata de mantener una gran BD 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 las que hay en las BD 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 estudios* los analistas financieros, los analistas de mercado, etc.

*** Con frecuencia se trata de estadísticas multidimensionales.**

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.

3. 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. A continuación los comentaremos, pero sin entrar en detalles que ya se verán más adelante en otras asignaturas.

3.1. 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.

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.

Ficheros tradicionales

En los ficheros tradicionales, cada vez que se quería hacer una consulta se tenía que escribir un programa a medida.

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 explicaremos en otro módulo didáctico.

3.2. Flexibilidad e independencia

La complejidad de las BD y la necesidad de irlas adaptando a la evolución del SI 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 BD, 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 BD 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 ficheros ya había independencia física en un cierto grado, pero en el mundo de las BD 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 BD (por ejemplo, cuando se añaden/suprimen entidades o interrelaciones, atributos, etc.

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

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*.

Las diferentes visiones lógicas de una BD se verán en el apartado 4 de esta unidad didáctica.

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 ficheros 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.

3.3. Problemas de la redundancia

En el mundo de los ficheros tradicionales, cada aplicación utilizaba su fichero. Sin embargo, puesto que se daba mucha coincidencia de datos entre aplicacio-

nes, se producía también mucha redundancia entre los ficheros. Ya hemos dicho que uno de los objetivos de los SGBD es **facilitar la eliminación de la redundancia**.

Consultad el apartado 1 de esta unidad didáctica.

Seguramente pensáis 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 apuntado 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.

¿Por qué queremos evitar la redundancia? ¿Qué problema nos comporta?

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 BD. 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.

Para recordar lo que se ha dicho sobre datos replicados, consultad el subapartado 2.3 de esta unidad didáctica.

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 BD datos derivados (o calculados) a partir de otros datos de la misma BD. De este modo podemos responder rápidamente a consultas globales, ya que nos ahorramos la lectura de gran cantidad de registros.

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.

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.

3.4. 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 podremos decir al SGBD que nos lleve el control de las actualizaciones en el caso de las redundancias, para garantizar la integridad. Del mismo modo, podremos darle otras **reglas de integridad** –o restricciones– para que asegure que los programas las cumplen cuando efectúan las actualizaciones.

Reglas de integridad

Por ejemplo, podremos declarar que el atributo *DNI* debe ser clave o que la *fecha de nacimiento* debe ser una fecha correcta y, además, se debe cumplir que el alumno no pueda tener menos de dieciocho años ni más de noventa y nueve, o que el número de alumnos matriculados de una asignatura no sea superior a veintisiete, etc.

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

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

Aparte de las reglas de integridad que el diseñador de la BD 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.

Reglas de integridad del modelo

Por ejemplo, un SGBD relacional nunca aceptará que una tabla tenga filas duplicadas, un SGBD jerárquico nunca aceptará que una entidad tipo esté definida como hija de dos entidades tipo diferentes, etc.

Los **procesos de restauración** (*restore* o *recovery*) de los que todo SGBD dispone pueden reconstruir la BD 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 BD.

3.5. Concurrencia de usuarios

Usuarios concurrentes

Actualmente ya no son raros los SI que tienen, en un instante determinado, miles de sesiones de usuario abiertas simultáneamente. No hace falta pensar en los típicos sistemas de los consorcios de compañías aéreas o casos similares, sino en los servidores de páginas web.

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

Cuando los accesos concurrentes son todos de lectura (es decir, cuando la BD 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 BD.

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

Denominamos **transacción de BD** 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 BD quedaría en un estado incorrecto. Queremos que la ejecución de este programa sea tratada por el SGBD como una transacción de BD.

2) Otro ejemplo de programa que querríamos 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 BD 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

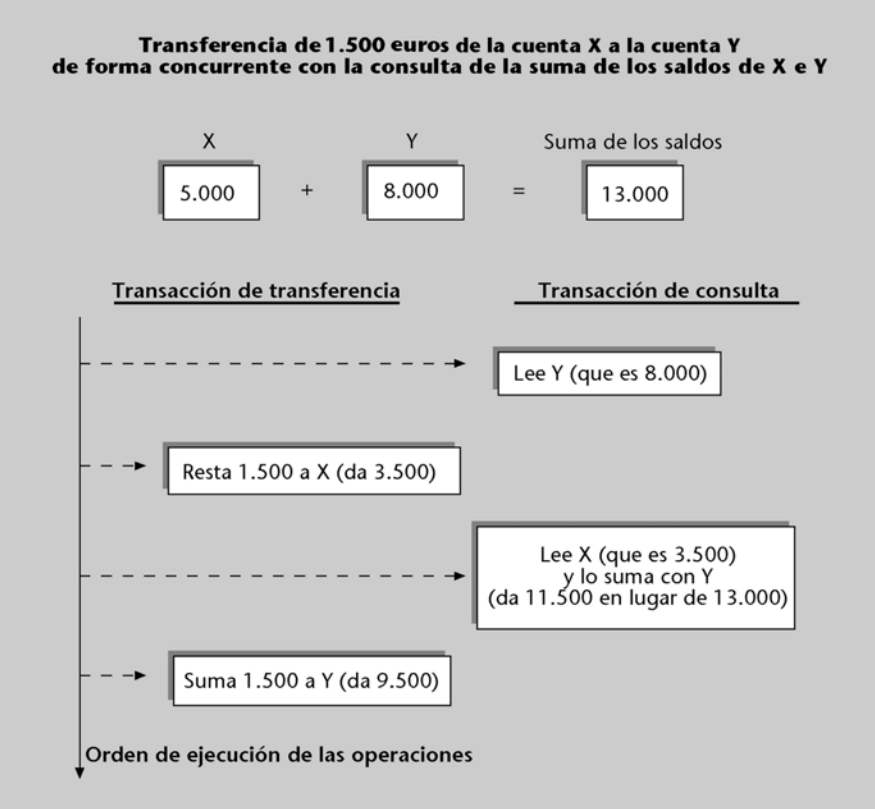
1) Imaginemos que una transacción que transfiere dinero de X a Y se ejecuta concurrentemente con una transacción que observa el saldo de las cuentas Y y X, en este orden, y nos muestra su suma. Si la ejecución de forma concurrente de las dos transacciones casualmente

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 BD, ésta quedará inconsistente (consultad la figura 3).

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 BD y en la corrección del resultado de las consultas.

Figura 3



Nos interesará 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.

Bloqueos en la transferencia de dinero

Por ejemplo, si la transacción de transferencia de dinero modifica el saldo de la cuenta X, el SGBD bloquea esta cuenta de forma que, cuando la transacción de suma quiera leerla, tenga que esperar a que la transacción de transferencia acabe; esto se debe al hecho de que al acabar una transacción, cuando se efectúan las operaciones `COMMIT` o `ROLLBACK` se liberan los objetos que tenía bloqueados.

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.

Recordad que esta asignatura es sólo introductoria y que más adelante, en otras asignaturas, estudiaréis con más detalle los temas que, como el de la concurrencia, aquí sólo introducimos.

3.6. 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 SI militares, las agencias de información y en ámbitos similares, pero durante los años noventa han ido adquiriendo importancia en cualquier SI donde se almacenen datos sobre personas. Recordad que en el Estado español tenemos una ley*, que exige la protección de la confidencialidad de estos datos.

*** Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos de Carácter Personal. (BOE núm. 298, de 12/12/1999, págs. 43088-43099).**

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

Derechos de acceso

Por ejemplo, el usuario SECRE3 podría tener autorización para consultar y modificar todas las entidades de la BD, excepto el valor del atributo *nota de los alumnos*, y no estar autorizado a hacer ningún tipo de supresión o inserción.

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, identificación por reconocimiento de la voz, etc.

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 SI con unas exigencias altísimas, como por ejemplo los militares.

3.7. Otros objetivos

Para recordar las tendencias actuales sobre SGBD, podéis revisar el subapartado 2.4. de esta unidad didáctica.

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 las nuevas 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.

4. Arquitectura de los SGBD

4.1. Esquemas y niveles

Para trabajar con nuestras BD, 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 BD. Esta descripción recibe el nombre de **esquema de la BD**, y los SGBD la tendrán continuamente a su alcance.

El esquema de la BD es un elemento fundamental de la arquitectura de un SGBD que permite independizar el SGBD de la BD; de este modo, se puede cambiar el diseño de la BD (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.

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 BD. El comité conocido como ANSI/SPARC recomendó que la arquitectura de los SGBD previese tres niveles de descripción de la BD, no sólo dos*.

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

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:

Nota

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

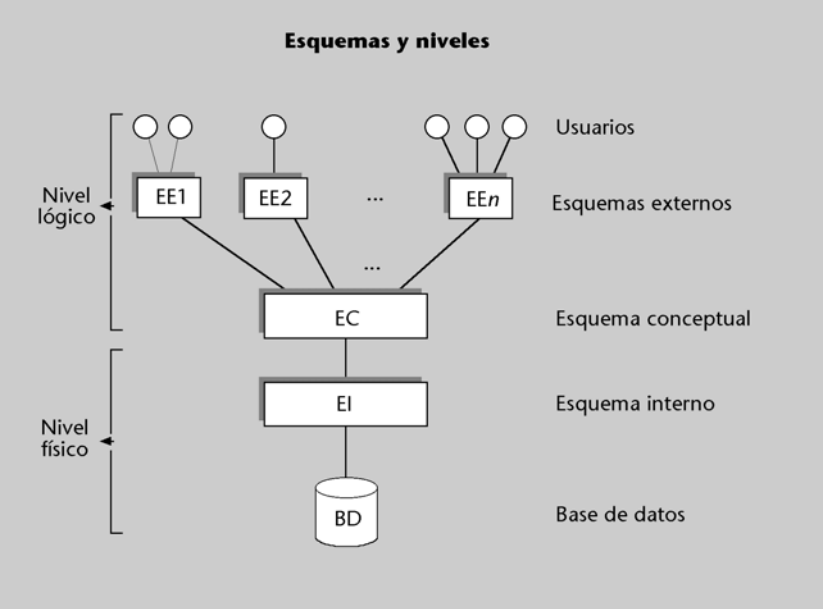
En el caso de las BD, el esquema interno corresponde a la parte física, y el externo a la lógica; en el caso de los ficheros, sucede lo contrario.

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 BD 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**.

Figura 4



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 SI, por lo que se escribirá de forma centralizada durante el denominado *diseño lógico* de la BD.

Sin embargo, cada aplicación podrá tener su visión particular, y seguramente parcial, del esquema conceptual. Los usuarios (programas o usuarios directos) verán la BD 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 BD 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 BD 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 *direccion* y el atributo *edad* (que deriva dinámicamente de *fechanac*).

El **esquema interno** o físico contendrá la descripción de la organización física de la BD: 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 BD y al ajustarlo* posteriormente a las necesidades cambiantes.

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

De acuerdo con la arquitectura ANSI/SPARC, para crear una BD 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.

¿Qué hay que hacer para crear una BD?

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 conceptua*l (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*, *cluster*, espacios para excesos, etc.).

4.2. Independencia de los datos

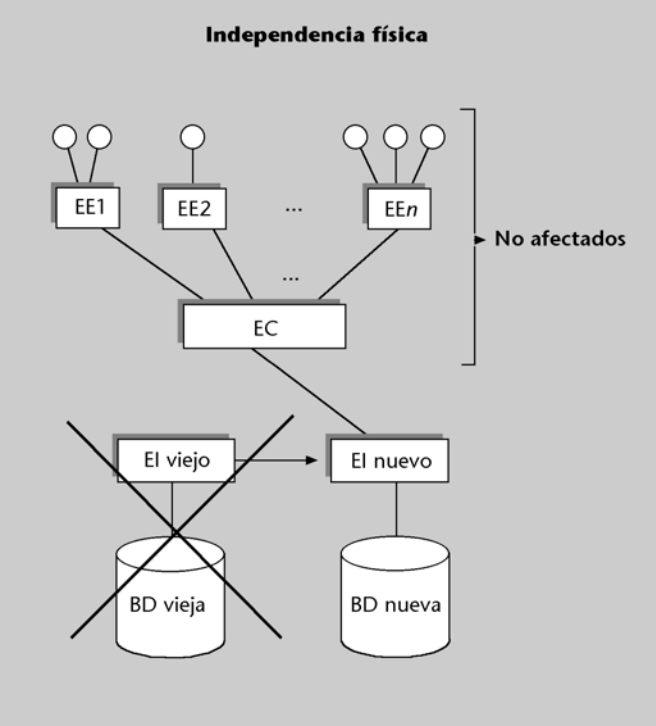
En este subapartado veremos cómo la arquitectura de tres niveles que acabamos de presentar nos proporciona los dos tipos de independencia de los datos: la física y la lógica.

Los dos tipos de independencia de los datos se han explicado en el subapartado 3.2 de esta unidad didáctica.

Hay **independencia física** cuando los cambios en la organización física de la BD 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.

Figura 5



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 determinados*, el formato o la codificación, etc. Ninguno de estos casos debería afectar al mundo exterior, sino sólo a la BD física, el esquema interno, etc.

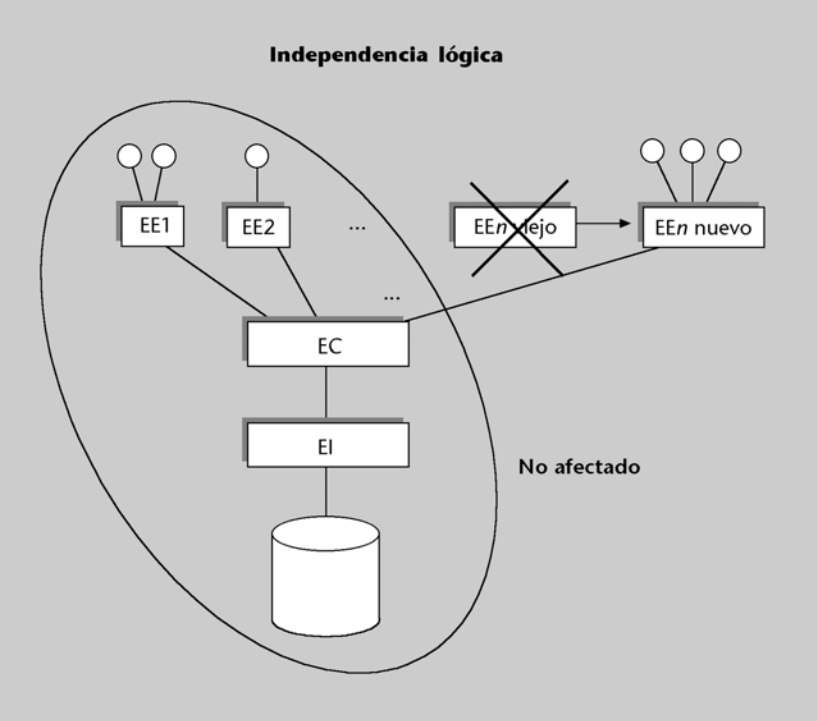
*** Por ejemplo, eliminando un índice en árbol-B o sustituyéndolo por un *hashing*.**

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 BD real (la física).

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

*** Programas de aplicación o usuarios directos.**

Figura 6



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.

Si eliminamos...

... el atributo *apellido*, por ejemplo, no se verán afectados los esquemas externos (ni los usuarios) que no hagan referencia a este atributo. Si se alarga el atributo *dirección* al esquema conceptual, no será necesario modificar el esquema externo donde se ha definido la *dirección* y obviamente, tampoco resultarán afectados los programas y los usuarios que la utilicen.

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 BD física.

Usuarios no afectados por los cambios

Notad 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 definíamos *nombre*, porque allí estaba definido como concatenación de *nombre* y *apellido*. 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 SI piden grados muy elevados de flexibilidad. Los sistemas de ficheros tradicionales, en cambio, no ofrecen ninguna independencia lógica.

4.3. Flujo de datos y de control

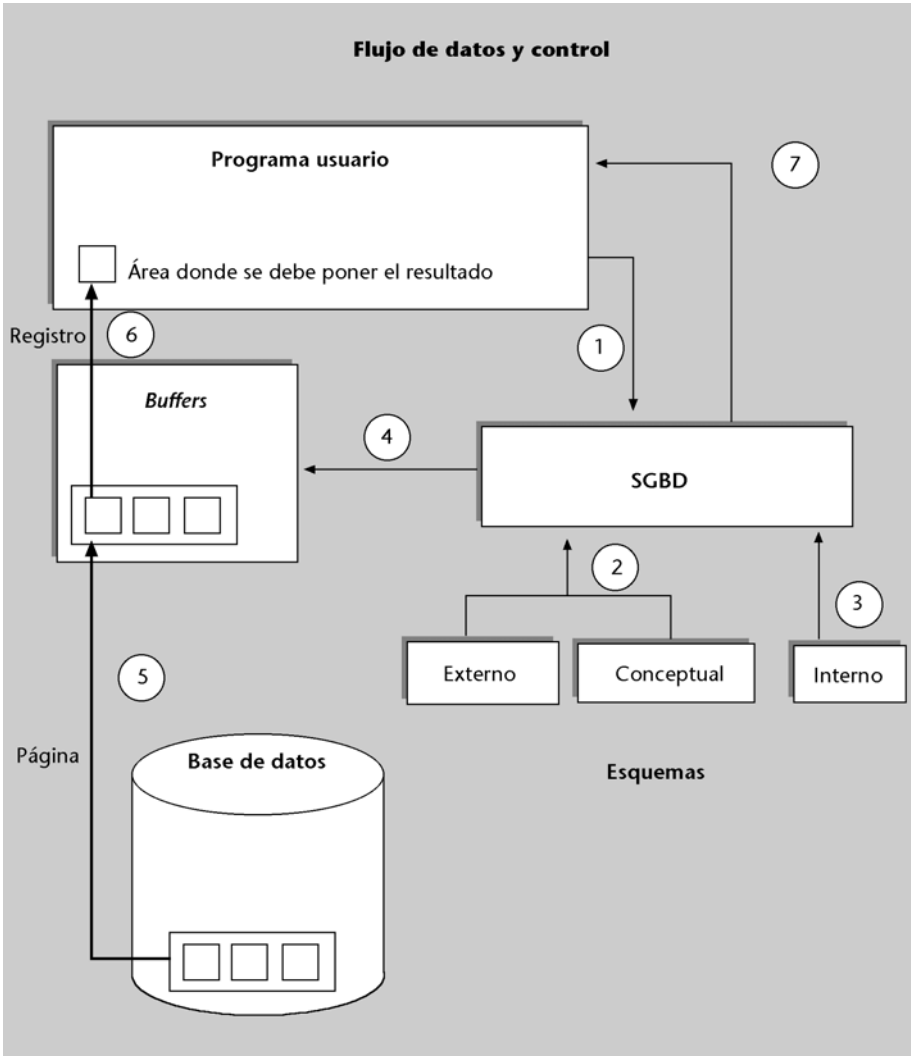
Para entender el funcionamiento de un SGBD, a continuación veremos los principales pasos de la ejecución de una consulta sometida al SGBD por un programa de aplicación. Explicaremos las líneas generales del flujo de datos y de control entre el SGBD, los programas de usuario y la BD.

Recordad que el SGBD, con la ayuda del SO, lee páginas (bloques) de los soportes donde está almacenada la BD 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 propia*.

* Por ejemplo, una variable con estructura de tupla.

Figura 7



Ejecución de una consulta

En la figura vemos representada la BD, los tres niveles de esquemas, el área de los *buffers*, el SGBD y el programa de aplicación que le hace la consulta.

El proceso que se sigue es el siguiente:

a) 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 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 consulta*. 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.

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

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).

Diferencias entre SGBD

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

f) A continuación, el SGBD retorna el control al programa (7) y da por terminada la ejecución de la consulta.

5. Modelos de BD

Una BD es una representación de la realidad (de la parte de la realidad que nos interesa en nuestro SI). Dicho de otro modo, una BD 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.

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

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

¡Cuidado con las confusiones!

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

Todo modelo de BD nos proporciona tres tipos de herramientas: 

a) **Estructuras de datos** con las que se puede construir la BD: tablas, árboles, etc.

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 BD

De los cuatro modelos de BD 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.

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 SGBD*. Sin embargo, encontró la oposición de IBM, la empresa entonces dominante. La propuesta de CODASYL-DBTG ya definía tres niveles de esquemas.

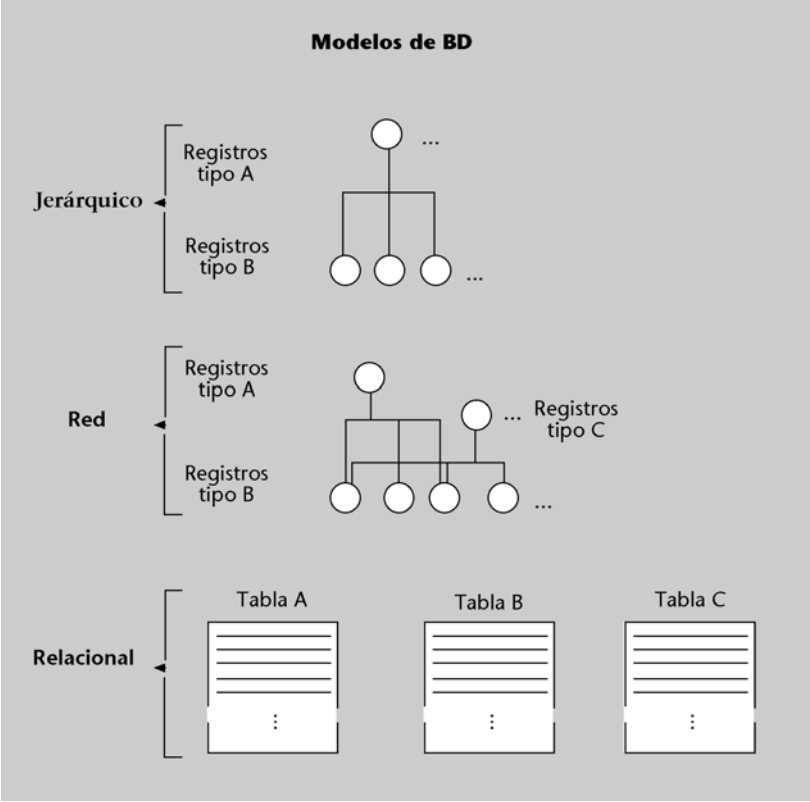
*** Por ejemplo, IDS de Bull, DMS de Univac y DBMS de Digital.**

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 SQL**. 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 SI que actualmente están en funcionamiento utilizan SGBD relacionales, pero algunos siguen utilizando los jerárquicos o en red (especialmente en SI antiguos muy grandes).

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

El modelo relacional se estudia con detalle en la unidad didáctica "El modelo relacional y el álgebra relacional" de este curso.

Figura 8



Así como en los modelos prerrelacionales (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 prerrelacionales 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 BD relacional con objetos**. Se trata de ampliar el modelo relacional, añadiéndole la posibilidad de que los tipos de 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 BD, 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 BD 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 SI 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 BD con un SGBD.

En esta asignatura hablamos sólo de BD con modelos de datos estructurados, que son los que normalmente se utilizan en los SI 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 BD geográficas.

6. 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 BD*. 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 BD 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.

*** Por ejemplo, personas dedicadas a introducir datos masivamente.**

Hay lenguajes especializados en la escritura de esquemas; es decir, en la descripción de la BD. 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 BD (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 BD relacionales, tiene verbos –instrucciones– de tres tipos diferentes:

El lenguaje SQL se explicará en la unidad didáctica "El lenguaje SQL" de este curso.

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.

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 BD).

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 prerrelacionales 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:

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

*** Empezaron a aparecer al final de los años ochenta.**

2) Herramientas o interfaces visuales** muy fáciles de utilizar, que permiten usar las BD 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 BD, así como definirla y actualizar su definición con mucha facilidad y claridad.

**** Han proliferado en los años noventa.**

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 BD, seguramente querremos utilizar nuestro lenguaje habitual de programación*****. Sin embargo, generalmente estos lenguajes no tienen instrucciones para realizar el acceso a las BD. Entonces tenemos las dos opciones siguientes:

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

1) Las **llamadas a funciones**: en el mercado hay librerías de funciones especializadas en BD (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 BD en nuestro programa. Sin embargo, esto exige utilizar un precompilador especializado que acepte en nuestro lenguaje de programación habitual las instrucciones del lenguaje de BD. 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*).

7. Administración de BD

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

Los **administradores de BD** son los responsables del correcto funcionamiento de la BD y velan para que siempre se mantenga útil. Intervienen en situaciones problemáticas o de emergencia, pero su responsabilidad fundamental es velar para 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.

2) Asegurar la máxima disponibilidad de los datos; por ejemplo, haciendo copias (*back-ups*), administrando diarios (*journals* o *logs*), reconstruyendo la BD, 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 BD.

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 BD 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 BD** son las estructuras, las restricciones y las operaciones. Los diferentes modelos de BD 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 BD**.

Actividades

1. Comparad la lista de los objetivos de los SGBD que hemos dado aquí con la que se da en otros libros, y haced otra lista con las diferencias que encontréis.

2. Buscad información técnica y comercial de los fabricantes de SGBD sobre sus productos y a partir de aquí intentad reconocer los conceptos que hemos introducido en esta unidad.

3. Leed algún informe "Estado del arte" sobre SGBD de los que se publican (normalmente cada año) en la revista *Byte* y en *Datamation*.

Lectura recomendada

Consultad las obras de la bibliografía que encontraréis al final de esta unidad.

Ejercicios de autoevaluación

1. ¿Qué ventajas aportaron los SGBD relacionales con respecto a los prerrelacionales?

2. Para mejorar la disponibilidad y el coste, hemos decidido que una cierta parte de una BD que está situada en el ordenador central de la empresa estará duplicada (replicada) en un ordenador situado en una oficina alejada (conectado permanentemente por vía telefónica). Los programas que actualizan la BD, ¿tendrían que preocuparse de actualizar también la réplica? ¿Por qué?

3. Hemos programado una transacción para consultar cuántos alumnos cursan una asignatura. Si este número es inferior a quince, se nos informará de cuántos hay y en una lista, en una hoja de papel o en la pantalla nos aparecerán todos ellos. Sin embargo, si es superior o igual a quince, simplemente dirá cuántos hay. Supongamos que de forma concurrente con esta transacción se podrán estar ejecutando otras que inserten nuevos alumnos o que los supriman. ¿Qué problema se podrá producir si el SGBD no aísla totalmente las transacciones?

4. De las siguientes afirmaciones, decid cuáles son ciertas y cuáles son falsas:
a) El modelo ER es más conocido como *modelo relacional*.
b) Los SGBD no permiten la redundancia.
c) El DML es un lenguaje declarativo.
d) El DDL es un lenguaje pensado para escribir programas de consulta y actualización de BD.
e) En un ordenador que actúa como servidor de BD, con dos RAID y tres discos duros y con un SGBD actual, no es necesario que los encargados de realizar los programas para consultar esta BD sepan en qué discos está.
f) Cuando un programa quiere acceder a unos datos mediante un índice, lo debe decir al SGBD.

Solucionario

Ejercicios de autoevaluación

1. Los SGBD relacionales aportaron una programación más sencilla: los lenguajes son más sencillos y no dependen tanto de las características físicas de la BD. Se da más flexibilidad a los cambios (más independencia física de los datos). El programador se debe preocupar mucho menos de las cuestiones de rendimiento, pues de ello ya se ocupa el SGBD. Incluyen lenguajes declarativos de consulta para usuarios no informáticos.

2. Si la actualización no se hace en los dos lugares, la redundancia nos puede comportar problemas de consistencia de los datos. El administrador de la BD debería poder describir qué quiere que esté replicado y cómo quiere que se haga el mantenimiento de la réplica. El SGBD debería encargarse de mantener la réplica actualizada correctamente. Si la actualización de la réplica la tuviesen que hacer los programas de aplicación, podría suceder que alguno de ellos no lo hiciese, o incluso que la actualización la hiciese (mal) un usuario directo, sin escribir un programa.
Y todavía más, los programas de aplicación y los usuarios directos deberían ser totalmente ajenos a estos temas físicos (rendimiento, disponibilidad, etc.), ya que de este modo se podrían cambiar las decisiones, como por ejemplo variar la política de réplicas, sin que se tuviesen que modificar los programas ni avisar a nadie. Simplemente debe intervenir el ABD mediante el lenguaje de descripción del nivel físico. Actualmente, los SGBD del mercado ya dan este nivel de independencia.

3. Supongamos que la transacción consulta al SGBD cuántos alumnos hay, y el SGBD cuenta trece. El programa preparará una línea de cabecera de lista que indique que hay trece alumnos, y a continuación los mostrará. Sin embargo, entre el momento en que fabrica esta cabecera y el momento de empezar a leer a los alumnos uno por uno para mostrarlos, otras transacciones (que se ejecutan concurrentemente con ésta) eliminan a dos alumnos. Entonces, el programa nos mostrará sólo a once, a pesar de que había anunciado trece.
Y si las transacciones concurrentes que hacen actualizaciones hubiesen insertado tres alumnos, la cabecera diría que hay trece, y mostraría en realidad dieciséis (¡y no tendría que mostrar nunca más de quince!).
Estos problemas de concurrencia pueden surgir si el SGBD no lleva un control que evite las interferencias.

4. a) Falsa, **b)** Falsa, **c)** Falsa, **d)** Falsa, **e)** Cierta, **f)** Falsa.

Glosario

administrador de BD
Tipo de usuario especial que realiza funciones de administración y control de la BD, que aseguran que la explotación de la BD es correcta.

base de datos
Conjunto estructurado de datos que representa entidades y sus interrelaciones. La representación será única e integrada, a pesar de que debe permitir diversas utilizaciones.
sigla: *BD*

BD
Ved: *base de datos*.

cliente/servidor
Tecnología habitual para distribuir datos. La idea es que dos procesos diferentes, que se pueden ejecutar en un mismo sistema o en sistemas separados, actúan de modo que uno actúa de cliente o peticionario de un servicio y el otro, como servidor. Un proceso cliente puede pedir servicios a distintos servidores. Un servidor puede recibir peticiones de muchos clientes. En general, un proceso *A* que actúa como cliente pidiendo un servicio a otro proceso *B* puede hacer también de servidor de un servicio que le pida otro proceso *C*.
sigla: *C/S*

C/S
Ved: *cliente/servidor*.

data definition language
Lenguaje especializado en la escritura de esquemas; es decir, en la descripción de BD.
Sigla: *DDL*

data manipulation language
Lenguaje especializado en la utilización de BD (consultas y mantenimiento).
Sigla: *DML*

DDL
Ved: *data definition language*.

DML
Ved: *data manipulation language*.

esquema
Descripción o definición de la BD. Esta descripción está separada de los programas y es utilizada por el SGBD para saber cómo es la BD con la que debe trabajar. La arquitectura ANSI/SPARC recomienda tres niveles de esquemas: el externo (visión de los usuarios), el conceptual (visión global) y el físico (descripción de características físicas).

SGBD
Ved: *sistema de gestión de BD*.

sistema de gestión de BD
Software que gestiona y controla BD. Sus principales funciones son facilitar la utilización de la BD a muchos usuarios simultáneos y de tipos diferentes, independizar al usuario del mundo físico y mantener la integridad de los datos.
sigla: SGBD

SQL
Ved: *structured query language*.

structured query language
Lenguaje especializado en la descripción (DDL) y la utilización (DML) de BD relacionales. Creado por IBM al final de los años setenta y estandarizado por ANSI-ISO en 1985 (el último estándar de SQL es de 1999). En la actualidad lo utilizan prácticamente todos los SGBD del mercado.
sigla: SQL

transacción
Conjunto de operaciones (de BD) que queremos que se ejecuten como un todo (todas o ninguna) y de forma aislada (sin interferencias) de otros conjuntos de operaciones que se ejecuten concurrentemente.

Bibliografía

Bibliografía básica

Date, C.J. (2001). *Introducción a los sistemas de bases de* datos (7ª ed.). Prentice Hall.

Silberschatz, A; Korth, H.F; Sudarshan, S. (1998). *Fundamentos de bases de datos* (3.ª ed.). Madrid: McGraw-Hill.
La introducción de este libro es similar a la que hemos hecho aquí.

Bibliografía complementaria

Para actualizar vuestra visión global de los SGBD del mercado, podéis consultar en Internet las páginas de los fabricantes de SGBD; por ejemplo:

- *Informix*: http://www.informix.com
- *Oracle*: http://www.oracle.com
- *IBM*: http://www.ibm.com
- *Computer Associates*: http://www.ca.com
- *Microsoft*: http://www.microsoft.com
- *NCR*: http://www.ncr.com

El modelo relacional y el álgebra relacional

Dolors Costal Costa

UOC
Formación de Posgrado

Índice

Introducción

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

Consultad el concepto de *modelo de datos* en la unidad didáctica "Introducción a las bases de datos" de este curso.

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.

Las construcciones del SQL se estudian en la unidad didáctica "El lenguaje SQL".

En esta unidad se analizan también 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 este curso. 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 del presente curso, pero es relevante para estudios más avanzados sobre bases de datos.

Objetivos

En los materiales didácticos de esta unidad encontraréis 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.

4. Conocer las operaciones del álgebra relacional.

5. Saber utilizar las operaciones del álgebra relacional para consultar una base de datos.

1. 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:

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

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

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

El concepto de *modelo de datos* se ha explicado en la unidad didáctica "Introducción a las bases de datos"de este curso.

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.

El concepto de *SGBD* ha sido presentado en la unidad didáctica "Introducción a las bases de datos" de este curso.

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. 

2. Estructura de los datos

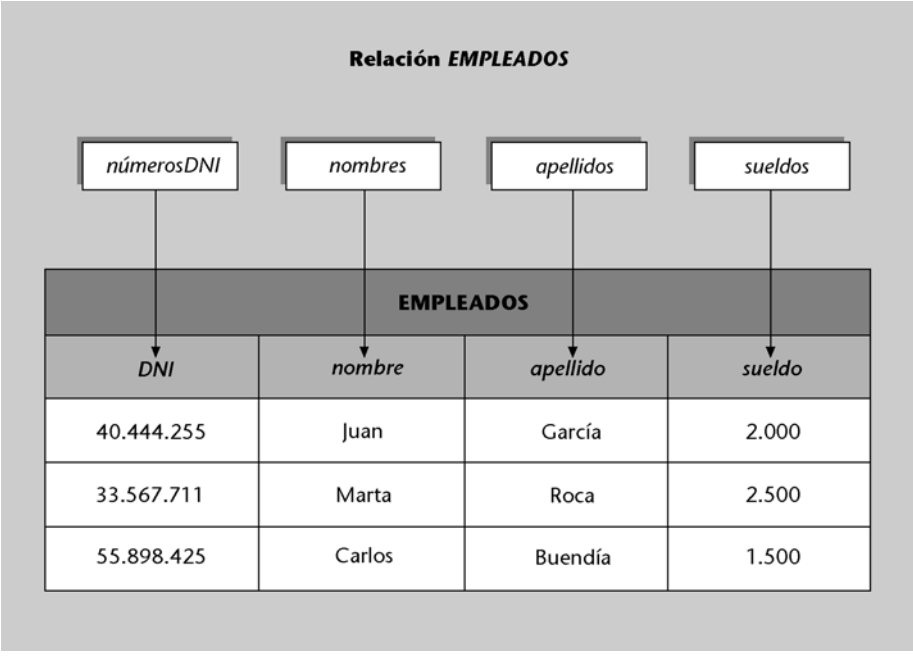
El modelo relacional proporciona una estructura de los datos que consiste en un conjunto de relaciones con objeto de representar la información que nos interesa del mundo real.

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

2.1. 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*.

Figura1



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.

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.

2.2. 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 denomine *dom_edad* y que contenga los valores enteros que están entre 16 y 65.

Un **relación** se compone del **esquema** (o intensión 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:

Figura 2

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 $\{A_1, A_2, ..., A_n\}$.

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, ..., A_n), donde R es el nombre la relación y A_1, A_2, ..., A_n es una ordenación cualquiera de los atributos que pertenecen al conjunto {A_1, A_2, ..., A_n}.

Denotación del esquema 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, dominio(*DNI*) = *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: dominio(*telcasa*)= = *teléfono* y dominio(*teltrabajo*) = *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, ..., A_n)** es un conjunto de tuplas t_i (i = 1, 2, ..., m), donde cada tupla t_i es, a su vez un conjunto de pares t_i = {<A_1:v_{i1}>, <A_2:v_{i2}> ... >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*.

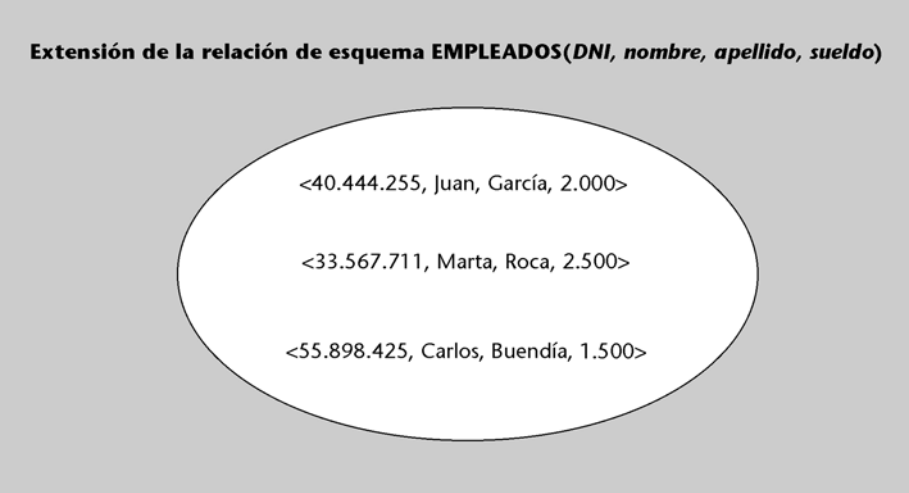
Algunos autores...

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

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

Si denotamos el esquema de la relación representada en la figura 1 como EMPLEADOS(*DNI, nombre, apellido, sueldo*), el conjunto de tuplas de su extensión será el de la figura siguiente:

Figura 3



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 $t_i = <v_{i1}, v_{i2}, ..., v_{in}>$, 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 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.

2.3. 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.

El concepto de *extensión de una relación* se ha explicado en el subapartado 2.2. de esta unidad didáctica.

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 $\{A_1, A_2, ..., A_n\}$. Así pues, no hay un orden entre los atributos de un esquema de relación, teniendo en cuenta que estos atributos forman un conjunto.

El concepto de *esquema de una relación* se ha explicado en el subapartado 2.2. de esta unidad didáctica.

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*).

2.4. 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*.

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.

Una **superclave de una relación de esquema** $R(A_1, A_2, ..., A_n)$ 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.

Observad 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, NSS, nombre, apellido, teléfono*) esta superclave sería: {*DNI, NSS, nombre, apellido, teléfono*}.

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

Algunas superclaves de la relación *EMPLEADOS*

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

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

Notad que, ...

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

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 ningún atributo para hacer la identificación.

Claves candidatas de *EMPLEADOS*

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

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.

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.

Las claves candidatas no elegidas como 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($\underline{A_1, A_2, ..., A_i}$, ..., A_n) indica que los atributos A_1, A_2, ..., A_i forman la clave primaria de *R*.

Elección de la clave primaria de *EMPLEADOS*

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

2.5. 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.

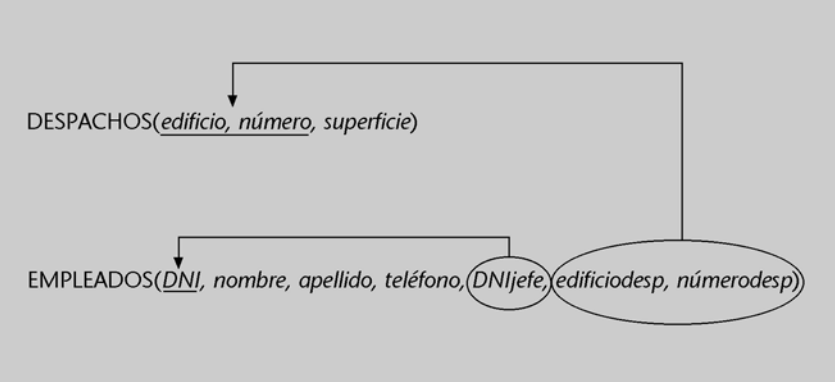
En ocasiones, incluso puede ser necesario reflejar lazos entre tuplas que pertenecen a una misma relación. Por ejemplo, en la misma base de datos anterior puede

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		
edificio	***número***	***superficie***
Marina	120	10
Marina	122	15
Marina	230	20
Diagonal	120	10

- Relación *EMPLEADOS*

EMPLEADOS					
DNI	***nombre***	***apellido***	***DNIjefe***	***edificiodesp***	***númerodesp***
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 dominio(*edificiodesp*) = dominio(*edificio*) y también que dominio(*númerodesp*) = dominio(*número*).

Lectura recomendada

Encontraréis explicaciones detalladas sobre la coincidencia de dominios en la obra siguiente:

C.J. Date (2001). *Introducción a los sistemas de bases de datos* (7ª ed., cap. 19). Prentice Hall.

Observad 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*.

2.6. 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.

El lenguaje SQL se explica en la unidad didáctica "El lenguaje SQL" de este curso.

3. 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 sintaxis y el funcionamiento de las operaciones de actualización y consulta, en el caso concreto del lenguaje relacional SQL, se estudian con detalle en la unidad "El lenguaje SQL" de este curso.

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 relacionales 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.

El álgebra relacional se explica en el apartado 5 de esta unidad didáctica.

4. 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 relacionales, 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. Observad 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.

La forma de definir estas restricciones con el lenguaje SQL se explica en la unidad "El lenguaje SQL" de este curso.

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 base 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.

4.1. 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.

Es preciso destacar que el mismo concepto de *clave primaria* implica esta condición. El concepto de *clave primaria* se ha explicado en el subapartado 2.4. de esta unidad didáctica.

Ejemplo

Tenemos la siguiente relación:

DESPACHOS		
edificio	***número***	***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		
edificio	***número***	***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>.

4.2. 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		
edificio	***número***	***superficie***
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 <NULO, 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, NULO, 15>. Tampoco debería ser posible modificar la tupla <Marina, 120, 10> de modo que pasara a ser <NULO, 120, 10>.

4.3. 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.

Observad que todo lo que impone la regla de integridad referencial viene implicado por la misma noción de *clave foránea* que se ha explicado en el subapartado 2.5 de esta unidad.

Ejemplo

Si tenemos las siguientes relaciones:

- Relación *DESPACHOS*:

DESPACHOS		
edificio	***número***	***superficie***
Marina	120	10
Marina	122	15
Marina	230	20
Diagonal	120	10

- Relación *EMPLEADOS*:

EMPLEADOS				
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	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		
edificio	***número***	***superficie***
Marina	120	10
Marina	122	15
Marina	230	20
Diagonal	120	10

- Relación *EMPLEADOS*:

EMPLEADOS				
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	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.

4.3.1. 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	
numcliente	...
10	–
15	–
18	–

- Relación *PEDIDOS_PENDIENTES*

PEDIDOS_PENDIENTES		
numped	...	***numcliente****
1.234	–	10
1.235	–	10
1.236	–	15

* {*numcliente*} referencia *CLIENTES*.

a) 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.

b) 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.

4.3.2. 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	
nombreedificio	...
Marina	–
Diagonal	–

- Relación *DESPACHOS*:

DESPACHOS		
edificio*	***número***	***superficie***
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	
nombreedificio	...
Marina	–

- Relación *DESPACHOS*:

DESPACHOS		
edificio*	***número***	***superficie***
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	
nombreedificio	...
Mar	–

- Relación *DESPACHOS*:

DESPACHOS		
edificio*	***número***	***superficie***
Mar	120	10
Mar	122	15
Mar	230	20

* {*edificio*} referencia *EDIFICIOS*.

4.3.3. 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	
numvendedor	...
1	–
2	–
3	–

- Relación *CLIENTES*:

CLIENTES		
numcliente	...	***vendedorasig****
23	–	1
35	–	1
38	–	2
42	–	2
50	–	3

* {*vendedorasig*} referencia *VENDEDORES*.

a) 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	
numvendedor	...
2	–
3	–

- Relación *CLIENTES*:

CLIENTES		
numcliente	...	***vendedorasig****
23	–	NULO
35	–	NULO
38	–	2
42	–	2
50	–	3

* {*vendedorasig*} referencia *VENDEDORES*.

b) 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	
numvendedor	...
5	–
3	–

- Relación *CLIENTES*:

CLIENTES		
numcliente	...	***vendedorasig****
23	–	NULO
35	–	NULO
38	–	NULO
42	–	NULO
50	–	3

* {*vendedorasig*} referencia *VENDEDORES*.

4.3.4. 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.

La forma de definir estas políticas de mantenimiento de la integridad con el lenguaje SQL se explica en la unidad "El lenguaje SQL" de este curso.

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.

4.4. 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 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.

Recordad que los conceptos de *dominio predefinido* y *dominio definido por el usuario* se han explicado en el subapartado 2.2 de esta unidad didáctica.

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.

Lectura complementaria

Para estudiar con más detalle la segunda condición de la regla de integridad de dominio, podéis consultar la siguiente obra:

C.J. Date (2001). *Introducción a los sistemas de bases de datos* (7ª ed., cap. 19). Prentice Hall.

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. 

5. El 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.

Consultad el apartado 3 de esta unidad didáctica.

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**.

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.

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.

Las operaciones del álgebra relacional clasificadas según sean conjuntistas o específicamente relacionales se estudian en los subapartados 5.1 y 5.2 de esta unidad.

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* $\cup \cup$ *EMPLEADOS_PROD* 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* $\cup$ *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 *redenominar* con este objetivo.

La **operación *redenominar***, que denotaremos con el símbolo :=, permite asignar 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* $\cup$ *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 *redenominar*.

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

$$R(A_1, A_2, ..., A_n) := S(B_1, B_2, ..., B_n).$$

A continuación presentaremos un ejemplo que utilizaremos para ilustrar las operaciones del álgebra relacional. Después 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_ADM*:

EMPLEADOS_ADM				
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	***nombreemp***	***apellidoemp***	***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_ADM* indican que el empleado correspondiente no tiene despacho.

5.1. Operaciones conjuntistas

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

5.1.1. 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 habéis podido 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 dominio(A_i) = 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

Notad 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:

$$R := EMPLEADOS\_ADM \cup EMPLEADOS\_PROD.$$

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

R				
DNI	***nombre***	***apellido***	***edificiodesp***	***númerodesp***
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.

5.1.2. 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 *EMPLEADOS_ADM* $\cap$ *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:

$$R := EMPLEADOS\_ADM \cap EMPLEADOS\_PROD.$$

Entonces, la relación *R* resultante será:

R				
DNI	***nombre***	***apellido***	***edificiodesp***	***númerodesp***
33.567.711	Marta	Roca	Marina	120

Observad 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.

5.1.3. 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.

5.1.4. 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*.

*** Recordad que *T* y *S* no tienen ningún nombre de atributo común.**

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

EDIFICIOS(*nombreedificio*, *supmediadesp*) := EDICIOS_EMP(*edificio*, *supmediadesp*).

$R := EDIFICIOS \times DESPACHOS$.

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

R				
nombreedificio	*supmediadesp*	*edificio*	*número*	*superficie*
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.

5.2. Operaciones específicamente relacionales

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

5.2.1. 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 como T(*C*).

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* = Marina y *superficie* > 12; se indicaría DESPACHOS(*edificio* = Marina y *superficie* > 12).

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

$$A_i \, \theta \, v,$$

o bien:

$$A_i \, \theta \, A_j,$$

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

* Es decir, =, ≠, <, ≤, >, o ≥.

- En las cláusulas de la forma A_i θ v, v es un valor del dominio de A_i.
- En las cláusulas de la forma A_i, θ 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" (∧) y "o" (∨).

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

Los **atributos del esquema de la relación resultante de T(*C*)** coinciden con los atributos del esquema de la relación *T*.

La **extensión de la relación resultante de T(*C*)** 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:

$$R := DESPACHOS(edificio = \text{Marina } y\ superficie > 12).$$

La relación *R* resultante será:

R		
edificio	***número***	***superficie***
Marina	230	20

5.2.2. 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.

La proyección es una operación unaria. Siendo $\{A_i, A_j, ..., A_k\}$ un subconjunto de los atributos del esquema de la relación T, la proyección de T sobre $\{A_i, A_j, ..., A_k\}$ se indica como $T[A_i, A_j, ..., A_k]$.

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: *EMPLEADOS_ADM* [*nombre, apellido*].

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

Los **atributos del esquema de la relación resultante de $T[A_i, A_j, ..., A_k]$** son los atributos $\{A_i, A_j, ..., A_k\}$.

La **extensión de la relación resultante de $T[A_i, A_j, ..., A_k]$** es el conjunto de todas las tuplas de la forma $<t.A_i, t.A_j, ..., t.A_k>$, 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

Notad 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 := EMPLEADOS_ADM[*nombre, apellido*].

Entonces, la relación R resultante será:

R	
nombre	***apellido***
Juan	García
Marta	Roca

5.2.3. Combinación

La combinación es una operación que, a partir de dos relaciones, obtiene una nueva relación formada por todas las tuplas que resultan de concadenar 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 $T[B]S$.

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[*edificiodesp = edificio, 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*[*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 (=, ≠, <, ≤, >, ≥), y se cumple que A_i y A_j tienen el mismo dominio. Las comparaciones de una condición de combinación se separan mediante comas.

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*[*B*]*S*** son todos los atributos de *T* y todos los atributos de *S**.

*** Recordad que *T* y *S* no tienen ningún nombre de atributo común.**

La e**xtensión de la relación resultante de *T*[*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 (observad que es preciso redenominar previamente los atributos):

EDIFICIOS(*nombreeedficio,supmediadesp*) := EDIFICIOS_EMP(e*dificio, supmediadesp*),

R := *EDIFICIOS*[*nombreedificio* = *edificio, supmediadesp* ≤ *superficie*] DESPACHOS.

Entonces, la relación *R* resultante será:

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 := *EMPLEADOS_ADM[edificiodesp = edificio, númerodesp = número]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	120	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.

Observad 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 $T * S$ 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*.

Observad 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 := *EDIFICIOS_EMP* * *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

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

En ocasiones, antes de la combinación natural es necesario aplicar la operación *redenominar* 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(*edificiodesp, númerodesp, superficie*) := DESPACHOS(*edificio, número, superficie*),

$$R := EMPLEADOS\_ADM * D.$$

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

5.3. 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:

$$R := (EMPLEADOS\_ADM \cup EMPLEADOS\_PROD)\ [nombre, apellido].$$

b) O bien podemos expresarlo en dos pasos:

- *EMPS := EMPLEADOS_ADM ∪ EMPLEADOS_PROD;*
- *R* := EMPS[*nombre, apellido*]

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.

Recordad que la base de datos que se utiliza en los ejemplos se ha descrito en la introducción del apartado 5 de esta unidad didáctica.

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* := EDIFICIOS_EMP(*supmediadesp* > 12);
- *B := DESPACHOS * A*;
- *R := B*[*edificio, número*]

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 ∪ EMPLEADOS_PROD*;
- *B* := A(*edificiodesp* = Marina y *númerodesp* = 120);
- *R* := B[*nombre, apellido*].

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* := DESPACHOS [*edificio, número*];
- *B* := EMPLEADOS_ADM[*edificiodesp, númerodesp*];
- *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[*DNI, nombre, apellido, edificio, número*] := EMPLEADOS_ADM[*DNI, nombre, apellido, edificiodesp, númerodesp*];
- *B := A * DESPACHOS*;
- *R := B[DNI, nombre, apellido, superficie].*

5.4. 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.

Las combinaciones se han explicado en el subapartado 5.3.3 de esta unidad didáctica.

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

D(*edificiodesp, númerodesp, superficie*) := DESPACHOS (*edificio, número, superficie*),

R := *EMPLEADOS_PROD* * *D*.

Puesto que se trata de una combinación natural, se considera que la condición de combinación es *edificio* = *edificio* y *número* = *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

Notad 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 $T[C]_{\mathrm{I}}S$, 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 $T[C]_{\mathrm{D}}S$, 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 $T[C]_{\mathrm{p}}S$, 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, $T * S$, concretamente:

a) La combinación natural externa izquierda entre dos relaciones T y S, que se indica como $T *_I S$, 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 $T *_D S$, 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 $T *_P S$, 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):

D(*edificiodesp, númerodesp, superficie*) := DESPACHOS (*edificio, número, superficie*),

$$R := EMPLADOS\_PROD *_D D,$$

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. Notad que los atributos *DNI*, *nombre* y *apellido* para los despachos que no tienen empleado reciben valores nulos.

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

D(*edificiodesp, númerodesp, superficie*) := DESPACHOS(*edificio, número, superficie*),

$$R := EMPLEADOS\_PROD *_I D,$$

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. Observad que el atributo superficie para los empleados que no tienen despacho contiene un valor nulo.

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

$$D(\textit{edificiodesp, númerodesp, superficie}) := \text{DESPACHOS}(\textit{edificio, número, superficie}),$$
$$R := \textit{EMPLEADOS\_PROD} *_{\text{P}} D,$$

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.

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.

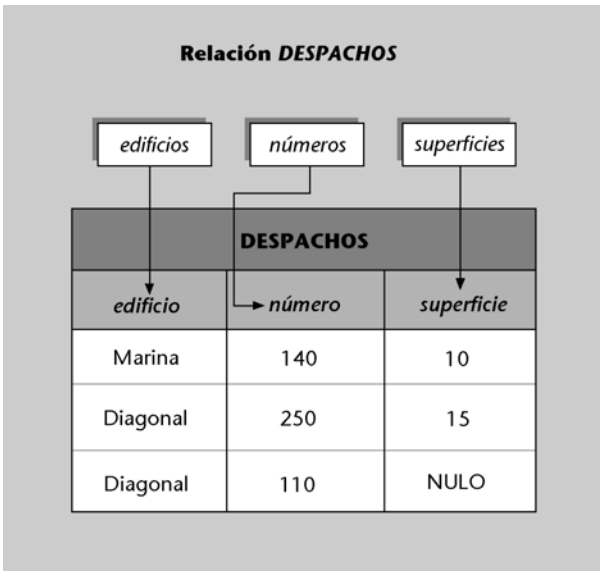
b) Operaciones específicamente relacionales: selección, proyección y combinación.

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

Ejercicios de autoevaluación

1. Dada la relación que corresponde a la siguiente representación tabular:

Figura 4



a) Indicad qué conjunto de atributos tiene.
b) Decid qué dominio tiene cada uno de sus atributos.
c) Escribid todas las distintas formas de denotar su esquema de relación.
d) Elegid una de las formas de denotar su esquema de relación y utilizadla para dibujar el conjunto de tuplas correspondiente a su extensión.

2. Indicad cuáles son todas las superclaves de las siguientes relaciones:
a) *DESPACHOS*(*edificio, número, superficie*), que tiene como única clave candidata la siguiente: *edificio, número*.
b) *EMPLEADOS*(*DNI, NSS, nombre, apellido*), que tiene las siguientes claves candidatas: *DNI y NSS*.

3. Decid, para cada una de las siguientes operaciones de actualización, si se podría aceptar su aplicación sobre la base de datos que se ha utilizado en esta unidad:
a) Insertar en *EDIFICIOS_EMP* la tupla <Nexus, 30>.
b) Insertar en *DESPACHOS* la tupla <Diagonal, NULO, 15>.
c) Insertar en *EMPLEADOS_ADM* la tupla <55.555.555, María, Puig, Diagonal, 500>.
d) Modificar en *DESPACHOS* la tupla <Marina, 230, 20> por <Marina, 120, 20>.
e) Borrar en *EMPLEADOS_PROD* la tupla <88.999.20, Pedro, González, NULO, NULO>.
f) Modificar en *EMPLEADOS_ADM* la tupla <40.444.255, Juan, García, Marina, 120> por <33.567.711, Juan, García, Marina, 120>.
g) Borrar en *EDIFICIOS_EMP* la tupla <Marina, 15> si para la clave foránea edificio de *DESPACHOS* se ha seleccionado la política de restricción en caso de borrado.
h) Borrar en *EDIFICIOS_EMP* la tupla <Marina, 15> si para la clave foránea edificio de *DESPACHOS* se ha seleccionado la política de actualización en cascada en caso de borrado.

4. Escribid secuencias de operaciones del álgebra relacional que resuelvan las siguientes consultas en la base de datos que hemos utilizado en esta unidad:
a) Obtener los despachos con una superficie mayor que 15. Concretamente, se quiere saber el nombre del edificio, el número y la superficie de estos despachos, junto con la superficie media de los despachos del edificio donde están situados.
b) Obtener el nombre del edificio y el número de los despachos que no tienen asignado a ningún empleado (ni de producción ni de administración).
c) Obtener el nombre y el apellido de los empleados (tanto de administración como de producción), que no tienen despacho.
d) Obtener el nombre y el apellido de todos los empleados (tanto de administración como de producción) que tienen despacho asignado, junto con la superficie de su despacho y la superficie media de los despachos del edificio al que pertenece su despacho.
e) Obtener los despachos con una superficie mayor que la superficie del despacho Diagonal, 120. Concretamente, se quiere saber el nombre del edificio y el número de estos despachos.
f) Obtener todos los despachos de la empresa (tanto si tienen empleados como si no), junto con los empleados que tienen asignados (en caso de que los tengan). Concretamente, se quiere conocer el nombre del edificio, el número de despacho y el DNI del empleado.

5. Sea R la relación que resulta de la intersección de las relaciones T y S, es decir, $R := T \cap S$. Escribid una secuencia de operaciones del álgebra relacional que incluya sólo operaciones primitivas y que obtenga como resultado R.

6. Sean las relaciones de esquema T(A, B, C) y S(D, E, F), y sea R la relación que resulta de la siguiente combinación:

$$R := T[B = D,\ C = E]S.$$

Escribid una secuencia de operaciones del álgebra relacional que incluya sólo operaciones primitivas y que obtenga como resultado R.

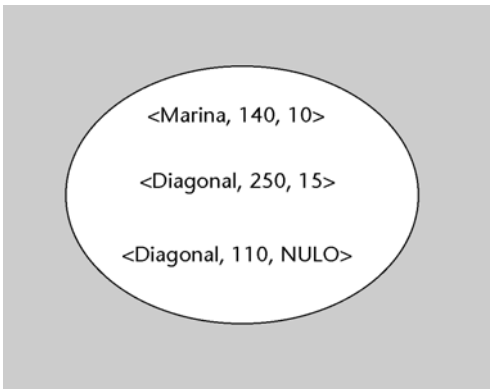
Solucionario

Ejercicios de autoevaluación

1.
a) La relación representada tiene el siguiente conjunto de atributos: *edificio, número, superficie*.
b) Los dominios son dominio(*edificio*) = *edificios*, dominio(*número*) = *números* y dominio(*superficie*) = *sups*.
c) Las formas de denotar el esquema de relación son:
- DESPACHOS(*edificio, número, superficie*),
- DESPACHOS(*edificio, superficie, número*),
- DESPACHOS(*número, edificio, superficie*),
- DESPACHOS(*número, superficie, edificio*),
- DESPACHOS(*superficie, edificio, número*),
- DESPACHOS(*superficie, número, edificio*),

que corresponden a las posibles ordenaciones de sus atributos.
d) Elegiremos la siguiente forma de denotar el esquema de relación:
DESPACHOS(*edificio, número, superficie*).
Entonces el conjunto de tuplas de su extensión será:

Figura 5



2. Las superclaves de las relaciones correspondientes son:
a) {*edificio, número*} y {*edificio, número, superficie*}.
b) {*DNI*}, {*NSS*}, {*DNI, NSS*}, {*DNI, nombre*}, {*DNI, apellido*}, {*NSS, nombre*}, {*NSS, apellido*}, {*DNI, nombre, apellido*}, {*NSS, nombre, apellido*}, {*DNI, NSS, nombre, apellido*} y {*DNI, NSS, nombre, apellido*}.

3.
a) Se acepta.
b) Se rechaza porque viola la regla de integridad de entidad de la clave primaria.
c) Se rechaza porque viola la regla de integridad referencial.
d) Se rechaza porque viola la regla de integridad de unicidad de la clave primaria.
e) Se acepta.
f) Se rechaza porque viola la regla de integridad de unicidad de la clave primaria.
g) Se rechaza porque viola la regla de integridad referencial.
h) Se acepta y se borran el edificio Marina y todos sus despachos.

4.
a) Podemos utilizar la siguiente secuencia de operaciones:
- A := *DESPACHOS*(*superficie*>15),
- $R := A * EDIFICIOS\_EMP$.

b) Podemos utilizar la siguiente secuencia de operaciones:
- A := *DESPACHOS*[*edificio, número*],
- $B := EMPLEADOS\_ADM \cup EMPLEADOS\_PROD$,
- C := B[*edificiodesp, númerodesp*],
- $R := A - C$.

c) Podemos utilizar la siguiente secuencia de operaciones:
- $A := EMPLEADOS\_ADM \cup EMPLEADOS\_PROD$,
- B := A(*edificiodesp* = NULO y *númerodesp* = NULO),
- R := B[*nombre, apellido*].

d) Podemos utilizar la siguiente secuencia de operaciones:
- $A := EMPLEADOS\_ADM \cup EMPLEADOS\_PROD$,
- *B(DNI, nombre, apellido, edificio, número) := A(DNI, nombre, apellido, edificiodesp, númerodesp)*,
- $C := B * DESPACHOS$,
- $D := C * EDIFICIOS\_EMP$,
- R := D[*nombre, apellido, superficie, supmediadesp*].

e) Podemos utilizar la siguiente secuencia de operaciones:
- A := *DESPACHOS*(*edificio = Diagonal y número = 120*),
- B(*Ed*, *Num*, *Sup*) := A(*edificio*, *número*, *superficie*),
- C := *DESPACHOS*[*superficie*>Sup]B,
- R := C[*edificio*, *número*].

f) Podemos utilizar la siguiente secuencia de operaciones:
- A := *EMPLEADOS_ADM* $\cup$ *EMPLEADOS_PROD*,
- B(*DNI*, *nombre*, *apellido*, *edificio*, *número*) := A(*DNI*, *nombre*, *apellido*, *edificiodesp*, *númerodesp*),
- C := *DESPACHOS* $*_1$ B,
- R := C[*edificio*, *número*, *DNI*].

5. La secuencia siguiente:
- $A := T - S$,
- $R := T - A$,

sólo incluye operaciones primitivas, dado que la diferencia es primitiva, y obtiene el mismo resultado que $R := T \cap S$.

6. La siguiente secuencia:
- $A := T \times S$,
- $R := A(B = D \text{ y } C = E)$,

que sólo incluye operaciones primitivas (un producto cartesiano y una selección), obtiene el mismo resultado que $R := T[B = D, C = E]S$.

Glosario

actualización
Hecho de reflejar los cambios que se producen en la realidad en las relaciones de una base de datos.

actualización en cascada para el caso de borrado
Política de mantenimiento de la integridad referencial que consiste en borrar una tupla t que tiene una clave primaria referenciada, así como borrar todas las tuplas que referencian t.

actualización en cascada para el caso de modificación
Política de mantenimiento de la integridad referencial que consiste en permitir modificar atributos de la clave primaria de una tupla t con una clave primaria referenciada, y modificar del mismo modo todas las tuplas que referencian la tupla t.

anulación en caso de borrado
Política de mantenimiento de la integridad referencial que consiste en borrar una tupla t con 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.

anulación en caso de modificación
Política de mantenimiento de la integridad referencial que consiste en modificar atributos de la clave primaria de una tupla t con 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.

atributo (en el contexto del modelo relacional)
Nombre del papel que ejerce un dominio en un esquema de relación.

borrado
Hecho de borrar una o más tuplas de una relación.

cardinalidad de una relación
Número de tuplas que pertenecen a su extensión.

cierre relacional
Propiedad de todas las operaciones del álgebra relacional según la cual tanto sus operandos como su resultado son relaciones.

clave alternativa de una relación
Clave candidata de la relación que no se ha elegido como clave primaria.

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

clave primaria de una relación
Clave candidata de la relación que se ha elegido para identificar las tuplas de la relación.

clave foránea de una relación *R*
Subconjunto de los atributos del esquema de la relación, *CF*, tal que existe una relación *S* (*S* no debe ser necesariamente diferente de *R*) que tiene por clave primaria *CP*, y se cumple que, para toda tupla *t* de la extensión de *R*, los valores para *CF* de *t* son o bien valores nulos, o bien valores que coinciden con los valores para *CP* de alguna tupla *s* de *S*.

combinación
Operación del álgebra relacional 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 relación, y que cumplen una condición de combinación especificada.

combinación externa
Extensión de combinación entre dos relaciones, *T* y *S*, que conserva en el resultado todas las tuplas de *T*, de *S* o de las dos relaciones.

combinación natural
Variante de combinación que consiste básicamente en una equicombinación seguida de la eliminación de los atributos superfluos.

consulta
Obtención de datos deducibles a partir de las relaciones que contiene la base de datos.

diferencia
Operación del álgebra relacional 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.

dominio (en el contexto del modelo relacional)
Conjunto de valores atómicos.

equicombinación
Combinación en la que todas las comparaciones de la condición tienen el operador "=".

esquema de relación
Componente de una relación que consiste en un nombre de relación R y en un conjunto de atributos $\{A_1, A_2, ..., A_n\}$.

extensión de una relación de esquema $R(A_1, A_2, ..., A_n)$
Conjunto de tuplas t_i ($i = 1, 2, ..., m$) donde cada tupla t_i es un conjunto de pares $t_i = \{<A_1:V_{i1}>, <A_2V_{i2}>, ..., <A_n:V_{in}>\}$ y, para cada par $<A_j:V_{ij}>$, se cumple que v_{ij} es un valor de dominio(A_j) o bien un valor nulo.

grado de una relación
Número de atributos que pertenecen a su esquema.

inserción
Hecho de añadir una o más tuplas a una relación.

integridad
Propiedad de los datos de corresponder a representaciones plausibles del mundo real.

intersección
Operación del álgebra relacional que, a partir de dos relaciones, obtiene una nueva relación formada por las tuplas que están en las dos relaciones de partida.

lenguaje basado en el cálculo relacional
Lenguaje que proporciona un tipo de formulación de consultas fundamentado en el cálculo de predicados de la lógica matemática.

lenguaje basado en el álgebra relacional
Lenguaje que proporciona un tipo de formulación de consultas inspirado en la teoría de conjuntos.

modificación
Hecho de alterar los valores que tienen una o más tuplas de una relación para uno o más de sus atributos.

producto cartesiano
Operación del álgebra relacional 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 relación.

proyección
Operación del álgebra relacional 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.

redenominar
Operación auxiliar del álgebra relacional que permite cambiar los nombres que figuran en el esquema de una relación.

regla de integridad de dominio
Regla que establece que un valor no nulo de un atributo A_i debe pertenecer al dominio del atributo A_i, y que los operadores que es posible aplicar sobre los valores dependen de los dominios de estos valores.

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

regla de integridad de modelo
Condiciones generales que deben cumplirse en toda base de datos de un modelo determinado.

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

regla de integridad referencial
Regla que 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 *CF* de *t* son o bien valores nulos o bien valores que coinciden con los valores para *CP* de alguna tupla *s* de *S*.

relación
Elemento de la estructura de los datos de una base de datos relacional formado por un esquema (o intensión) y una extensión.

restricción en caso de modificación
Política de mantenimiento de la integridad referencial, que consiste en no permitir modificar ningún atributo de la clave primaria de una tupla si se trata de una clave primaria referenciada.

restricción en caso de borrado
Política de mantenimiento de la integridad referencial que consiste en no permitir borrar una tupla si tiene una clave primaria referenciada.

restricciones de integridad de usuario
Condiciones específicas que se deben cumplir en una base de datos concreta.

selección
Operación del álgebra relacional 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.

superclave de una relación de esquema R(A_1, A_2, ..., A_n)
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.

unión
Operación del álgebra relacional que, a partir de dos relaciones, obtiene una nueva relación formada por todas las tuplas que están en alguna de las relaciones de partida.

Bibliografía

Bibliografía básica

Date, C.J. (2001). *Introducción a los sistemas de bases de datos* (7ª ed.). Prentice-Hall.

Elmasri, R.; Navathe, S.B. (2000). *Sistemas de bases de datos. Conceptos fundamentales* (3ª ed.). Madrid: Addison-Wesley Iberoamericana.

El lenguaje SQL

Carme Martín Escofet

UOC
Formación de Posgrado

Índice

Introducción

El **SQL** es el lenguaje estándar ANSI/ISO de definición, manipulación y control de bases de datos relacionales. Es un lenguaje declarativo: sólo hay que indicar qué se quiere hacer. En cambio, en los lenguajes procedimentales es necesario especificar cómo hay que hacer cualquier acción sobre la base de datos. El SQL es un lenguaje muy parecido al lenguaje natural; concretamente, se parece al inglés, y es muy expresivo. Por estas razones, y como lenguaje estándar, el SQL es un lenguaje con el que se puede acceder a todos los sistemas relacionales comerciales.

Recordad que el álgebra relacional, que hemos visto en la unidad "El modelo relacional y el álgebra relacional", es un lenguaje procedimental.

Empezamos con una breve explicación de la forma en que el SQL ha llegado a ser el lenguaje estándar de las bases de datos relacionales:

1) Al principio de los años setenta, los laboratorios de investigación Santa Teresa de IBM empezaron a trabajar en el proyecto System R. El objetivo de este proyecto era implementar un prototipo de SGBD relacional; por lo tanto, también necesitaban investigar en el campo de los lenguajes de bases de datos relacionales. A mediados de los años setenta, el proyecto de IBM dio como resultado un primer lenguaje denominado SEQUEL (*Structured English Query Language*), que por razones legales se denominó más adelante *SQL* (*Structured Query Language*). Al final de la década de los setenta y al principio de la de los ochenta, una vez finalizado el proyecto System R, IBM y otras empresas empezaron a utilizar el SQL en sus SGBD relacionales, con lo que este lenguaje adquirió una gran popularidad.

2) En 1982, ANSI (*American National Standards Institute*) encargó a uno de sus comités (X3H2) la definición de un lenguaje de bases de datos relacionales. Este comité, después de evaluar diferentes lenguajes, y ante la aceptación comercial del SQL, eligió un lenguaje estándar que estaba basado en éste prácticamente en su totalidad. El SQL se convirtió oficialmente en el lenguaje estándar de ANSI en el año 1986, y de ISO (*International Standards Organization*) en 1987. También ha sido adoptado como lenguaje estándar por FIPS (*Federal Information Processing Standard*), Unix X/Open y SAA (*Systems Application Architecture*) de IBM.

3) En el año 1989, el estándar fue objeto de una revisión y una ampliación que dieron lugar al lenguaje que se conoce con el nombre de SQL1 o SQL89. En el año 1992 el estándar volvió a ser revisado y ampliado considerablemente para cubrir carencias de la versión anterior. Esta nueva versión del SQL, que se conoce con el nombre de **SQL2** o **SQL92**, es la que nosotros presentaremos en esta unidad didáctica.

Como veremos más adelante, aunque aparezca sólo la sigla SQL, siempre nos estaremos refiriendo al SQL92, ya que éste tiene como subconjunto el SQL89;

por lo tanto, todo lo que era válido en el caso del SQL89 lo continuará siendo en el SQL92.

De hecho, se pueden distinguir tres niveles dentro del SQL92:

1) El **nivel introductorio (*entry*)**, que incluye el SQL89 y las definiciones de *clave primaria* y *clave foránea* al crear una tabla.

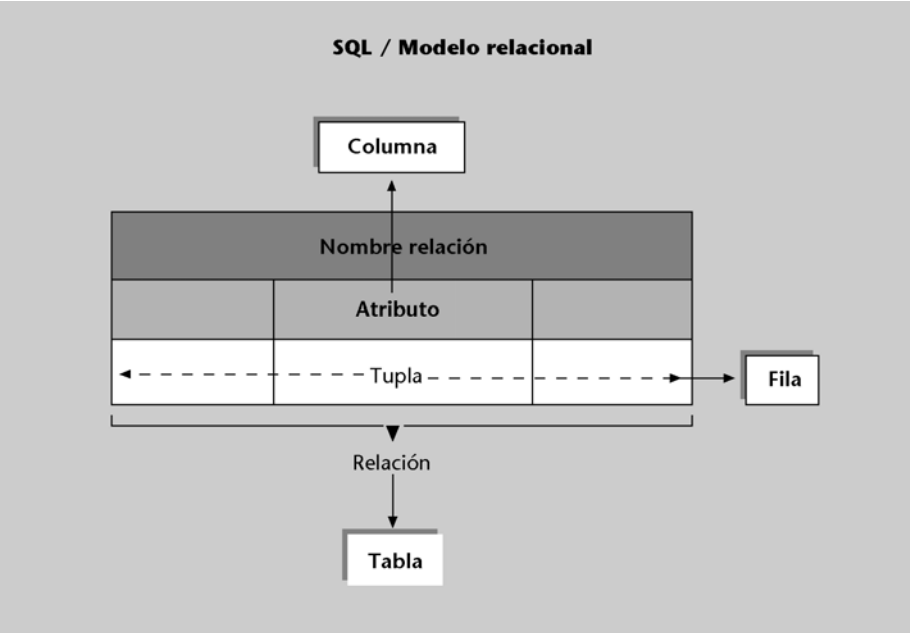
El concepto de *clave primaria* y su importancia en una relación o tabla se ha visto en la unidad "El modelo relacional y el álgebra relacional" de este curso.

2) El **nivel intermedio (*intermediate*)**, que, además del SQL89, añade algunas ampliaciones del SQL92.

3) El **nivel completo (*full*)**, que ya tiene todas las ampliaciones del SQL92.

El modelo relacional tiene como estructura de almacenamiento de los datos las relaciones. La intensión o esquema de una relación consiste en el nombre que hemos dado a la relación y un conjunto de atributos. La extensión de una relación es un conjunto de tuplas. Al trabajar con SQL, esta nomenclatura cambia, como podemos apreciar en la siguiente figura:

El modelo relacional se ha presentado en la unidad "El modelo relacional y el álgebra relacional" de este curso.

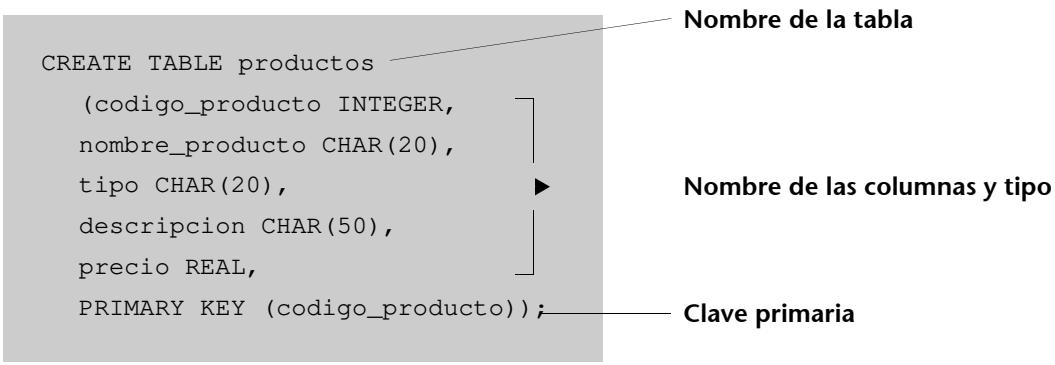


- Hablaremos de **tablas** en lugar de relaciones.
- Hablaremos de **columnas** en lugar de atributos.
- Hablaremos de **filas** en lugar de tuplas.

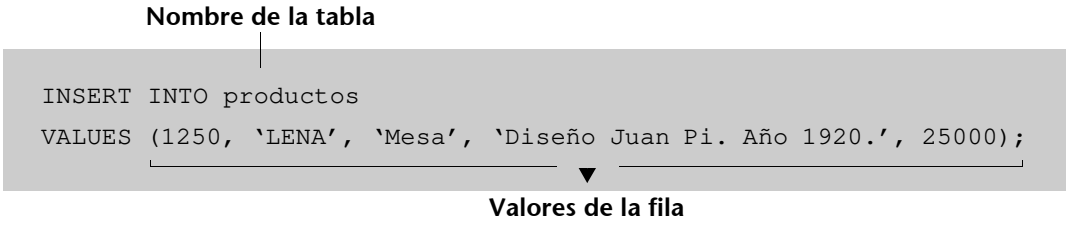
Sin embargo, a pesar de que la nomenclatura utilizada sea diferente, los conceptos son los mismos.

Con el SQL se puede definir, manipular y controlar una base de datos relacional. A continuación veremos, aunque sólo en un nivel introductorio, cómo se pueden realizar estas acciones:

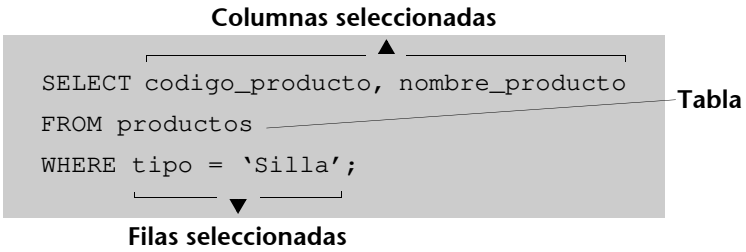
1) Sería necesario crear una tabla que contuviese los datos de los productos de nuestra empresa:



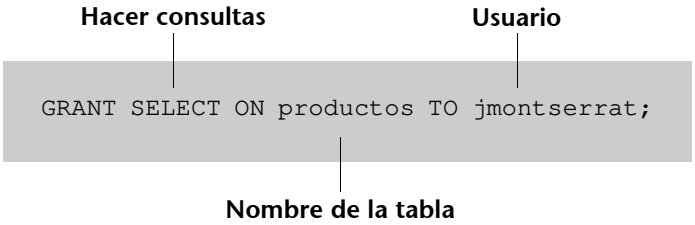
2) Insertar un producto en la tabla creada anteriormente:



3) Consultar qué productos de nuestra empresa son sillas:



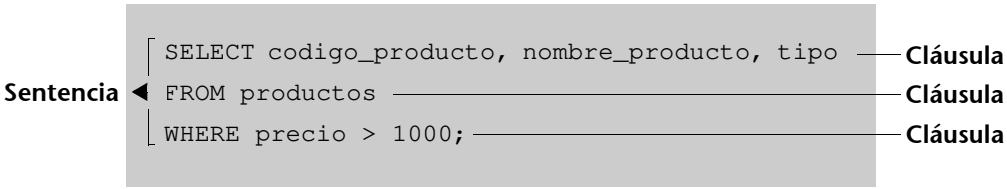
4) Dejar acceder a uno de nuestros vendedores a la información de la tabla `productos`:



Y muchas más cosas que iremos viendo punto por punto en los siguientes apartados.

Fijémonos en la estructura de todo lo que hemos hecho hasta ahora con SQL. Las operaciones de SQL reciben el nombre de **sentencias** y están formadas por

diferentes partes que denominamos **cláusulas**, tal y como podemos apreciar en el siguiente ejemplo:



Esta consulta muestra el código, el nombre y el tipo de los productos que cuestan más de 1.000 euros.

Los tres primeros apartados de este módulo tratan sobre un tipo de SQL denominado **SQL interactivo**, que permite acceder directamente a una base de datos relacional:

a) En el primer apartado definiremos las denominadas sentencias de definición, donde crearemos la base de datos, las tablas que la compondrán y los dominios, las aserciones y las vistas que queramos.

b) En el segundo aprenderemos a manipular la base de datos, ya sea introduciendo, modificando o borrando valores en las filas de las tablas, o bien haciendo consultas.

c) En el tercero veremos las sentencias de control, que aseguran un buen uso de la base de datos.

Sin embargo, muchas veces querremos acceder a la base de datos desde una aplicación hecha en un lenguaje de programación cualquiera, que nos ofrece mucha más potencia fuera del entorno de las bases de datos. Para utilizar SQL desde un lenguaje de programación necesitaremos sentencias especiales que nos permitan distinguir entre las instrucciones del lenguaje de programación y las sentencias de SQL. La idea es que trabajando básicamente con un lenguaje de programación anfitrión se puede cobijar SQL como si fuese un huésped. Por este motivo, este tipo de SQL se conoce con el nombre de **SQL hospedado**. Para trabajar con SQL hospedado necesitamos un precompilador que separe las sentencias del lenguaje de programación de las del lenguaje de bases de datos. Una alternativa a esta forma de trabajar son las **rutinas SQL/CLI*** (*SQL/Call-Level Interface*), que resolviendo también el problema de acceder a SQL desde un lenguaje de programación, no necesitan precompilador.

Introduciremos SQL hospedado y el concepto de SQL/CLI en el apartado 4 de esta unidad didáctica.

*** Las rutinas SQL/CLI se añadieron al estándar SQL92 en 1995.**

Antes de empezar a conocer el lenguaje, es necesario añadir un último comentario. Aunque SQL es el lenguaje estándar para bases de datos relacionales y ha sido ampliamente aceptado por los sistemas relacionales comerciales, no ha sido capaz de reflejar toda la teoría del modelo relacional establecida por E.F. Codd; esto lo iremos viendo a medida que profundicemos en el lenguaje.

Encontraréis la teoría del modelo relacional de E.F. Codd en la unidad "El modelo relacional y el álgebra relacional" de este curso.

Los sistemas relacionales comerciales y los investigadores de bases de datos son una referencia muy importante para mantener el estándar actualizado. En estos momentos ya se dispone de una nueva versión de SQL92 que se denomina SQL: 1999 o SQL3. SQL: 1999 tiene a SQL92 como subconjunto, e incorpora nuevas prestaciones de gran interés. En informática, en general, y particularmente en bases de datos, es necesario estar siempre al día, y por eso es muy importante tener el hábito de leer publicaciones periódicas que nos informen y nos mantengan al corriente de las novedades.

Objetivos

Una vez finalizado el estudio de los materiales didácticos de esta unidad, dispondréis de las herramientas indispensables para alcanzar los siguientes objetivos:

1. Conocer el lenguaje estándar ANSI/ISO SQL92.
2. Definir una base de datos relacional, incluyendo dominios, aserciones y vistas.
3. Saber introducir, borrar y modificar datos.
4. Ser capaz de plantear cualquier tipo de consulta a la base de datos.
5. Saber utilizar sentencias de control.
6. Conocer los principios básicos de la utilización del SQL desde un lenguaje de programación.

1. Sentencias de definición

Para poder trabajar con bases de datos relacionales, lo primero que tenemos que hacer es definirlas. Veremos las órdenes del estándar SQL92 para crear y borrar una base de datos relacional y para insertar, borrar y modificar las diferentes tablas que la componen.

En este apartado también veremos cómo se definen los dominios, las aserciones (restricciones) y las vistas. 

Vistas

Una vista en el modelo relacional no es sino una tabla virtual derivada de las tablas reales de nuestra base de datos, un esquema externo puede ser un conjunto de vistas.

La sencillez y la homogeneidad del SQL92 hacen que:

1) Para crear bases de datos, tablas, dominios, aserciones y vistas se utilice la **sentencia `CREATE`**.

2) Para modificar tablas y dominios se utilice la **sentencia `ALTER`.**

3) Para borrar bases de datos, tablas, dominios, aserciones y vistas se utilice la **sentencia `DROP`**.

La adecuación de estas sentencias a cada caso nos dará diferencias que iremos perfilando al hacer la descripción individual de cada una.

Para ilustrar la aplicación de las sentencias de SQL que veremos, utilizaremos **una base de datos de ejemplo** muy sencilla de una pequeña empresa con sede en Barcelona, Girona, Lleida y Tarragona, que se encarga de desarrollar proyectos informáticos. La información que nos interesará almacenar de esta empresa, que denominaremos *BDUOC*, será la siguiente:

1) Sobre los empleados que trabajan en la empresa, querremos saber su código de empleado, el nombre y apellido, el sueldo, el nombre y la ciudad de su departamento y el número de proyecto al que están asignados.

2) Sobre los diferentes departamentos en los que está estructurada la empresa, nos interesa conocer su nombre, la ciudad donde se encuentran y el teléfono. Será necesario tener en cuenta que un departamento con el mismo nombre puede estar en ciudades diferentes, y que en una misma ciudad puede haber departamentos con nombres diferentes.

3) Sobre los proyectos informáticos que se desarrollan, querremos saber su código, el nombre, el precio, la fecha de inicio, la fecha prevista de finalización, la fecha real de finalización y el código de cliente para quien se desarrolla.

4) Sobre los clientes para quien trabaja la empresa, querremos saber el código de cliente, el nombre, el NIF, la dirección, la ciudad y el teléfono.

1.1. Creación y borrado de una base de datos relacional

El estándar SQL92 no dispone de ninguna sentencia de creación de bases de datos. La idea es que una base de datos no es más que un conjunto de tablas y, por lo tanto, las sentencias que nos ofrece el SQL92 se concentran en la creación, la modificación y el borrado de estas tablas.

La instrucción CREATE DATABASE

Muchos de los sistemas relacionales comerciales (como ocurre en el caso de Informix, DB2, SQL Server y otros) han incorporado sentencias de creación de bases de datos con la siguiente sintaxis:

CREATE DATABASE

En cambio, disponemos de una sentencia más potente que la de creación de bases de datos: la **sentencia de creación de esquemas** denominada **CREATE SCHEMA**. Con la creación de esquemas podemos agrupar un conjunto de elementos de la base de datos que son propiedad de un usuario. La sintaxis de esta sentencia es la que tenéis a continuación:

```
CREATE SCHEMA {[nombre_esquema]} | [AUTHORIZATION usuario]}
            [lista_de_elementos_del_esquema];
```

La **nomenclatura utilizada en la sentencia** es la siguiente:

- Las palabras en negrita son palabras reservadas del lenguaje:
- La notación [...] quiere decir que lo que hay entre los corchetes se podría poner o no.
- La notación {A| ... |B} quiere decir que tenemos que elegir entre todas las opciones que hay entre las llaves, pero debemos poner una obligatoriamente.

La sentencia de creación de esquemas hace que varias tablas (lista_de_elementos_del_esquema) se puedan agrupar bajo un mismo nombre (nombre_esquema) y que tengan un propietario (usuario). Aunque todos los parámetros de la sentencia CREATE SCHEMA son opcionales, como mínimo se debe dar o bien el nombre del esquema, o bien el nombre del usuario propietario de la base de datos. Si sólo especificamos el usuario, éste será el nombre del esquema.

La creación de esquemas puede hacer mucho más que agrupar tablas, porque lista_de_elementos_del_esquema puede, además de tablas, ser también dominios, vistas, privilegios y restricciones, entre otras cosas.

La sentencia DROP DATABASE

Muchos de los sistemas relacionales comerciales (como por ejemplo Informix, DB2, SQL Server y otros) han incorporado sentencias de borrado de bases de datos con la siguiente sintaxis:

DROP DATABASE

Para borrar una base de datos encontramos el mismo problema que para crearla. El estándar SQL92 sólo nos ofrece la **sentencia de borrado de esquemas** **DROP SCHEMA**, que presenta la siguiente sintaxis:

```
DROP SCHEMA nombre_esquema {RESTRICT|CASCADE};
```

Donde tenemos lo siguiente:

- La opción de borrado de esquemas **RESTRICT** hace que el esquema sólo se pueda borrar si no contiene ningún elemento.
- La opción **CASCADE** borra el esquema aunque no esté completamente vacío.

1.2. Creación de tablas

Como ya hemos visto, la estructura de almacenamiento de los datos del modelo relacional son las tablas. Para **crear una tabla**, es necesario utilizar la **sentencia CREATE TABLE**. Veamos su formato:

Recordad que las tablas se han estudiado en la unidad "El modelo relacional y el álgebra relacional" de este curso.

```
CREATE TABLE nombre_tabla
            ( definición_columna
            [, definición_columna...]
            [, restricciones_tabla]
            );
```

Donde **definición_columna** es:

```
nombre_columna {tipo_datos|dominio} [def_defecto] [restric_col]
```

El proceso que hay que seguir para crear una tabla es el siguiente:

1) Lo primero que tenemos que hacer es decidir qué nombre queremos poner a la tabla (`nombre_tabla`).

2) Después, iremos dando el nombre de cada uno de los atributos que formarán las columnas de la tabla (`nombre_columna`).

3) A cada una de las columnas le asignaremos un tipo de datos predefinido o bien un dominio definido por el usuario. También podremos dar definiciones por defecto y restricciones de columna.

4) Una vez definidas las columnas, sólo nos quedará dar las restricciones de tabla.

1.2.1. Tipos de datos

Para cada columna tenemos que elegir entre algún dominio definido por el usuario o alguno de los tipos de datos predefinidos que se describen a continuación:

Recordad que las correspondencias entre los tipos de datos y los dominios predefinidos del modelo relacional se han visto en el subapartado 2.2 de la unidad "El modelo relacional y el álgebra relacional" de este curso.

Tipos de datos predefinidos	
Tipos de datos	**Descripción**
`CHARACTER (longitud)`	Cadenas de caracteres de longitud fija.
`CHARACTER VARYING (longitud)`	Cadenas de caracteres de longitud variable.

Tipos de datos predefinidos	
Tipos de datos	**Descripción**
`BIT (longitud)`	Cadenas de bits de longitud fija.
`BIT VARYING (longitud)`	Cadenas de bits de longitud variables.
`NUMERIC (precisión, escala)`	Número decimales con tantos dígitos como indique la precisión y tantos decimales como indique la escala.
`DECIMAL (precisión, escala)`	Número decimales con tantos dígitos como indique la precisión y tantos decimales como indique la escala.
`INTEGER`	Números enteros.
`SMALLINT`	Números enteros pequeños.
`REAL`	Números con coma flotante con precisión predefinida.
`FLOAT (precisión)`	Números con coma flotante con la precisión especificada.
`DOUBLE PRECISION`	Números con coma flotante con más precisión predefinida que la del tipo `REAL`.
`DATE`	Fechas. Están compuestas de: `YEAR año, MONTH mes, DAY día`.
`TIME`	Horas. Están compuestas de `HOUR hora, MINUT minutos, SECOND segundos`.
`TIMESTAMP`	Fechas y horas. Están compuestas de `YEAR año, MONTH mes, DAY día, HOUR hora, MINUT minutos, SECOND segundos`.

Los tipos de datos `NUMERIC` y `DECIMAL`

`NUMERIC` y `DECIMAL` se describen igual, y es posible utilizar tanto el uno como el otro para definir números decimales.

El tratamiento del tiempo

El estándar SQL92 define la siguiente nomenclatura para trabajar con el tiempo:

`YEAR`	(0001..9999)
`MONTH`	(01..12)
`DAY`	(01..31)
`HOUR`	(00..23)
`MINUT`	(00..59)
`SECOND`	(00..59.precisión)

De todos modos, los sistemas relacionales comerciales disponen de diferentes formatos, entre los cuales podemos elegir cuando tenemos que trabajar con columnas temporales.

Ejemplos de asignaciones de columnas

Veamos algunos ejemplos de asignaciones de columnas en los tipos de datos predefinidos `DATE`, `TIME` y `TIMESTAMP`:

- La columna `fecha_nacimiento` podría ser del tipo `DATE` y podría tener como valor '1978-12-25'.
- La columna `inicio_partido` podría ser del tipo `TIME` y podría tener como valor '17:15:00.000000'.
- La columna `entrada_trabajo` podría ser de tipo `TIMESTAMP` y podría tener como valor '1998-7-8 9:30:05'.

1.2.2. Creación, modificación y borrado de dominios

Además de los dominios dados por el tipo de datos predefinidos, el SQL92 nos ofrece la posibilidad de trabajar con dominios definidos por el usuario.

Dominios definidos por el usuario

Aunque el SQL92 nos ofrece la sentencia `CREATE DOMAIN`, hay pocos sistemas relacionales comerciales que nos permitan utilizarla.

Para **crear un dominio** es necesario utilizar la sentencia **`CREATE DOMAIN`**:

```
CREATE DOMAIN nombre dominio [AS] tipos_datos
             [def_defecto] [restricciones_dominio];
```

donde **`restricciones_dominio`** tiene el siguiente formato:

```
[CONSTRAINT nombre_restricción] CHECK (condiciones)
```

Explicaremos la construcción de condiciones más adelante, en el subapartado 2.5 cuando hablemos de cómo se hacen consultas a una base de datos. Veremos `def_defecto` en el subapartado 1.2.3 de esta unidad.

Creación de un dominio en BDUOC

Si quisiéramos definir un dominio para las ciudades donde se encuentran los departamentos de la empresa BDUOC, haríamos:

```
CREATE DOMAIN dom_ciudades AS CHAR (20)
CONSTRAINT ciudades_validas
CHECK (VALUE IN ('Barcelona', 'Tarragona', 'Lleida', 'Girona'));
```

De este modo, cuando definimos la columna `ciudades` dentro de la tabla departamentos no se tendrá que decir que es de tipo `CHAR (20)`, sino de tipo `dom_ciudades`. Esto nos debería asegurar, según el modelo relacional, que sólo haremos operaciones sobre la columna `ciudades` con otras columnas que tengan este mismo dominio definido por el usuario; sin embargo, el SQL92 no nos ofrece herramientas para asegurar que las comparaciones que hacemos sean entre los mismos dominios definidos por el usuario.

Por ejemplo, si tenemos una columna con los nombres de los empleados definida sobre el tipo de datos `CHAR (20)`, el SQL nos permite compararla con la columna `ciudades`, aunque semánticamente no tenga sentido. En cambio, según el modelo relacional, esta comparación no se debería haber permitido.

Para **borrar un dominio definido por el usuario** es preciso utilizar la **sentencia `DROP DOMAIN`**, que tiene este formato:

```
DROP DOMAIN nombre_dominio {RESTRICT|CASCADE};
```

En este caso, tenemos que:

- La opción de borrado de dominios **`RESTRICT`** hace que el dominio sólo se pueda borrar si no se utiliza en ningún sitio.
- La opción **`CASCADE`** borra el dominio aunque esté referenciado, y pone el tipo de datos del dominio allí donde se utilizaba.

Borrar un dominio de BDUOC

Si quisiéramos borrar el dominio que hemos creado antes para las ciudades donde se encuentran los departamentos de la empresa BDUOC, haríamos:

```
DROP DOMAIN dom_ciudades RESTRICT;
```

En este caso nos deberíamos asegurar de que ninguna columna está definida sobre `dom_ciudades` antes de borrar el dominio.

Para **modificar un dominio semántico** es necesario utilizar la **sentencia `ALTER DOMAIN`**. Veamos su formato:

```
ALTER DOMAIN nombre_dominio {acción_modificar_dominio|
                             acción_modif_restricción_dominio};
```

Donde tenemos lo siguiente:

- **acción_modificar_dominio** puede ser:

```
{SET def_defecto|DROP DEFAULT}
```

- **acción_modif_restricción_dominio** puede ser:

```
{ADD restricciones_dominio|DROP CONSTRAINT nombre_restricción}
```

Modificar un dominio en BDUOC

Si quisiéramos añadir una nueva ciudad (Mataró) al dominio que hemos creado antes para las ciudades donde se encuentran los departamentos de la empresa BDUOC, haríamos:

```
ALTER DOMAIN dom_ciudades DROP CONSTRAINT ciudades_validas;
```

Con esto hemos eliminado la restricción de dominio antigua. Y ahora tenemos que introducir la nueva restricción:

```
ALTER_DOMAIN dom_ciudades ADD CONSTRAINT ciudades_validas
CHECK (VALUE IN ('Barcelona', 'Tarragona', 'Lleida', 'Girona', 'Mataro'));
```

1.2.3. Definiciones por defecto

Ya hemos visto en otros módulos la importancia de los valores nulos y su inevitable aparición como valores de las bases de datos.

La opción **def_defecto** nos permite especificar qué nomenclatura queremos dar a nuestros valores por omisión.

Por ejemplo, para un empleado que todavía no se ha decidido cuánto ganará, podemos elegir que, de momento, tenga un sueldo de 0 euros (`DEFAULT 0.0`), o bien que tenga un sueldo con un valor nulo (`DEFAULT NULL`).

Sin embargo, hay que tener en cuenta que si elegimos la **opción DEFAULT NULL**, la columna para la que daremos la definición por defecto de valor nulo debería admitir valores nulos.

La opción **DEFAULT** tiene el siguiente formato:

```
DEFAULT (literal|función|NULL)
```

La posibilidad más utilizada y la opción por defecto, si no especificamos nada, es la palabra reservada **`NULL`**. Sin embargo, también podemos definir nuestro propio literal, o bien recurrir a una de las funciones que aparecen en la tabla siguiente:

Función	Descripción
`{USER|CURRENT_USER}`	Identificador del usuario actual
`SESSION_USER`	Identificador del usuario de esta sesión
`SYSTEM_USER`	Identificador del usuario del sistema operativo
`CURRENT_DATE`	Fecha actual
`CURRENT_TIME`	Hora actual
`CURRENT_TIMESTAMP`	Fecha y hora actuales

1.2.4. Restricciones de columna

En cada una de las columnas de la tabla, una vez les hemos dado un nombre y hemos definido su dominio, podemos imponer ciertas restricciones que siempre se tendrán que cumplir. Las restricciones que se pueden dar son las que aparecen en la tabla que tenemos a continuación:

Restricciones de columna	
Restricción	**Descripción**
`NOT NULL`	La columna no puede tener valores nulos.
`UNIQUE`	La columna no puede tener valores repetidos. Es una clave alternativa.
`PRIMARY KEY`	La columna no puede tener valores repetidos ni nulos. Es la clave primaria.
`REFERENCES tabla [(columna)]`	La columna es la clave foránea de la columna de la tabla especificada.
`CHECK (condiciones)`	La columna debe cumplir las condiciones especificadas.

1.2.5. Restricciones de tabla

Una vez hemos dado un nombre, hemos definido una tabla y hemos impuesto ciertas restricciones para cada una de las columnas, podemos aplicar restricciones sobre toda la tabla, que siempre se deberán cumplir. Las restricciones que se pueden dar son las siguientes:

Restricciones de tabla	
Restricción	**Descripción**
`UNIQUE (columna [, columna...])`	El conjunto de las columnas especificadas no puede tener valores repetidos. Es una clave alternativa.

Restricciones de tabla	
Restricción	**Descripción**
`PRIMARY KEY (columna [, columna...])`	El conjunto de las columnas especificadas no puede tener valores nulos ni repetidos. Es una clave primaria.
`FOREIGN KEY (columna [, columna...]) REFERENCES tabla [(columna2 [, columna2...])]`	El conjunto de las columnas especificadas es una clave foránea que referencia la clave primaria formada por el conjunto de las columnas2 de la tabla dada. Si las columnas y las columnas2 se denominan exactamente igual, entonces no sería necesario poner columnas2.
`CHECK (condiciones)`	La tabla debe cumplir las condiciones especificadas.

1.2.6. Modificación y borrado de claves primarias con claves foráneas que hacen referencia a éstas

En otra unidad de este curso hemos visto tres políticas aplicables a los casos de borrado y modificación de filas que tienen una clave primaria referenciada por claves foráneas. Estas políticas eran la restricción, la actualización en cascada y la anulación.

Para recordar las políticas que se pueden aplicar a los casos de borrado y modificación de las filas, consultad los subapartados 4.3.1, 4.3.2 y 4.3.3 de la unidad "El modelo relacional y el álgebra relacional" de este curso.

El SQL nos ofrece la posibilidad de especificar, al definir una clave foránea, qué política queremos seguir. Veamos su formato:

```
CREATE TABLE nombre_tabla
            ( definición_columna
            [, definición_columna. . .]
            [, restricciones_tabla]
            );
```

Donde una de las restricciones de tabla era la definición de claves foráneas, que tiene el siguiente formato:

```
FOREIGN KEY clave_secundaria REFERENCES tabla [(clave_primaria)]
[ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
[ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
```

Donde `NO ACTION` corresponde a la política de restricción; `CASCADE`, a la actualización en cascada, y `SET NULL` sería la anulación. `SET DEFAULT` se podría considerar una variante de `SET NULL`, donde en lugar de valores nulos se puede poner el valor especificado por defecto.

1.2.7. Aserciones

Una aserción es una restricción general que hace referencia a una o más columnas de más de una tabla. Para **definir una aserción** se utiliza la **sentencia** **`CREATE ASSERTION`**, y tiene el siguiente formato:

```
CREATE ASSERTION nombre_aserción CHECK (condiciones);
```

Crear una aserción en BDUOC

Creamos una aserción sobre la base de datos BDUOC que nos asegure que no hay ningún empleado con un sueldo superior a 80.000 asignado al proyecto SALSA:

```
CREATE ASSERTION restriccion1 CHECK (NOT EXISTS (SELECT *
                              FROM proyectos p, empleados e
                              WHERE p.codigo_proyec =
                              = e.num_proyec and e.sueldo > 8.0E+4
                              and p.nom_proj = 'SALSA') );
```

Para **borrar una aserción** es necesario utilizar la **sentencia** **`DROP ASSERTION`**, que presenta este formato:

```
DROP ASSERTION nombre_aserción;
```

Borrar una aserción en BDUOC

Por ejemplo, para borrar la aserción `restriccion1`, utilizaríamos la sentencia `DROP ASSERTION` de la forma siguiente:

```
DROP ASSERTION restriccion1;
```

1.3. Modificación y borrado de tablas

Para **modificar una tabla** es preciso utilizar la **sentencia** **`ALTER TABLE`**. Veamos su formato:

```
ALTER TABLE nombre_tabla {acción_modificar_columna|
                          acción_modif_restricción_tabla};
```

En este caso, tenemos que:

- **acción_modificar_columna** puede ser:

```
{ADD [COLUMN] columna def_columna |
ALTER [COLUMN] columna {SET def_defecto|DROP DEFAULT}|
DROP [COLUMN ] columna {RESTRICT|CASCADE}}
```

- **acción_modif_restricción_tabla** puede ser:

```
{ADD restricción|
DROP CONSTRAINT restricción {RESTRICT|CASCADE}}
```

Si queremos modificar una tabla es que queremos realizar una de las siguientes operaciones:

1) Añadirle una columna (`ADD columna`).

2) Modificar las definiciones por defecto de la columna (`ALTER columna`).

3) Borrar la columna (`DROP columna`).

4) Añadir alguna nueva restricción de tabla (`ADD restricción`).

5) Borrar alguna restricción de tabla (`DROPCONSTRAINT restricción`).

Para **borrar una tabla** es preciso utilizar la **sentencia `DROP TABLE`**:

```
DROP TABLE nombre_tabla {RESTRICT|CASCADE};
```

En este caso tenemos que:

- Si utilizamos la opción **`RESTRICT`**, la tabla no se borrará si está referenciada, por ejemplo, por alguna vista.
- Si usamos la opción **`CASCADE`**, todo lo que referencie a la tabla se borrará con ésta.

1.4. Creación y borrado de vistas

Como hemos observado, la arquitectura ANSI/SPARC distingue tres niveles, que se describen en el esquema conceptual, el esquema interno y los esquemas externos. Hasta ahora, mientras creábamos las tablas de la base de datos, íbamos

Los tres niveles de la arquitectura ANSI/SPARC se han estudiado en el subapartado 4.1 de la unidad "Introducción a las bases de datos" de este curso.

describiendo el esquema conceptual. Para describir los diferentes esquemas externos utilizamos el concepto de vista del SQL.

Para **crear una vista** es necesario utilizar la **sentencia `CREATE VIEW`**. Veamos su formato:

```
CREATE VIEW nombre_vista [(lista_columnas)] AS (consulta)
            [WITH CHECK OPTION];
```

Lo primero que tenemos que hacer para crear una vista es decidir qué nombre le queremos poner (`nombre_vista`). Si queremos cambiar el nombre de las columnas, o bien poner nombre a alguna que en principio no tenía, lo podemos hacer en `lista_columnas`. Y ya sólo nos quedará definir la consulta que formará nuestra vista.

Por lo que respecta a la construcción de consultas, consultad el subapartado 2.5 de esta unidad didáctica.

Las vistas no existen realmente como un conjunto de valores almacenados en la base de datos, sino que son tablas ficticias, denominadas *derivadas* (no materializadas). Se construyen a partir de tablas reales (materializadas) almacenadas en la base de datos, y conocidas con el nombre de *tablas básicas* (o tablas de base). La no-existencia real de las vistas hace que puedan ser actualizables o no.

Creación de una vista en BDUOC

Creamos una vista sobre la base de datos BDUOC que nos dé para cada cliente el número de proyectos que tiene encargados el cliente en cuestión.

```
CREATE VIEW proyectos_por_cliente (codigo_cli, numero_proyectos) AS
(SELECT c.codigo_cli, COUNT(*)
 FROM proyectos p, clientes c
 WHERE p.codigo_cliente = c.codigo_cli
 GROUP BY c.codigo_cli);
```

Si tuviésemos las siguientes extensiones:

- Tabla `clientes`:

clientes					
codigo_cli	**nombre_cli**	**nif**	**direccion**	**ciudad**	**telefono**
10	ECIGSA	38.567.893-C	Aragón 11	Barcelona	NULL
20	CME	38.123.898-E	Valencia 22	Girona	972.23.57.21
30	ACME	36.432.127-A	Mallorca 33	Lleida	973.23.45.67

- Tabla `proyectos`:

proyectos						
codigo_proyec	**nombre_proyec**	**precio**	**fecha_inicio**	**fecha_prev_fin**	**fecha_fin**	**codigo_cliente**
1	GESCOM	1,0E+6	1-1-98	1-1-99	NULL	10
2	PESCI	2,0E+6	1-10-96	31-3-98	1-5-98	10

proyectos						
codigo_proyec	**nombre_proyec**	**precio**	**fecha_inicio**	**fecha_prev_fin**	**fecha_fin**	**codigo_cliente**
3	SALSA	1,0E+6	10-2-98	1-2-99	NULL	20
4	TINELL	4,0E+6	1-1-97	1-12-99	NULL	30

Y mirásemos la extensión de la vista `proyectos_por_clientes`, veríamos lo que encontramos en el margen.

proyectos_por_clientes	
codigo_cli	**numero_proyectos**
10	2
20	1
30	1

En las vistas, además de hacer consultas, podemos insertar, modificar y borrar filas.

Actualización de vistas en BDUOC

Si alguien insertase en la vista `proyectos_por_cliente`, los valores para un nuevo cliente 60 con tres proyectos encargados, encontraríamos que estos tres proyectos tendrían que figurar realmente en la tabla `proyectos` y, por lo tanto, el SGBD los debería insertar con la información que tenemos, que es prácticamente inexistente. Veamos gráficamente cómo quedarían las tablas después de esta hipotética actualización, que no llegaremos a hacer nunca, ya que iría en contra de la teoría del modelo relacional:

- Tabla `clientes`

clientes					
codigo_cli	**nombre_cli**	**nif**	**direccion**	**ciudad**	**telefono**
10	ECIGSA	38.567.893-C	Aragón 11	Barcelona	NULL
20	CME	38.123.898-E	Valencia 22	Girona	972.23.57.21
30	ACME	36.432.127-A	Mallorca 33	Lleida	973.23.45.67
60	NULL	NULL	NULL	NULL	NULL

- Tabla proyectos:

proyectos						
codigo_proyec	**nombre_proyec**	**precio**	**fecha_inicio**	**fecha_prev_fin**	**fecha_fin**	**codigo_cliente**
1	GESCOM	1,0E+6	1-1-98	1-1-99	NULL	10
2	PESCI	2,0E+6	1-10-96	31-3-98	1-5-98	10
3	SALSA	1,0E+6	10-2-98	1-2-99	NULL	20
NULL	NULL	NULL	NULL	NULL	NULL	60
NULL	NULL	NULL	NULL	NULL	NULL	60
NULL	NULL	NULL	NULL	NULL	NULL	60

El SGBD no puede actualizar la tabla básica `clientes` si sólo sabe la clave primaria, y todavía menos la tabla básica `proyectos` sin la clave primaria; por lo tanto, esta vista no sería actualizable.

En cambio, si definimos una vista para saber los clientes que tenemos en Barcelona o en Girona, haríamos:

```
CREATE VIEW clientes_Barcelona_Girona AS
(SELECT *
 FROM clientes
 WHERE ciudad IN ('Barcelona', 'Girona'))
WHITH CHECK OPTION;
```

Si queremos asegurarnos de que se cumpla la condición de la cláusula `WHERE`, debemos poner la opción **`WHITH CHECK OPTION`**. Si no lo hiciésemos, podría ocurrir que alguien incluyese en la vista `clientes_Barcelona_Girona` a un cliente nuevo con el código 70, de nombre JMB, con el NIF 36.788.224-C, la dirección en NULL, la ciudad Lleida y el teléfono NULL.

Si consultásemos la extensión de la vista `clientes_Barcelona_Girona`, veríamos:

clientes_Barcelona_Girona					
codigo_cli	**nombre_cli**	**nif**	**direccion**	**ciudad**	**telefono**
10	ECIGSA	38.567.893-C	Aragón 11	Barcelona	NULL
20	CME	38.123.898-E	Valencia 22	Girona	972.23.57.21

Esta vista sí podría ser actualizable. Podríamos insertar un nuevo cliente con código 50, de nombre CEA, con el NIF 38.226.777-D, con la dirección París 44, la ciudad Barcelona y el teléfono 93.422.60.77. Después de esta actualización, en la tabla básica `clientes` encontraríamos, efectivamente:

clientes					
codigo_cli	**nombre_cli**	**nif**	**direccion**	**ciudad**	**telefono**
10	ECIGSA	38.567.893-C	Aragón 11	Barcelona	NULL
20	CME	38.123.898-E	Valencia 22	Girona	972.23.57.21
30	ACME	36.432.127-A	Mallorca 33	Lleida	973.23.45.67
50	CEA	38.226.777-D	París, 44	Barcelona	93.442.60.77

Para borrar una vista es preciso utilizar la **sentencia `DROP VIEW`**, que presenta el formato:

```
DROP VIEW nombre_vista (RESTRICT|CASCADE);
```

Si utilizamos la opción **`RESTRICT`**, la vista no se borrará si está referenciada, por ejemplo, por otra vista. En cambio, si ponemos la opción **`CASCADE`**, todo lo que referencie a la vista se borrará con ésta.

Borrar una vista en BDUOC

Para borrar la vista `clientes_Barcelona_Girona`, haríamos lo siguiente:

```
DROP VIEW clientes_Barcelona_Girona RESTRICT;
```

1.5. Definición de la base de datos relacional BDUOC

Veamos cómo se crearía la base de datos BDUOC, utilizando, por ejemplo, un SGBD relacional que disponga de la sentencia `CREATE DATABASE`:

Orden de creación

Antes de crear una tabla con una o más claves foráneas, se deben haber creado las tablas que tienen como clave primaria las referenciadas por las foráneas.

```
CREATE DATABASE bduoc;
CREATE TABLE clientes
```

```
   (codigo_cli INTEGER,
   nombre_cli CHAR(30) NOT NULL,
   nif CHAR (12),
   direccion CHAR (30),
   ciudad CHAR (20),
   telefono CHAR (12),
   PRIMARY KEY (codigo_cli),
   UNIQUE(nif)
 );

CREATE TABLE departamentos
   (nombre_dep CHAR(20) PRIMARY KEY,*
   ciudad_dep CHAR(20),
   telefono INTEGER DEFAULT NULL,
   PRIMARY KEY (nombre_dep, ciudad_dep)
 );

CREATE TABLE proyectos
   (codigo_proyec INTEGER,
   nombre_proyec CHAR(20),
   precio REAL,
   fecha_inicio DATE,
   fecha_prev_fin DATE,
   fecha_fin DATE DEFAULT NULL,
   codigo_cliente INTEGER,
   PRIMARY KEY (codigo_proyec),
   FOREIGN KEY codigo_cliente REFERENCES clientes (codigo_cli),
   CHECK (fecha_inicio < fecha_prev_fin),
   CHECK (fecha_inicio < fecha_fin)
 );

CREATE TABLE empleados
   (codigo_empl INTEGER,
   nombre_empl CHAR (20),
   apellido_empl CHAR(20),
   sueldo REAL CHECK (sueldo > 7000),
   nombre_dep CHAR(20)
   ciudad_dep CHAR(20),
   num_proyec INTEGER,
   PRIMARY KEY (codigo_empl),
   FOREIGN KEY (nombre_dep, ciudad_dep) REFERENCES
   departamentos (nombre_dep, ciudad_dep),
   FOREIGN KEY (num_proyec) REFERENCES proyectos (codigo_proyec)
 );

COMMIT;
```

*** Tenemos que elegir restricción de tabla porque la clave primaria está compuesta por más de un atributo.**

La sentencia `COMMIT` se explica en el subapartado 3.1 de esta unidad didáctica.

Al crear una tabla vemos que muchas restricciones se pueden imponer de dos formas: como restricciones de columna o como restricciones de tabla. Por ejemplo, cuando queremos decir cuál es la clave primaria de una tabla, tenemos las dos posibilidades. Esto se debe a la flexibilidad del SQL:

- En el caso de que la restricción haga referencia a un solo atributo, podemos elegir la posibilidad que más nos guste.

- En el caso de la tabla `departamentos`, tenemos que elegir por fuerza la opción de restricciones de tabla, porque la clave primaria está compuesta por más de un atributo.

En general, lo pondremos todo como restricciones de tabla, excepto `NOT NULL` y `CHECK` cuando haga referencia a una sola columna. 

2. Sentencias de manipulación

Una vez creada la base de datos con sus tablas, debemos poder insertar, modificar y borrar los valores de las filas de las tablas. Para poder hacer esto, el SQL92 nos ofrece las siguientes sentencias: **INSERT** para insertar, **UPDATE** para modificar y **DELETE** para borrar. Una vez hemos insertado valores en nuestras tablas, tenemos que poder consultarlos. La sentencia para hacer consultas a una base de datos con el SQL92 es **SELECT FROM**. Veamos a continuación estas sentencias.

2.1. Inserción de filas en una tabla

Antes de poder consultar los datos de una base de datos, es preciso **introducirlos** con la sentencia **INSER TINTO VALUES**, que tiene el formato:

```
INSERT INTO nombre_tabla [(columnas)]
{VALUES ({v1|DEFAULT|NULL}, ..., {vn/DEFAULT/NULL})|<consulta>};
```

Inserción de múltiples filas

Para insertar más de una fila con una sola sentencia, tenemos que obtener los valores como resultado de una consulta realizada en una o más tablas.

Los valores `v1`, `v2`, ..., `vn` se deben corresponder exactamente con las columnas que hemos dicho que tendríamos con el `CREATE TABLE` y deben estar en el mismo orden, a menos que las volvamos a poner a continuación del nombre de la tabla. En este último caso, los valores se deben disponer de forma coherente con el nuevo orden que hemos impuesto. Podría darse el caso de que quisiéramos que algunos valores para insertar fuesen valores por omisión, definidos previamente con la opción `DEFAULT`. Entonces pondríamos la palabra reservada `DEFAULT`. Si se trata de introducir valores nulos, también podemos utilizar la palabra reservada `NULL`.

Inserción de una fila en BDUOC

La forma de insertar a un cliente en la tabla `clientes` de la base de datos de BDUOC es:

```
INSERT INTO clientes
VALUES (10, 'ECIGSA', '37.248.573-C', 'ARAGON 242', 'Barcelona', DEFAULT);
```

o bien:

```
INSERT INTO clientes (nif, nombre_cli, codigo_cli, telefono, direccion,
ciudad)
VALUES ('37.248.573-C', 'ECIGSA', 10, DEFAULT, 'ARAGON 242', 'Barcelona');
```

2.2. Borrado de filas de una tabla

Para **borrar valores de algunas filas de una tabla** podemos utilizar la **sentencia `DELETE FROM WHERE`**. Su formato es el siguiente:

```
DELETE FROM nombre_tabla
[WHERE condiciones];
```

En cambio, si lo que quisiéramos conseguir es **borrar todas las filas de una tabla**, entonces sólo tendríamos que poner la sentencia **`DELETE FROM`**, sin `WHERE`.

Borrar todas las filas de una tabla en BDUOC

Podemos dejar la tabla `proyectos` sin ninguna fila:

```
DELETE FROM proyectos;
```

En nuestra base de datos, borrar los proyectos del cliente 2 se haría de la forma que mostramos a continuación:

```
DELETE FROM proyectos
WHERE codigo_cliente = 2;
```

Borrado de múltiples filas

Notemos que el cliente con el código 2 podría tener más de un proyecto contratado y, por lo tanto, se borraría más de una fila con una sola sentencia.

2.3. Modificación de filas de una tabla

Si quisiéramos **modificar los valores de algunas filas de una tabla**, tendríamos que utilizar la sentencia **`UPDATE SET WHERE`**. A continuación presentamos su formato:

```
UPDATE nombre_tabla
SET columna = {expresión|DEFAULT|NULL}
    [, columna = {expr|DEFAULT|NULL} ...]
WHERE condiciones;
```

Modificación de los valores de algunas filas en BDUOC

Supongamos que queremos incrementar el sueldo de todos los empleados del proyecto 2 en 1.000 euros. La modificación a ejecutar sería:

```
UPDATE empleados
SET sueldo = sueldo + 1000
WHERE num_proyec = 2;
```

Modificación de múltiples filas

Notemos que el proyecto número 2 podría tener a más de un empleado asignado y, por lo tanto, se modificaría la columna `sueldo`, de más de una fila con una sola sentencia.

2.4. Introducción de filas en la base de datos relacional BDUOC

Antes de empezar a hacer consultas a la base de datos BDUOC, habremos introducido unas cuantas filas en sus tablas con la sentencia `INSERT INTO`. De esta forma, podremos ver reflejado el resultado de las consultas que iremos haciendo, a partir de este momento, sobre cada extensión; esto lo podemos observar en las tablas correspondientes a cada extensión, que presentamos a continuación:

- **Tabla** `departamentos`:

departamentos		
nombre_dep	**ciudad_dep**	**telefono**
DIR	Barcelona	93.422.60.70
DIR	Girona	972.23.89.70
DIS	Lleida	973.23.50.40
DIS	Barcelona	93.224.85.23
PROG	Tarragona	977.33.38.52
PROG	Girona	972.23.50.91

- **Tabla** `clientes`:

clientes					
codigo_cli	**nombre_cli**	**nif**	**direccion**	**ciudad**	**telefono**
10	ECIGSA	38.567.893-C	Aragón 11	Barcelona	NULL
20	CME	38.123.898-E	Valencia 22	Girona	972.23.57.21
30	ACME	36.432.127-A	Mallorca 33	Lleida	973.23.45.67
40	JGM	38.782.345-B	Rosellón 44	Tarragona	977.33.71.43

- **Tabla** `empleados`:

empleados						
codigo_empleado	**nombre_empl**	**apellido_empl**	**sueldo**	**nombre_dep**	**ciudad_dep**	**num_proyec**
1	María	Puig	1,0E+5	DIR	Girona	1
2	Pedro	Mas	9,0E+4	DIR	Barcelona	4
3	Ana	Ros	7,0E+4	DIS	Lleida	3
4	Jorge	Roca	7,0E+4	DIS	Barcelona	4
5	Clara	Blanc	4,0E+4	PROG	Tarragona	1
6	Laura	Tort	3,0E+4	PROG	Tarragona	3
7	Rogelio	Salt	4,0E+4	NULL	NULL	4
8	Sergio	Grau	3,0E+4	PROG	Tarragona	NULL

- **Tabla** `proyectos`:

proyectos						
codigo_proyec	**nombre_proyec**	**precio**	**fecha_inicio**	**fecha_prev_fin**	**fecha_fin**	**codigo_cliente**
1	GESCOM	1,0E+6	1-1-98	1-1-99	NULL	10
2	PESCI	2,0E+6	1-10-96	31-3-98	1-5-98	10
3	SALSA	1,0E+6	10-2-98	1-2-99	NULL	20
4	TINELL	4,0E+6	1-1-97	1-12-99	NULL	30

2.5. Consultas a una base de datos relacional

Para hacer **consultas** sobre una tabla con el SQL es preciso utilizar la **sentencia** **`SELECT FROM`**, que tiene el siguiente formato:

```
SELECT nombre_columna_a_seleccionar [[AS] col_renombrada]
[,nombre_columna_a_seleccionar [[AS] col_renombrada]...]
FROM tabla_a_consultar [[AS] tabla_renombrada];
```

La opción **`AS`** nos permite renombrar las columnas que queremos seleccionar o las tablas que queremos consultar que en este caso, es sólo una. Dicho de otro modo, nos permite la definición de alias. Fijémonos en que la palabra clave `AS` es opcional, y es bastante habitual poner sólo un espacio en blanco en lugar de toda la palabra.

Consultas a BDUOC

A continuación presentamos un ejemplo de consulta a la base de datos BDUOC para conocer todos los datos que aparece en la tabla `clientes`:

```
SELECT *
FROM clientes;
```

El * después de `SELECT` indica que queremos ver todos los atributos que aparecen en la tabla.

La respuesta a esta consulta sería:

codigo_cli	**nombre_cli**	**nif**	**direccion**	**ciudad**	**telefono**
10	ECIGSA	38.567.893-C	Aragón 11	Barcelona	NULL
20	CME	38.123.898-E	Valencia 22	Girona	972.23.57.21
30	ACME	36.432.127-A	Mallorca 33	Lleida	973.23.45.67
40	JGM	38.782.345-B	Rosellón 44	Tarragona	977.33.71.43

Si hubiésemos querido ver sólo el código, el nombre, la dirección y la ciudad, habríamos hecho:

```
SELECT codigo_cli, nombre_cli, direccion, ciudad
FROM clientes;
```

Y habríamos obtenido la respuesta siguiente:

codigo_cli	nombre_cli	direccion	ciudad
10	ECIGSA	Aragón 11	Barcelona
20	CME	Valencia 22	Girona
30	ACME	Mallorca 33	Lleida
40	JGM	Rosellón 44	Tarragona

Con la sentencia `SELECT FROM` podemos seleccionar columnas de una tabla, pero para seleccionar filas de una tabla es preciso añadirle la cláusula `WHERE`. El formato es:

```
SELECT nombre_columnas_a_seleccionar
FROM tabla_a_consultar
WHERE condiciones;
```

La cláusula `WHERE` nos permite obtener las filas que cumplen la condición especificada en la consulta.

Consultas a BDUOC seleccionando filas

Veamos un ejemplo en el que pedimos "los códigos de los empleados que trabajan en el proyecto número 4":

```
SELECT codigo_empl
FROM empleados
WHERE num_proyec = 4;
```

La respuesta a esta consulta sería la que podéis ver en el margen.

codigo_empl
2
4
7

Para definir las condiciones en la cláusula `WHERE`, podemos utilizar alguno de los operadores de los que dispone el SQL, que son los siguientes:

Operadores de comparación	
=	Igual
<	Menor
>	Mayor
<=	Menor o igual
>=	Mayor o igual
< >	Diferente

Operadores lógicos	
NOT	Para la negación de condiciones
AND	Para la conjunción de condiciones
OR	Para la disyunción de condiciones

Si queremos que en una consulta nos aparezcan las filas resultantes sin repeticiones, es preciso poner la palabra clave **`DISTINCT`** inmediatamente después

de SELECT. También podríamos explicitar que lo queremos todo, incluso con repeticiones, poniendo **ALL** (opción por defecto) en lugar de DISTINCT. El formato de DISTINCT es:

```
SELECT DISTINCT nombre_columnas_a_seleccionar
FROM tabla_a_consultar
[WHERE condiciones];
```

Consulta a BDUOC seleccionando filas sin repeticiones

Por ejemplo, si quisiéramos ver qué sueldos se están pagando en nuestra empresa, podríamos hacer:

```
SELECT DISTINCT sueldo
FROM empleados;
```

La respuesta a esta consulta, sin repeticiones, sería la que aparece en el margen.

sueldo
3,0E+4
4,0E+4
7,0E+4
9,0E+4
1,0E+5

2.5.1. Funciones de agregración

El SQL nos ofrece las siguientes funciones de agregación para efectuar varias operaciones sobre los datos de una base de datos:

Funciones de agregación	
Función	**Descripción**
COUNT	Nos da el número total de filas seleccionadas
SUM	Suma los valores de una columna
MIN	Nos da el valor mínimo de una columna
MAX	Nos da el valor máximo de una columna
AVG	Calcula el valor medio de una columna

En general, las funciones de agregación se aplican a una columna, excepto la función de agregación COUNT, que normalmente se aplica a todas las columnas de la tabla o tablas seleccionadas. Por lo tanto, COUNT (*) contará todas las filas de la tabla o las tablas que cumplan las condiciones. Si se utilizase COUNT(distinct columna), sólo contaría los valores que no fuesen nulos ni repetidos, y si se utilizase COUNT (columna), sólo contaría los valores que no fuesen nulos.

Ejemplo de utilización de la función COUNT (*)

Veamos un ejemplo de uso de la función COUNT, que aparece en la cláusula SELECT, para hacer la consulta "¿Cuántos departamentos están ubicados en la ciudad de Lleida?":

```
SELECT COUNT(*) AS numero_dep
FROM departamentos
WHERE ciudad_dep = 'Lleida';
```

La respuesta a esta consulta sería la que aparece reflejada en la tabla que encontraréis en el margen.

numero_dep
1

Veremos ejemplos de las demás funciones de agregación en los siguientes apartados.

2.5.2. Subconsultas

Una subconsulta es una consulta incluida dentro de una cláusula `WHERE` o `HAVING` de otra consulta. En ocasiones, para expresar ciertas condiciones no hay más remedio que obtener el valor que buscamos como resultado de una consulta.

Veremos la cláusula `HAVING` en el subapartado 2.5.5 de esta unidad didáctica.

Subconsulta en BDUOC

Si quisiéramos saber los códigos y los nombres de los proyectos de precio más elevado, en primer lugar tendríamos que encontrar los proyectos que tienen el precio más elevado. Lo haríamos de la forma siguiente:

```
SELECT codigo_proyec, nombre_proyec
FROM proyectos
WHERE precio = (SELECT MAX(precio)
                FROM proyectos);
```

El resultado de la consulta anterior sería lo que puede verse al margen.

codigo_proyec	nombre_proyec
4	TINELL

Los proyectos de precio más bajo

Si en lugar de los códigos y los nombres de proyectos de precio más alto hubiésemos querido saber los de precio más bajo, habríamos aplicado la función de agregación `MIN`.

2.5.3. Otros predicados

1) Predicado `BETWEEN`

Para expresar una condición que quiere encontrar un valor entre unos límites concretos, podemos utilizar `BETWEEN`:

```
SELECT nombre_columnas_a_seleccionar
FROM tabla_a_consultar
WHERE columna BETWEEN límite1 AND límite2;
```

Ejemplo de uso del predicado `BETWEEN`

Un ejemplo en el que se pide "Los códigos de los empleados que ganan entre 20.000 y 50.000 euros anuales" sería:

```
SELECT codigo_empl
FROM empleados
WHERE sueldo BETWEEN 2.0E+4 and 5.0E+4;
```

La respuesta a esta consulta sería la que se ve en el margen.

codigo_empl
5
6
7
8

2) Predicado IN

Para comprobar si un valor coincide con los elementos de una lista utilizaremos IN, y para ver si no coincide, NOT IN:

```
SELECT nombre_columnas_a_seleccionar
FROM tabla_a_consultar
WHERE columna [NOT] IN (valor1, ..., valorN);
```

Ejemplo de uso del predicado IN

"Queremos saber el nombre de todos los departamentos que se encuentran en las ciudades de Lleida o Tarragona":

```
SELECT nombre_dep, ciudad_dep
FROM departamentos
WHERE ciudad_dep IN ('Lleida', 'Tarragona');
```

La respuesta sería la que aparece en el margen.

nombre_dep	ciudad_dep
DIS	Lleida
PROG	Tarragona

3) Predicado LIKE

Para comprobar si una columna de tipo carácter cumple alguna propiedad determinada, podemos usar LIKE:

```
SELECT nombre_columnas_a_seleccionar
FROM tabla_a_consultar
WHERE columna LIKE característica;
```

Los **patrones del SQL92 para expresar características** son los siguientes:

Otros patrones

Aunque _ y % son los caracteres elegidos por el estándar, cada sistema relacional comercial ofrece diversas variantes.

a) Pondremos un carácter _ para cada carácter individual que queramos considerar.

b) Pondremos un carácter % para expresar una secuencia de caracteres, que puede no estar formada por ninguno.

Ejemplo de uso del predicado LIKE

A continuación presentamos un ejemplo en el que buscaremos los nombres de los empleados que empiezan por J, y otro ejemplo en el que obtendremos los proyectos que comienzan por S y tienen cinco letras:

a) Nombres de empleados que empiezan por la letra J:

Atributos añadidos

Aunque la consulta pide sólo los nombres de empleados añadimos el código para poder diferenciar dos empleados con el mismo nombre.

```
SELECT codigo_empl, nombre_empl
FROM empleados
WHERE nombre_empl LIKE 'J%';
```

codigo_empl	nombre_empl
4	Jorge

La respuesta a esta consulta sería la que se muestra en el margen.

b) Proyectos que empiezan por S y tienen cinco letras:

```
SELECT codigo_proyec
FROM proyectos
WHERE nombre_proyec LIKE 'S_ _ _ _';
```

codigo_proyec
3

Y la respuesta a esta otra consulta sería la que aparece en el margen.

4) Predicado `IS NULL`

Para comprobar si un valor es nulo utilizaremos `IS NULL`, y para averiguar si no lo es, `IS NOT NULL`. El formato es:

```
SELECT nombre_columnas_a_seleccionar
FROM tabla_a_consultar
WHERE columna IS [NOT] NULL;
```

Ejemplo de uso del predicado `IS NULL`

Un ejemplo de uso de este predicado sería "Queremos saber el código y el nombre de todos los empleados que no están asignados a ningún proyecto":

```
SELECT codigo_empl, nombre_empl
FROM empleados
WHERE num_proyec IS NULL;
```

codigo_empl	nombre_empl
8	Sergio

Obtendríamos la respuesta que tenemos al margen.

5) Predicados `ANY/SOME` y `ALL`

Para ver si una columna cumple que todas sus filas (`ALL`) o algunas de sus filas (`ANY/SOME`) satisfagan una condición, podemos hacer:

Los predicados `ANY/SOME`

Podemos elegir cualquiera de los dos predicados para pedir que alguna fila satisfaga una condición.

```
SELECT nombre_columnas_a seleccionar
FROM tabla_a_consultar
WHERE columna operador_comparación {ALL|ANY|SOME}subconsulta;
```

Ejemplo de uso de los predicados `ALL` y `ANY/SOME`

a) Veamos un ejemplo de aplicación de `ALL` para encontrar los códigos y los nombres de los proyectos en los que los sueldos de todos los empleados asignados son menores que el precio del proyecto:

```
SELECT codigo_proyec, nombre_proyec
FROM proyectos
WHERE precio > ALL (SELECT sueldo
                    FROM empleados
                    WHERE codigo_proyec = num_proyec);
```

Fijémonos en la condición de WHERE de la subconsulta, que nos asegura que los sueldos que observamos son los de los empleados asignados al proyecto de la consulta. La respuesta a esta consulta sería la que aparece en el margen.

codigo_proyec	nombre_proyec
1	GESCOM
2	PESCI
3	SALSA
4	TINELL

b) A continuación, presentamos un ejemplo de ANY/SOME para buscar los códigos y los nombres de los proyectos que tienen algún empleado que gana un sueldo más elevado que el precio del proyecto en el que trabaja.

```
SELECT codigo_proyec, nombre_proyec
FROM proyectos
WHERE precio < ANY (SELECT sueldo
                    FROM empleados
                    WHERE codigo_proyec = num_proyec);
```

codigo_proyec	nombre_proyec

La respuesta a esta consulta está vacía, como se ve en el margen.

6) Predicado EXISTS

Para comprobar si una subconsulta produce alguna fila de resultados, podemos utilizar la sentencia denominada *test de existencia*: EXISTS. Para comprobar si una subconsulta no produce ninguna fila de resultados, podemos utilizar NOT EXISTS.

```
SELECT nombre_columnas_a_seleccionar
FROM tabla_a_consultar
WHERE [NOT] EXISTS subconsulta;
```

Ejemplo de uso del predicado EXISTS

Un ejemplo en el que se buscan los códigos y los nombres de los empleados que están asignados a algún proyecto sería:

codigo_empl	nombre_empl
1	María
2	Pedro
3	Ana
4	Jorge
5	Clara
6	Laura
7	Rogelio

```
SELECT codigo_empl, nombre_empl
FROM empleados
WHERE EXISTS (SELECT *
              FROM proyectos
              WHERE codigo_proyec = num_proyec);
```

La respuesta a esta consulta sería la que se muestra en el margen.

2.5.4. Ordenación de los datos obtenidos en respuestas a consultas

Si se desea que, al hacer una consulta, los datos aparezcan en un orden determinado, es preciso utilizar la **cláusula ORDER BY** en la sentencia SELECT, que presenta el siguiente formato:

```
SELECT nombre_columnas_a seleccionar
FROM tabla_a_consultar
```

```
[WHERE condiciones]
ORDER BY columna_según_la_cual_se_quiere_ordenar [DESC]
        [, col_ordenación [DESC]...];
```

Consulta a BDUOC con respuesta ordenada

Imaginemos que queremos consultar los nombres de los empleados ordenados según el sueldo que ganan, y si ganan el mismo sueldo, ordenados alfabéticamente por el nombre:

```
SELECT codigo_empl, nombre_empl, apellido_empl, sueldo
FROM empleados
ORDER BY sueldo, nombre_empl;
```

Esta consulta daría la respuesta siguiente:

codigo_empl	nombre_empl	apellido_empl	sueldo
6	Laura	Tort	3,0E+4
8	Sergio	Grau	3,0E+4
5	Clara	Blanc	4,0E+4
7	Rogelio	Salt	4,0E+4
3	Ana	Ros	7,0E+4
4	Jorge	Roca	7,0E+4
2	Pedro	Mas	9,0E+4
1	María	Puig	1,0E+5

Si no se especifica nada más, se seguirá un orden ascendente, pero si se desea seguir un orden descendente es necesario añadir `DESC` detrás de cada factor de ordenación expresado en la cláusula `ORDER BY`:

```
ORDER BY columna_ordenación [DESC] [, columna [DESC] ...];
```

También se puede explicitar un orden ascendente poniendo la palabra clave **`ASC`** (opción por defecto).

2.5.5. Consultas con agrupación de filas de una tabla

Las cláusulas siguientes, añadidas a la instrucción `SELECT FROM`, permiten organizar las filas por grupos:

a) La **cláusula `GROUP BY`** nos sirve para agrupar filas según las columnas que indique esta cláusula.

b) La **cláusula `HAVING`** especifica condiciones de búsqueda para grupos de filas; lleva a cabo la misma función que antes cumplía la cláusula `WHERE` para las filas de toda la tabla, pero ahora las condiciones se aplican a los grupos obtenidos.

Presenta el siguiente formato:

```
SELECT nombre_columnas_a seleccionar
FROM tabla_a_consultar
[WHERE condiciones]
GROUP BY columnas_según_las_cuales_se_quiere_agrupar
[HAVING condiciones_por_grupos]
[ORDER BY columna_ordenación [DESC] [, columna [DESC]...]];
```

Notemos que en las sentencias SQL se van añadiendo cláusulas a medida que la dificultad o la exigencia de la consulta lo requiere.

Consulta con agrupación de filas en BDUOC

Imaginemos que queremos saber el sueldo medio que ganan los empleados de cada departamento:

```
SELECT nombre_dep, ciudad_dep, AVG(sueldo) AS sueldo_medio
FROM empleados
GROUP BY nombre_dep, ciudad_dep;
```

El resultado de esta consulta sería:

nombre_dep	ciudad_dep	sueldo_medio
DIR	Barcelona	9,0E + 4
DIR	Girona	1,0E + 5
DIS	Lleida	7,0E + 4
DIS	Barcelona	7,0E + 4
PROG	Tarragona	3,3E + 4
NULL	NULL	4,0E + 4

Factores de agrupación

Los factores de agrupación de la cláusula `GROUP BY` deben ser, como mínimo, las columnas que figuran en `SELECT`, exceptuando las columnas afectadas por funciones de agregación.

Ejemplo de uso de la función de agregación `SUM`

Veamos un ejemplo de uso de una función de agregación `SUM` del SQL que aparece en la cláusula `HAVING` de `GROUP BY`: "Queremos saber los códigos de los proyectos en los que la suma de los sueldos de los empleados es mayor que 180.000 euros":

```
SELECT num_proyec
FROM empleados
GROUP BY num_proyec
HAVING SUM (sueldo) >1.8E+5;
```

El resultado de esta consulta sería el que se ve al margen.

num_proyec
4

`DISTINCT` y `GROUP BY`

En este ejemplo no es necesario poner `DISTINCT`, a pesar de que la columna `num_proyec` no es atributo identificador. Fijémonos en que en la tabla `empleados` hemos puesto que todos los proyectos tienen el mismo código juntos en un mismo grupo y no es posible que aparezcan repetidos.

2.5.6. Consultas a más de una tabla

Muchas veces queremos consultar datos de más de una tabla haciendo combinaciones de columnas de tablas diferentes. En el SQL es posible listar más de una tabla que se quiere consultar especificándolo en la cláusula `FROM`.

1) Combinación

La combinación consigue crear una sola tabla a partir de las tablas especificadas en la cláusula `FROM`, haciendo coincidir los valores de las columnas relacionadas de estas tablas.

Recordad que la misma operación de combinación, pero del álgebra relacional, se ha visto en el subapartado 5.3.3. de la unidad "El modelo relacional y el álgebra relacional" de este curso.

Ejemplo de combinación en BDUOC

A continuación mostramos un ejemplo con la base de datos BDUOC en el que queremos saber el NIF del cliente y el código y el precio del proyecto que desarrollamos para el cliente número 20:

```
SELECT proyectos.codigo_proyecto, proyectos.precio, clientes.nif
FROM clientes, proyectos
WHERE clientes.codigo_cli = proyectos.codigo_cliente AND clientes.
codigo_cli = 20;
```

El resultado sería:

proyectos.codigo_proyecto	proyectos.precio	clientes.nif
3	1,0E+6	38.123.898-E

Si trabajamos con más de una tabla, puede ocurrir que la tabla resultante tenga dos columnas con el mismo nombre. Por ello es obligatorio especificar a qué tabla corresponden las columnas a las que nos estamos refiriendo, denominando la tabla a la que pertenecen antes de ponerlas (por ejemplo, `clientes.codigo_cli`). Para simplificarlo, se utilizan los alias que, en este caso, se definen en la cláusula `FROM`.

Ejemplo de alias en BDUOC

`c` podría ser el alias de la tabla `clientes`. De este modo, para indicar a qué tabla pertenece `codigo_cli`, sólo haría falta poner: `c.codigo_cli`.

Veamos cómo quedaría la consulta anterior expresada mediante alias, aunque en este ejemplo no serían necesarios, porque todas las columnas de las dos tablas tienen nombres diferentes. Pediremos, además, las columnas `c.codigo_cli` y `p.codigo_cliente`.

```
SELECT p.codigo_proyecto, p.precio, c.nif, p.codigo_cliente, c.codigo_cli
FROM clientes c, proyectos p
WHERE c.codigo_cli = p.codigo_cliente AND c.codigo_cli = 20;
```

Entonces obtendríamos este resultado:

p.codigo_proyec	p.precio	c.nif	p.codigo_cliente	c.codigo_cli
3	1,0E+6	38.123.898-E	20	20

Notemos que en `WHERE` necesitamos expresar el vínculo que se establece entre las dos tablas, en este caso `codigo_cli` de `clientes` y `codigo_cliente` de `proyectos`. Expresado en operaciones del álgebra relacional, esto significa que hacemos una combinación en lugar de un producto cartesiano.

Fijémonos en que, al igual que en álgebra relacional, la operación que acabamos de hacer es una equicombinación (*equi-join*); por lo tanto, nos aparecen dos columnas idénticas: `c.codigo_cli` y `p.codigo_cliente`.

Las operaciones del álgebra relacional se han visto en el apartado 5 de la unidad "El modelo relacional y el álgebra relacional" de este curso.

La forma de expresar la combinación que acabamos de ver pertenece al SQL92 introductorio. Una forma alternativa de realizar la equicombinación anterior, utilizando el SQL92 intermedio o completo, sería la siguiente:

```
SELECT nombre_columnas_a_seleccionar
FROM tabla1 JOIN tabla2
     {ON condiciones|USING (columna [, columna...])}
[WHERE condiciones];
```

Ejemplo anterior con el SQL92 intermedio o completo

El ejemplo que hemos expuesto antes utilizando el SQL92 intermedio o completo sería:

```
SELECT p.codigo_proyecto, p.precio, c.nif, p.codigo_cliente, c.codigo_cli
FROM clientes c JOIN proyectos p ON c.codigo_cli = p.codigo_cliente
WHERE c.codigo_cli = 20;
```

Y obtendríamos el mismo resultado de antes.

La opción `ON`, además de expresar condiciones con la igualdad, en el caso de que las columnas que queramos vincular tengan nombres diferentes, nos ofrece la posibilidad de expresar condiciones con los demás operadores de comparación que no sean el de igualdad. Sería el equivalente a la operación que en álgebra relacional hemos denominado θ-*combinación* (θ-*join*).

Podéis ver la equicombinación y la θ-combinación en el subapartado 5.3.3 de la unidad "El modelo relacional y el álgebra relacional" de este curso.

También podemos utilizar una misma tabla dos veces con alias diferentes, para distinguirlas.

Dos alias para una misma tabla en BDUOC

Si pidiésemos los códigos y los apellidos de los empleados que ganan más que el empleado que tiene por código el número 5, haríamos lo siguiente:

```
SELECT e1.codigo_empl, e1.apellido_empl
FROM empleados e1 JOIN empleados e2 ON e1.sueldo > e2.sueldo
WHERE e2.codigo_empl = 5;
```

Hemos tomado la tabla `e2` para fijar la fila del empleado con código número 5, de modo que podamos comparar el sueldo de la tabla `e1`, que contiene a todos los empleados, con el sueldo de la tabla `e2`, que contiene sólo al empleado 5.

La respuesta a esta consulta sería:

e1.codigo_empl	e1.apellido_empl
1	Puig
2	Mas
3	Ros
4	Roca

2) Combinación natural

La combinación natural (*natural join*) de dos tablas consiste básicamente, al igual que en el álgebra relacional, en hacer una equicombinación entre columnas del mismo nombre y eliminar las columnas repetidas. La combinación natural, utilizando el SQL92 intermedio o completo, se haría de la forma siguiente:

```
SELECT nombre_columnas_a_seleccionar
FROM tabla1 NATURAL JOIN tabla2
[WHERE condiciones];
```

Combinación natural en BDUOC

Veamos a continuación un ejemplo en el que las columnas para las que se haría la combinación natural se denominan igual en las dos tablas. Ahora queremos saber el código y el nombre de los empleados que están asignados al departamento cuyo teléfono es 977.33.38.52:

```
SELECT codigo_empl, nombre_empl
FROM empleados NATURAL JOIN departamentos
WHERE telefono = '977.333.852';
```

La combinación natural también se podría hacer con la cláusula `USING`, sólo aplicando la palabra reservada `JOIN`:

```
SELECT codigo_empl, nombre_empl
FROM empleados JOIN departamentos USING (nombre_dep, ciudad_dep)
WHERE telefono = '977.333.852';
```

La respuesta que daría sería:

empleados.codigo_empl	empleados.nombre_empl
5	Clara
6	Laura
8	Sergio

3) Combinación interna y externa

Cualquier combinación puede ser interna o externa:

a) La **combinación interna** (*inner join*) sólo se queda con las filas que tienen valores idénticos en las columnas de las tablas que compara. Esto puede hacer que perdamos alguna fila interesante de alguna de las dos tablas; por ejemplo,

porque se encuentra a NULL en el momento de hacer la combinación. Su formato es el siguiente:

```
SELECT nombre_columnas_a_seleccionar
FROM t1 [NATURAL] [INNER] JOIN t2
       {ON condiciones|
       |USING (columna [,columna...])}
[WHERE condiciones];
```

b) Por ello disponemos de la **combinación externa** (*outer join*), que nos permite obtener todos los valores de la tabla que hemos puesto a la derecha, los de la tabla que hemos puesto a la izquierda o todos los valores de las dos tablas. Su formato es:

```
SELECT nombre_columnas_a_seleccionar
FROM t1 [NATURAL] [LEFT|RIGHT|FULL] [OUTER] JOIN t2
       {ON condiciones|
       [USING (columna [,columna...])}
[WHERE condiciones];
```

Combinación natural interna en BDUOC

Si quisiéramos vincular con una combinación natural interna las tablas `empleados` y `departamentos` para saber el código y el nombre de todos los empleados y el nombre, la ciudad y el teléfono de todos los departamentos, haríamos:

Combinación interna

Aunque en el ejemplo estamos haciendo una combinación natural interna, no es necesario poner la palabra `INNER`, ya que es la opción por defecto.

```
SELECT e.codigo_empl, e.nombre_empl, e.nombre_dep, e.ciudad_dep, d.telefono
FROM empleados e NATURAL JOIN departamentos d;
```

Y obtendríamos el siguiente resultado:

e.codigo_empl	e.nombre_empl	e.nombre_dep	e.ciudad_dep	d.telefono
1	María	DIR	Girona	972.23.89.70
2	Pedro	DIR	Barcelona	93.422.60.70
3	Ana	DIS	Lleida	973.23.50.40
4	Jorge	DIS	Barcelona	93.224.85.23
5	Clara	PROG	Tarragona	977.33.38.52
6	Laura	PROG	Tarragona	977.33.38.52
8	Sergio	PROG	Tarragona	977.33.38.52

Fijémonos en que en el resultado no aparece el empleado número 7, que no está asignado a ningún departamento, ni el departamento de programación de Girona, que no tiene ningún empleado asignado.

Combinación natural externa a BDUOC

En los ejemplos siguientes veremos cómo varían los resultados que iremos obteniendo según los tipos de combinación externa:

a) Combinación externa izquierda

```
SELECT e.codigo_empl, e.nombre_empl, e.nombre_dep, e.ciudad_dep, d.telefono
FROM empleados e NATURAL LEFT OUTER JOIN departamentos d;
```

El resultado sería el que podemos ver a continuación:

e.codigo_empl	e.nombre_empl	e.nombre_dep	e.ciudad_dep	d.telefono
1	María	DIR	Girona	972.23.89.70
2	Pedro	DIR	Barcelona	93.422.60.70
3	Ana	DIS	Lleida	973.23.50.40
4	Jorge	DIS	Barcelona	93.224.85.23
5	Clara	PROG	Tarragona	977.33.38.52
6	Laura	PROG	Tarragona	977.33.38.52
7	Rogelio	NULL	NULL	NULL
8	Sergio	PROG	Tarragona	977.33.38.52

Combinación externa izquierda

Aquí figura el empleado 7.

b) Combinación externa derecha

```
SELECT e.codigo_empl, e.nombre_empl, e.nombre_dep, e.ciudad_dep, d.telefono
FROM empleados e NATURAL RIGHT OUTER JOIN departamentos d;
```

Obtendríamos este resultado:

e.codigo_empl	e.nombre_empl	e.nombre_dep	e.ciudad_dep	d.telefono
1	María	DIR	Girona	972.23.89.70
2	Pedro	DIR	Barcelona	93.422.60.70
3	Ana	DIS	Lleida	973.23.50.40
4	Jorge	DIS	Barcelona	93.224.85.23
5	Clara	PROG	Tarragona	977.33.38.52
6	Laura	PROG	Tarragona	977.33.38.52
8	Sergio	PROG	Tarragona	977.33.38.52
NULL	NULL	PROG	Girona	972.23.50.91

Combinación externa derecha

Aquí figura el departamento de programación de Girona.

c) Combinación externa plena

```
SELECT e.codigo_empl, e.nombre_empl, e.nombre_dep, e.ciudad_dep,
d.telefono
FROM empleados e NATURAL FULL OUTER JOIN departamentos d;
```

Y obtendríamos el siguiente resultado:

e.codigo_empl	e.nombre_empl	e.nombre_dep	e.ciudad_dep	d.telefono
1	María	DIR	Girona	972.23.89.70
2	Pedro	DIR	Barcelona	93.422.60.70
3	Ana	DIS	Lleida	973.23.50.40

e.codigo_empl	e.nombre_empl	e.nombre_dep	e.ciudad_dep	d.telefono
4	Jorge	DIS	Barcelona	93.224.85.23
5	Clara	PROG	Tarragona	977.33.38.52
6	Laura	PROG	Tarragona	977.33.38.52
7	Rogelio	NULL	NULL	NULL
8	Sergio	PROG	Tarragona	977.33.38.52
NULL	NULL	PROG	Girona	972.23.50.91

Combinación externa plena

Aquí figura el empleado 7 y el departamento de programación de Girona.

4) Combinaciones con más de dos tablas

Si queremos combinar tres tablas o más con el SQL92 introductorio, sólo tenemos que añadir todas las tablas en el `FROM` y los vínculos necesarios en el `WHERE`. Si queremos combinarlas con el SQL92 intermedio o con el completo, tenemos que ir haciendo combinaciones de tablas por pares, y la tabla resultante se convertirá en el primer componente del siguiente par.

Combinaciones con más de dos tablas en BDUOC

Veamos ejemplos de los dos casos, suponiendo que queremos combinar las tablas `empleados`, `proyectos` y `clientes`:

```
SELECT *
FROM empleados, proyectos, clientes
WHERE num_proyec = codigo_proyec AND codigo_cliente = codigo_cli;
```

o bien:

```
SELECT *
FROM (empleados JOIN proyectos ON num_proyec = codigo_proyec)
JOIN clientes ON codigo_cliente = codigo_cli;
```

2.5.7. La unión

La cláusula **`UNION`** permite unir consultas de dos o más sentencias `SELECT FROM`. Su formato es:

```
SELECT columnas
FROM tabla
[WHERE condiciones]
UNION [ALL]
SELECT columnas
FROM tabla
[WHERE condiciones];
```

Si ponemos la opción `ALL`, aparecerán todas las filas obtenidas a causa de la unión. No la pondremos si queremos eliminar las filas repetidas. Lo más

importante de la unión es que somos nosotros quienes tenemos que procurar que se efectúe entre columnas definidas sobre dominios compatibles; es decir, que tengan la misma interpretación semántica. Como ya hemos comentado, el SQL92 no nos ofrece herramientas para asegurar la compatibilidad semántica entre columnas.

Utilización de la unión en BDUOC

Si queremos saber todas las ciudades que hay en nuestra base de datos, podríamos hacer:

```
SELECT ciudad
FROM clientes
UNION
SELECT ciudad_dep
FROM departamentos;
```

ciudad
Barcelona
Girona
Lleida
Tarragona

El resultado de esta consulta sería el que se muestra al margen.

2.5.8. La intersección

Para hacer la intersección entre dos o más sentencias `SELECT FROM`, podemos utilizar la cláusula **`INTERSECT`**, cuyo formato es:

```
SELECT columnas
FROM tabla
[WHERE condiciones]
INTERSECT [ALL]
SELECT columnas
FROM tabla
[WHERE condiciones];
```

Si indicamos la opción `ALL`, aparecerán todas las filas obtenidas a partir de la intersección. No la pondremos si queremos eliminar las filas repetidas.

Lo más importante de la intersección es que somos nosotros quienes tenemos que vigilar que se haga entre columnas definidas sobre dominios compatibles; es decir, que tengan la misma interpretación semántica.

Utilización de la intersección en BDUOC

Si queremos saber todas las ciudades donde tenemos departamentos en los que podamos encontrar algún cliente, podríamos hacer:

```
SELECT ciudad
FROM clientes
INTERSECT
SELECT ciudad_dep
FROM departamentos;
```

ciudad
Barcelona
Girona
Lleida
Tarragona

El resultado de esta consulta sería el que se muestra al margen.

Sin embargo, la intersección es una de las operaciones del SQL que se puede hacer de más formas diferentes. También podríamos encontrar la intersección con `IN` o `EXISTS`:

a) Intersección utilizando `IN`

```
SELECT columnas
FROM tabla
WHERE columna IN (SELECT columna
                  FROM tabla
                  [WHERE condiciones]);
```

b) Intersección utilizando `EXISTS`

```
SELECT columnas
FROM tabla
WHERE EXISTS (SELECT *
              FROM tabla
              WHERE condiciones);
```

Ejemplo anterior expresado con `IN` y con `EXISTS`

El ejemplo que hemos propuesto antes se podría expresar con `IN`:

```
SELECT c.ciudad
FROM clientes c
WHERE c.ciudad IN (SELECT d.ciudad_dep
                   FROM departamentos d);
```

o también con `EXISTS`:

```
SELECT c.ciudad
FROM clientes c
WHERE EXISTS (SELECT *
              FROM departamentos d
              WHERE c.ciudad = d.ciudad_dep;
```

2.5.9. La diferencia

Para encontrar la diferencia entre dos o más sentencias `SELECT FROM` podemos utilizar la cláusula **`EXCEPT`**, que tiene este formato:

```
SELECT columnas
FROM tabla
```

```
[WHERE condiciones]
EXCEPT [ALL]
SELECT columnas
FROM tabla
[WHERE condiciones];
```

Si ponemos la opción `ALL` aparecerán todas las filas que da la diferencia. No la pondremos si queremos eliminar las filas repetidas.

Lo más importante de la diferencia es que somos nosotros quienes tenemos que vigilar que se haga entre columnas definidas sobre dominios compatibles.

Utilización de la diferencia en BDUOC

Si queremos saber los clientes que no nos han contratado ningún proyecto, podríamos hacer:

```
SELECT codigo_cli
FROM clientes
EXCEPT
SELECT codigo_cliente
FROM proyectos;
```

El resultado de esta consulta sería el que se ve en el margen.

codigo_cli
40

La diferencia es, junto con la intersección, una de las operaciones del SQL que se puede realizar de más formas diferentes. También podríamos encontrar la diferencia utilizando `NOT IN` o `NOT EXISTS`:

a) Diferencia utilizando `NOT IN`:

```
SELECT columnas
FROM tabla
WHERE columna NOT IN (SELECT columna
                      FROM tabla
                      [WHERE condiciones]);
```

b) Diferencia utilizando `NOT EXISTS`:

```
SELECT columnas
FROM tabla
WHERE NOT EXISTS (SELECT *
                  FROM tabla
                  WHERE condiciones);
```

Ejemplo anterior expresado con `NOT IN` y con `NOT EXISTS`

El ejemplo que hemos hecho antes se podría expresar con `NOT IN`:

```
SELECT c.codigo_cli
FROM clientes c
WHERE c.codigo_cli NOT IN (SELECT p.codigo_cliente
                           FROM proyectos p);
```

o también con `NOT EXISTS`

```
SELECT c.codigo_cli
FROM clientes c
WHERE NOT EXISTS (SELECT *
                  FROM proyectos p
                  WHERE c.codigo_cli = p.codigo_cliente);
```

3. Sentencias de control

Además de definir y manipular una base de datos relacional, es importante que se establezcan **mecanismos de control** para resolver problemas de concurrencia de usuarios y garantizar la seguridad de los datos. Para la concurrencia de usuarios utilizaremos el concepto de *transacción*, y para la seguridad veremos cómo se puede autorizar y desautorizar a usuarios a acceder a la base de datos.

3.1. Las transacciones

Una transacción es una unidad lógica de trabajo. O informalmente, y trabajando con SQL, un conjunto de sentencias que se ejecutan como si fuesen una sola. En general, las sentencias que forman parte de una transacción se interrelacionan entre sí, y no tiene sentido que se ejecute una sin que se ejecuten las demás.

La mayoría de las transacciones se inician de forma implícita al utilizar alguna sentencia que empieza con `CREATE`, `ALTER`, `DROP`, `SET`, `DECLARE`, `GRANT` o `REVOKE`, aunque existe la sentencia SQL para **iniciar transacciones**, que es la siguiente:

```
SET TRANSACTION {READ ONLY|READ WRITE};
```

Si queremos actualizar la base de datos utilizaremos la opción `READ WRITE`, y si no la queremos actualizar, elegiremos la opción `READ ONLY`.

Sin embargo, en cambio, una transacción siempre debe acabar explícitamente con alguna de las sentencias siguientes:

```
{COMMIT|ROLLBACK} [WORK];
```

La diferencia entre `COMMIT` y `ROLLBACK` es que mientras la sentencia `COMMIT` confirma todos los cambios producidos contra la BD durante la ejecución de la transacción, la sentencia `ROLLBACK` deshace todos los cambios que se hayan producido en la base de datos y la deja como estaba antes del inicio de nuestra transacción.

La palabra reservada `WORK` sólo sirve para aclarar lo que hace la sentencia, y es totalmente opcional.

Ejemplo de transacción

A continuación proponemos un ejemplo de transacción en el que se quiere disminuir el sueldo de los empleados que han trabajado en el proyecto 3 en 1.000 euros. y aumentar el sueldo de los empleados que han trabajado en el proyecto 1 también en 1.000 euros.

```
SET TRANSACTION READ WRITE;
UPDATE empleados SET sueldo = sueldo - 1000 WHERE num_proyec = 3;
UPDATE empleados SET sueldo = sueldo + 1000 WHERE num_proyec = 1;
COMMIT;
```

3.2. Las autorizaciones y desautorizaciones

Todos los privilegios sobre la base de datos los tiene su propietario, pero no es el único que accede a ésta. Por este motivo, el SQL nos ofrece sentencias para autorizar y desautorizar a otros usuarios.

1) Autorizaciones

Para autorizar, el SQL dispone de la siguiente sentencia:

```
GRANT privilegios ON objeto TO usuarios
[WITH GRANT OPTION];
```

Donde tenemos que:

a) **`privilegios`** puede ser:

- `ALL PRIVILEGES`: todos los privilegios sobre el objeto especificado.
- `USAGE`: utilización del objeto especificado; en este caso el dominio.
- `SELECT`: consultas.
- `INSERT` [(`columnas`)]: inserciones. Se puede concretar de qué `columnas`.
- `UPDATE` [(`columnas`)]: modificaciones. Se puede concretar de qué `columnas`.
- `DELETE`: borrados.
- `REFERENCES` [(`columna`)]: referencia del objeto en restricciones de integridad. Se puede concretar de qué `columnas`.

b) **`Objeto`** debe ser:

- `DOMAIN`: dominio

- `TABLE`: tabla.

- `Vista`.

c) **Usuarios** puede ser todo el mundo: `PUBLIC`, o bien una lista de los identificadores de los usuarios que queremos autorizar.

d) La opción **`WITH GRANT OPTION`** permite que el usuario que autoricemos pueda, a su vez, autorizar a otros usuarios a acceder al objeto con los mismos privilegios con los que ha sido autorizado.

2) Desautorizaciones

Para desautorizar, el SQL dispone de la siguiente sentencia:

```
REVOKE [GRANT OPTION FOR] privilegios ON objeto FROM
usuarios [RESTRICT|CASCADE];
```

Donde tenemos que:

a) `privilegios`, `objeto` y `usuarios` son los mismos que para la sentencia `GRANT`.

b) La opción **`GRANT OPTION FOR`** se utilizaría en el caso de que quisiéramos eliminar el derecho a autorizar (`WITH GRANT OPTION`).

c) Si un usuario al que hemos autorizado ha autorizado a su vez a otros, que al mismo tiempo pueden haber hecho más autorizaciones, la **opción `CASCADE`** hace que queden desautorizados todos a la vez.

d) La opción **`RESTRICT`** no nos permite desautorizar a un usuario si éste ha autorizado a otros.

4. Sublenguajes especializados

Muchas veces querremos acceder a la base de datos desde una aplicación hecha en un lenguaje de programación cualquiera. Para utilizar el SQL desde un lenguaje de programación, podemos utilizar el **SQL hospedado**, y para trabajar con éste necesitamos un precompilador que separe las sentencias del lenguaje de programación de las del lenguaje de bases de datos. Una alternativa muy interesante a esta forma de trabajar son las **rutinas SQL/CLI**.

El objetivo de este apartado no es explicar con detalle ni el SQL hospedado ni, aún menos, las rutinas SQL/CLI. Sólo introduciremos las ideas básicas del funcionamiento de ambos.

4.1. SQL hospedado

Para crear y manipular una base de datos relacional necesitamos SQL. Además, si la tarea que queremos hacer requiere el poder de procesamiento de un lenguaje de programación como Java, C, Cobol, Fortran, Pascal, etc., podemos utilizar el SQL hospedado en el lenguaje de programación elegido. De este modo, podemos utilizar las sentencias del SQL dentro de nuestras aplicaciones, poniendo siempre delante la palabra reservada **`EXEC SQL*`**.

*** Puede haber pequeñas diferencias dependiendo del lenguaje de programación concreto que estemos considerando.**

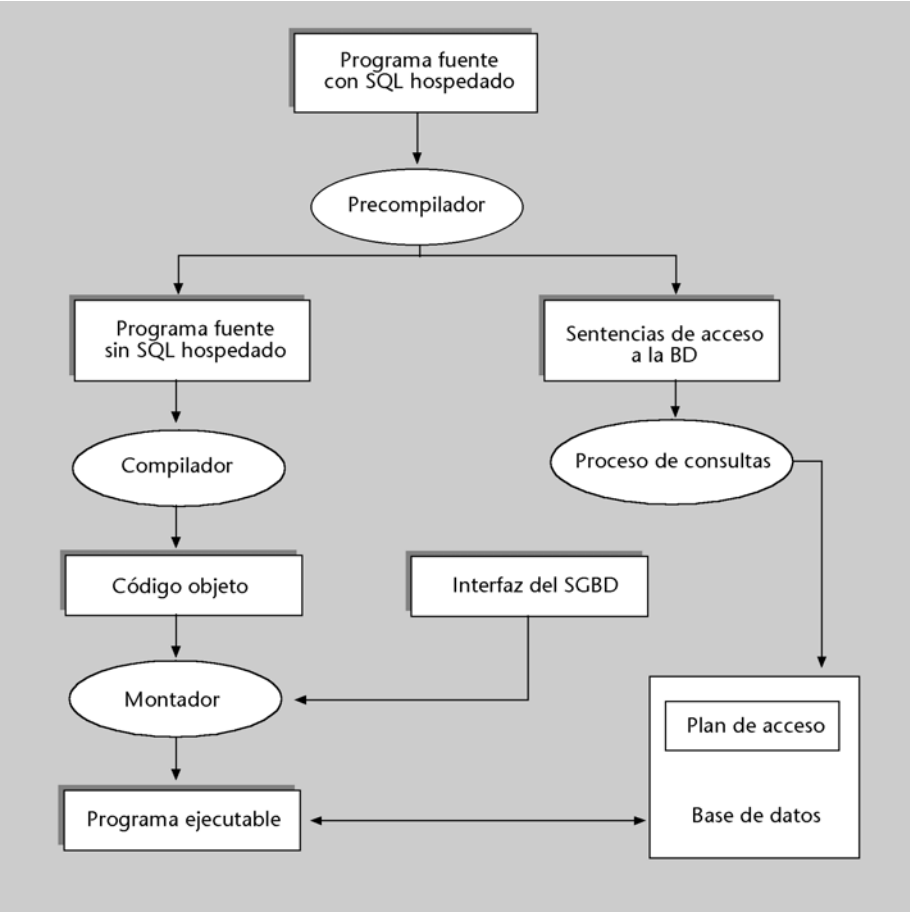
Para poder compilar la mezcla de llamadas de SQL y sentencias de programación, antes tenemos que utilizar un precompilador. Un **precompilador** es una herramienta que separa las sentencias del SQL y las sentencias de programación. Allá donde en el programa fuente haya una sentencia de acceso a la base de datos, se debe insertar una llamada a la interfaz del SGBD. El programa fuente resultante de la precompilación ya está únicamente en el lenguaje de programación, preparado para ser compilado, montado y ejecutado.

En la figura que encontraréis en la página siguiente podéis observar este funcionamiento.

Todas las sentencias de definición, manipulación y control que hemos visto para el SQL se pueden utilizar en el SQL hospedado, pero precedidas de la cláusula `EXEC SQL`. Sólo habrá una excepción: cuando el resultado de una sentencia SQL obtenga más de una fila o haga referencia también a más de una, deberemos trabajar con el **concepto de cursor**.

Un cursor se tiene que haber declarado antes de su utilización (`EXEC SQL DECLARE nombre_cursor CURSOR FOR`). Para utilizarlo, se debe abrir (`EXEC SQL OPEN nombre_cursor`), ir tomando los datos uno a uno, tratarlos

(`EXEC SQL FETCH nombre_cursor INTO`), y finalmente, cerrarlo (`EXEC SQL CLOSE nombre_cursor`).



4.2. Las SQL/CLI

Las SQL/CLI (*SQL/Call-Level Interface*), denominadas de forma abreviada CLI, permiten que aplicaciones desarrolladas en un cierto lenguaje de programación (con sólo las herramientas disponibles para este lenguaje y sin el uso de un precompilador) puedan incluir sentencias SQL mediante llamadas a librerías. Estas sentencias SQL se deben interpretar en tiempo de ejecución del programa, a diferencia del SQL hospedado, que requería el uso de un precompilador.

La **interfaz ODBC** (*Open Database Connectivity*) define una librería de funciones que permite a las aplicaciones acceder al SGBD utilizando el SQL. Las rutinas SQL/CLI están fuertemente basadas en las características de la interfaz ODBC, y gracias al trabajo desarrollado por SAG-X/Open (*SQL Access Group-X/Open*), fueron añadidas al estándar ANSI/ISO SQL92 en 1995.

Las SQL/CLI son simplemente rutinas que llaman al SGBD para interpretar las sentencias SQL que pide la aplicación. Desde el punto de vista del SGBD, las SQL/CLI se pueden considerar, simplemente, como otras aplicaciones.

Resumen

En esta unidad hemos presentado las sentencias más utilizadas del lenguaje estándar ANSI/ISO SQL92 de definición, manipulación y control de bases de datos relacionales. Como ya hemos comentado en la introducción, el SQL es un lenguaje muy potente, y esto hace que existan más sentencias y opciones de las que hemos explicado en este módulo. Sin embargo, no es menos cierto que hemos visto más sentencias que las que algunos sistemas relacionales ofrecen actualmente. Hemos intentado seguir con la mayor fidelidad el estándar, incluyendo comentarios sólo cuando en la mayoría de los sistemas relacionales comerciales alguna operación se hacía de forma distinta.

Conociendo el SQL92 podemos trabajar con cualquier sistema relacional comercial; sólo tendremos que dedicar unas cuantas horas a ver qué variaciones se dan con respecto al estándar.

Recordemos cómo será la **creación de una base de datos con SQL**:

1) En primer lugar, tendremos que dar nombre a la base de datos, con la sentencia `CREATE DATABASE`, si la hay, o con `CREATE SCHEMA`.

2) A continuación definiremos las tablas, los dominios, las aserciones y las vistas que formarán nuestra base de datos.

3) Una vez definidas las tablas, que estarán completamente vacías, se deberán llenar con la sentencia `INSERT INTO`.

Cuando la base de datos tenga un conjunto de filas, la podremos manipular, ya sea actualizando filas o bien haciéndo consultas.

Además, podemos usar todas las sentencias de control que hemos explicado.

Actividad

1. Seguro que siempre habéis querido saber dónde teníais aquella película de vídeo que nunca encontrabais. Por ello os proponemos crear una base de datos para organizar las cintas de vídeo y localizarlas rápidamente cuando os apetezca utilizarlas. Tendréis que crear la base de datos y las tablas; también deberéis decidir las claves primarias e insertar filas.
Para almacenar las cintas de vídeo, tendremos que crear las siguientes tablas:
a) Las cintas: querremos saber su código, la estantería donde se encuentran, el estante y la fila, suponiendo que en un estante haya más de una fila. Tendremos que poner nosotros el código de las cintas, con un rotulador, en el lomo de cada una.
b) Las películas: querremos saber su código, título, director principal (en el caso de que haya más de uno) y el tema. El código de las películas también lo tendremos que escribir nosotros con un rotulador para distinguir películas que tienen el mismo nombre.
c) Los actores: sólo querremos saber de ellos un código, el nombre y el apellido y, si somos aficionados al cine, otros datos que nos pueda interesar almacenar. El código de los actores, que inventaremos nosotros, nos permitirá distinguir entre actores que se llaman igual.
d) Películas que hay en cada cinta: en esta tabla pondremos el código de la cinta y el código de la película. En una cinta puede haber más de una película, y podemos tener una película repetida en más de una cinta; se debe tener en cuenta este hecho en el momento de elegir la clave primaria.
e) Actores que aparecen en las películas: en esta tabla indicaremos el código de la película y el código del actor. En una película puede participar más de un actor y un actor puede aparecer en más de una película; hay que tener presente este hecho cuando se elige la clave primaria.
Esperamos que, además de practicar sentencias de definición, manipulación y control del SQL, esta actividad os resulte muy útil.

Ejercicios de autoevaluación

Con la actividad anterior hemos practicado sentencias de definición y control del SQL. Mediante las sentencias de manipulación hemos insertado filas y, si nos hubiésemos equivocado, también habríamos borrado y modificado alguna fila. Con los ejercicios de autoevaluación practicaremos la parte de sentencias de manipulación que no hemos tratado todavía: las consultas. Los ejercicios que proponemos se harán sobre la base de datos relacional BDUOC que ha ido apareciendo a lo largo de esta unidad.

1. Obtened los códigos y los nombres y apellidos de los empleados, ordenados alfabéticamente de forma descendente por apellido y, en caso de repeticiones, por nombre.

2. Consultad el código y el nombre de los proyectos de los clientes que son de Barcelona.

3. Obtened los nombres y las ciudades de los departamentos que trabajan en los proyectos número 3 y número 4.

4. De todos los empleados que perciben un sueldo de entre 50.000 y 80.000 euros, buscad los códigos de empleado y los nombres de los proyectos que tienen asignados.

5. Buscad el nombre, la ciudad y el teléfono de los departamentos donde trabajan los empleados del proyecto GESCOM.

6. Obtened los códigos y los nombres y apellidos de los empleados que trabajan en los proyectos de precio más alto.

7. Averiguad cuál es el sueldo más alto de cada departamento. Concretamente, es necesario dar el nombre y la ciudad del departamento y el sueldo más elevado.

8. Obtened los códigos y los nombres de los clientes que tienen más de un proyecto contratado.

9. Averiguad los códigos y los nombres de los proyectos cuyos empleados asignados tienen un sueldo superior a 30.000 euros.

10. Buscad los nombres y las ciudades de los departamentos que no tienen ningún empleado asignado.

Solucionario

Ejercicios de autoevaluación

1.

```
SELECT apellido_empl, nombre_empl, codigo_empl
FROM empleados
ORDER BY apellido_empl DESC, nombre_empl DESC;
```

2. Con el SQL92 introductorio, la solución sería:

```
SELECT p.codigo_proyec, p.nombre_proyec
FROM proyectos p, clientes c
WHERE c.ciudad = 'Barcelona' and c.codigo_cli = p.codigo_cliente;
```

Con el SQL92 intermedio o con el completo, la solución sería:

```
SELECT p.codigo_proyec, p.nombre_proyec
FROM proyectos p JOIN clientes c ON c.codigo_cli = p.codigo_cliente
WHERE c.ciudad = 'Barcelona';
```

3.

```
SELECT DISTINCT e.nombre_dep, e.ciudad_dep
FROM empleados e
WHERE e.num_proyec IN (3,4);
```

4. Con el SQL92 introductorio, la solución sería:

```
SELECT e.codigo_empl, p.nombre_proyec
FROM empleados e, proyectos p
WHERE e.sueldo BETWEEN 5.0E+4 AND 8.0E+4 and e. num_proyec = p.codigo_proyec;
```

Con el SQL92 intermedio o con el completo, la solución sería:

```
SELECT e.codigo_empl, p.nombre_proyec
FROM empleados e JOIN proyectos p ON e.num_proyec = p.codigo_proyec
WHERE e.sueldo BETWEEN 5.0E+4 AND 8.0E+4;
```

5. Con el SQL92 introductorio, la solución sería:

```
SELECT DISTINCT d.*
FROM departamentos d, empleados e, proyectos p
WHERE p. nombre_proyec = 'GESCOM' and d.nombre_dep = e.nombre_dep AND
d.ciudad_dep = e.ciudad_dep and e. num_proyec = p.codigo_proyec;
```

Con el SQL92 intermedio o con el completo, la solución sería:

```
SELECT DISTINCT d.nombre_dep, d.ciudad_dep, d.telefono
FROM (departamentos d NATURAL JOIN empleados e) JOIN proyectos p ON e.num_proyec = p.codigo_proyec
WHERE p.nombre_proyec = 'GESCOM';
```

6. Con el SQL92 introductorio, la solución sería:

```
SELECT e.codigo_empl, e.nombre_empl, e.apellido_empl
FROM proyectos p, empleados e
WHERE e.num_proyec = p.codigo_proyec and p.precio = (SELECT MAX(p1. precio)
                                                     FROM proyectos p1);
```

Con el SQL92 intermedio o con el completo, la solución sería:

```
SELECT e.codigo_empl, e.nombre_empl, e.apellido_empl
FROM empleados e JOIN proyectos p ON e.num_proyec = p.codigo_proyec
WHERE p.precio = (SELECT MAX(p1.precio)
                  FROM proyectos p1);
```

7.

```
SELECT nombre_dep, ciudad_dep, MAX(sueldo) AS sueldo_maximo
FROM empleados
GROUP BY nombre_dep, ciudad_dep;
```

8. Con el SQL92 introductorio, la solución sería:

```
SELECT c.codigo_cli, c.nombre_cli
FROM proyectos p, clientes c
WHERE c.codigo_cli = p.codigo_cliente
GROUP BY c.codigo_cli, c.nombre_cli
HAVING COUNT(*) > 1;
```

Con el SQL92 intermedio o con el completo, la solución sería:

```
SELECT c.codigo_cli, c.nombre_cli
FROM proyectos p JOIN clientes c ON c.codigo_cliente = p.codigo_cliente
GROUP BY c.codigo_cli, c.nombre cli
HAVING COUNT(*) > 1;
```

9. Con el SQL92 introductorio, la solución sería:

```
SELECT p.codigo_proyec, p.nombre_proyec
FROM proyectos p, empleados e
WHERE e.num_proyec = p.codigo_proyec
GROUP BY p.codigo_proyec, p.nombre_proyec
HAVING MIN(e.sueldo) > 3.0E+4;
```

Con el SQL92 intermedio o con el completo, la solución sería:

```
SELECT p.codigo_proyec, p.nombre_proyec
FROM empleados e JOIN proyectos p ON e.num_proyec = p.codigo_proyec
GROUP BY p.codigo_proyec, p.nombre_proyec
HAVING MIN(e.sueldo)>3.0E+4;
```

10.

```
SELECT d.nombre_dep, d.ciudad_dep
FROM departamentos d
WHERE NOT EXISTS (SELECT *
                  FROM empleados e
                  WHERE e.nombre_dep = d.nombre_dep AND
                  e.ciudad_dep = d.ciudad_dep);
```

o bien:

```
SELECT nombre_dep, ciudad_dep
FROM departamentos
EXCEPT
SELECT nombre_dep, ciudad_dep
FROM empleados;
```

Bibliografía

Bibliografía básica

El SQL92 se define, según lo busquéis en ISO o en ANSI, en cualquiera de los dos documentos siguientes:

Database Language SQL (1992). Document ISO/IEC 9075:1992. International Organization for Standardization (ISO).

Database Language SQL (1992). Document ANSI/X3.135-1992. American National Standards Institute (ANSI).

Date, C.J.; Darwen, H. (1997). *A guide to the SQL Standard* (4.ª ed.). Reading, Massachusetts: Addison-Wesley.
Los libros que contienen la descripción del estándar ANSI/ISO SQL92 son bastante gruesos y pesados de leer. Este libro constituye un resumen del oficial.

Date, C.J. (2001). *Introducción a los sistemas de bases de datos* (7ª edición). Prentice Hall.
Tenéis todavía una versión más resumida de uno de los mismos autores del libro anterior en el capítulo 4 de este libro. Además en el apéndice B podéis encontrar una panorámica de SQL3.

Otros libros traducidos al castellano del SQL92 que os recomendamos son los siguientes:

Groff, J.R.; Weinberg, P.N. (1998). *LAN Times. Guía de SQL*. Osborne: McGraw-Hill.
Os recomendamos la consulta de este libro por su claridad y por los comentarios sobre el modo en el que se utiliza el estándar en los diferentes sistemas relacionales comerciales.

Silberschatz, A.; Korth, H.F.; Sudarshan, S. (1998). *Fundamentos de bases de datos*. (3.ª ed.). Madrid: McGraw-Hill.
Podéis encontrar una lectura rápida, resumida, pero bastante completa del SQL en el capítulo 4 de este libro.

Por último, para profundizar en el estudio de SQL:1999 recomendamos el siguiente libro:

Melton, J.; Simon, A.R. (2001). *SQL:1999. Undestandign Relational Language Components*. Morgan Kaufmann.

Anexos

Anexo 1

Sentencias de definición

1) Creación de esquemas:

```
CREATE SCHEMA {nombre_esquema | AUTHORIZATION usuario}
          [lista_de_elementos_del_esquema];
```

2) Borrado de esquemas:

```
DROP SCHEMA nombre_esquema {RESTRICT|CASCADE};
```

3) Creación de base de datos:

```
CREATE DATABASE nombre_base_de_datos;
```

4) Borrado de bases de datos:

```
DROP DATABASE nombre_base_de_datos;
```

5) Creación de tablas

```
CREATE TABLE nombre_tabla
             (definición_columna
             [, definición_columna...]
             [, restricciones_tabla]
             );
```

Donde tenemos lo siguiente:

- `definición_columna` es:

```
nombre_columna {tipos_datos|dominio} [def_defecto] [restric_col]
```

- Una de las restricciones de la tabla era la definición de claves foráneas:

```
FOREIGN KEY clave_foranea REFERENCES tabla [(clave_primaria)]
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
```

6) Modificación de una tabla:

```
ALTER TABLE nombre_tabla {acción_modificar_columna|
                          acción_modif_restricción_tabla};
```

Donde tenemos lo siguiente:

- `acción_modificar_columna` puede ser:

```
{ADD [COLUMN] columna def_columna|
ALTER [COLUMN] columna {SET def_defecto|DROP DEFAULT}|
DROP [COLUMN] columna {RESTRICT|CASCADE}}
```

- `acción_modif_restriccion_tabla` puede ser:

```
{ADD restricción|
DROP CONSTRAINT restricción {RESTRICT|CASCADE}}
```

7) Borrado de tablas:

```
DROP TABLE nombre_tabla {RESTRICT|CASCADE};
```

8) Creación de dominios:

```
CREATE DOMAIN nombre_dominio [AS] tipo_datos
[def_defecto] [restricciones_dominio];
```

Donde tenemos lo siguiente:

- `def_defecto` tiene el siguiente formato:

```
DEFAULT {literal|función|NULL}
```

- `restricciones_dominio` tiene el siguiente formato:

```
[CONSTRAINT nombre_restriccion] CHECK (condiciones)
```

9) Modificación de un dominio semántico:

```
ALTER DOMAIN nombre_dominio {acción_modificar_dominio|
                             acción_modif_restricción_dominio};
```

Donde tenemos lo siguiente:

- `acción_modificar_dominio` puede ser:

```
{SET def_defecto|DROP DEFAULT}
```

- `acción_modif_restricción_dominio` puede ser:

```
{ADD restricciones_dominio|DROP CONSTRAINT nombre_restricción}
```

10) Borrado de dominios creados por el usuario:

```
DROP DOMAIN nombre_dominio {RESTRICT|CASCADE};
```

11) Definición de una aserción:

```
CREATE ASSERTION nombre_aserción CHECK (condiciones);
```

12) Borrado de una aserción:

```
DROP ASSERTION nombre_aserción;
```

13) Creación de una vista:

```
CREATE VIEW nombre_vista [(lista_columnas)] AS (consulta)
[WITH CHECK OPTION];
```

14) Borrado de una vista:

```
DROP VIEW nombre_vista {RESTRICT|CASCADE};
```

Anexo 2

Sentencias de manipulación

1) Inserción de filas en una tabla:

```
INSERT INTO nombre_tabla [(columnas)]
{VALUES ({v1|DEFAULT|NULL}, ..., {{vn|DEFAULT|NULL})|<consulta>};
```

2) Borrado de filas de una tabla

```
DELETE FROM nombre_tabla
[WHERE condiciones];
```

3) Modificación de filas de una tabla:

```
UPDATE nombre_tabla
SET columna = {expresion|DEFAULT|NULL}
   [, columna = {expr|DEFAULT|NULL} ...]
WHERE condiciones;
```

4) Consultas de una base de datos relacional:

```
SELECT [DISTINCT] nombre_columnas_a_seleccionar
FROM tablas_a_consultar
[WHERE condiciones]
[GROUP BY atributos_según_los_cuales_se_quiere_agrupar]
[HAVING condiciones_por_grupos]
[ORDER BY columna_ordenación [DESC] [, columna [DESC]...]];
```

Anexo 3

Sentencias de control

1) Iniciación de transacciones:

```
SET TRANSACTION {READ ONLY|READ WRITE};
```

2) Finalización de transacciones:

```
{COMMIT|ROLLBACK} [WORK];
```

3) Autorizaciones:

```
GRANT privilegios ON objeto TO usuarios
[WITH GRANT OPTION];
```

4) Desautorizaciones:

```
REVOKE [GRANT OPTION FOR] privilegios ON objeto FROM usuarios
{RESTRICT|CASCADE};
```

Introducción al diseño de bases de datos

Dolors Costal Costa

UOC
Formación de Posgrado

Índice

Introducción al diseño de bases de datos

Introducción

En otras unidades didácticas se estudian las bases de datos relacionales y un lenguaje relacional, SQL, que nos proporciona mecanismos para crear, actualizar y consultar estas bases de datos.

Es necesario complementar estos conocimientos con un aspecto que es fundamental para poder utilizar adecuadamente la tecnología de las bases de datos relacionales: el diseño. Éste será el objeto de estudio de esta unidad, que tratará el **diseño de bases de datos** para el caso específico del modelo relacional.

Concretamente, en esta unidad explicaremos en qué consiste el diseño de una base de datos, analizaremos las etapas en las que se puede descomponer y describiremos con detalle las etapas del diseño conceptual y lógico de una base de datos relacional.

Objetivos

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

1. Conocer las etapas que integran el proceso del diseño de una base de datos.
2. Conocer las estructuras del modelo ER.
3. Saber hacer el diseño conceptual de los datos de un sistema de información mediante el modelo ER.
4. Saber hacer el diseño lógico de una base de datos relacional partiendo de un diseño conceptual expresado con el modelo ER.

1. Introducción al diseño de bases de datos

En otras unidades hemos aprendido cómo es una base de datos relacional y hemos estudiado un lenguaje, el SQL, que nos proporciona mecanismos para crear estas bases de datos, así como para actualizarlas y consultarlas.

Recordad que las bases de datos relacionales y los lenguajes SQL se han estudiado en las unidades "El modelo relacional y el álgebra relacional" y "El lenguaje SQL", respectivamente.

Sin embargo, todavía debemos resolver algunas cuestiones fundamentales para poder emplear la tecnología de las bases de datos relacionales; por ejemplo, cómo se puede decidir qué relaciones debe tener una base de datos determinada o qué atributos deben presentar las relaciones, qué claves primarias y qué claves foráneas se deben declarar, etc. La tarea de tomar este conjunto de decisiones recibe el nombre de *diseñar la base de datos*.

Una base de datos sirve para almacenar la información que se utiliza en un sistema de información determinado. Las necesidades y los requisitos de los futuros usuarios del sistema de información se deben tener en cuenta para poder tomar adecuadamente las decisiones anteriores.

En resumen, el **diseño de una base de datos** consiste en definir la estructura de los datos que debe tener la base de datos de un sistema de información determinado. En el caso relacional, esta estructura será un conjunto de esquemas de relación con sus atributos, dominios de atributos, claves primarias, claves foráneas, etc.

Si recordáis los tres mundos presentados –el real, el conceptual y el de las representaciones–, observaréis que el diseño de una base de datos consiste en la obtención de una representación informática concreta a partir del estudio del mundo real de interés.

1.1. Etapas del diseño de bases de datos

El diseño de una base de datos no es un proceso sencillo. Habitualmente, la complejidad de la información y la cantidad de requisitos de los sistemas de información hacen que sea complicado. Por este motivo, cuando se diseñan bases de datos es interesante aplicar la vieja estrategia de dividir para vencer.

Por lo tanto, conviene descomponer el proceso del diseño en varias etapas; en cada una se obtiene un resultado intermedio que sirve de punto de partida de la etapa siguiente, y en la última etapa se obtiene el resultado deseado. De este modo no hace falta resolver de golpe toda la problemática que plantea el diseño, sino que en cada etapa se afronta un solo tipo de subproblema. Así se divide el problema y, al mismo tiempo, se simplifica el proceso.

Descompondremos el diseño de bases de datos en tres etapas:

1) Etapa del diseño conceptual: en esta etapa se obtiene una estructura de la información de la futura BD independiente de la tecnología que hay que emplear. No se tiene en cuenta todavía qué tipo de base de datos se utilizará –relacional, orientada a objetos, jerárquica, etc.–; en consecuencia, tampoco se tiene en cuenta con qué SGBD ni con qué lenguaje concreto se implementará la base de datos. Así pues, la etapa del diseño conceptual nos permite concentrarnos únicamente en la problemática de la estructuración de la información, sin tener que preocuparnos al mismo tiempo de resolver cuestiones tecnológicas.

El resultado del diseño conceptual

Si retomamos la idea de los tres mundos, podemos afirmar que la etapa del diseño conceptual obtiene un resultado que se sitúa en el mundo de las concepciones, y no en el mundo de las representaciones.

El resultado de la etapa del diseño conceptual se expresa mediante algún modelo de datos de alto nivel. Uno de los más empleados es el **modelo entidad-interrelación** (*entity-relationship*), que abreviaremos con la sigla **ER**.

La forma de elaborar un diseño conceptual expresado con el modelo ER se explica en el apartado 2 de esta unidad.

2) Etapa del diseño lógico: en esta etapa se parte del resultado del diseño conceptual, que se transforma de forma que se adapte a la tecnología que se debe emplear. Más concretamente, es preciso que se ajuste al modelo del SGBD con el que se desea implementar la base de datos. Por ejemplo, si se trata de un SGBD relacional, esta etapa obtendrá un conjunto de relaciones con sus atributos, claves primarias y claves foráneas.

El resultado del diseño lógico

El resultado del diseño lógico se sitúa ya en el mundo de las representaciones.

Esta etapa parte del hecho de que ya se ha resuelto la problemática de la estructuración de la información en un ámbito conceptual, y permite concentrarnos en las cuestiones tecnológicas relacionadas con el modelo de base de datos.

Más adelante explicaremos cómo se hace el diseño lógico de una base de datos relacional, tomando como punto de partida un diseño conceptual expresado con el modelo ER; es decir, veremos cómo se puede transformar un modelo ER en un modelo relacional.

El diseño lógico de una base de datos relacional se explica en el apartado 3 de esta unidad didáctica.

3) Etapa del diseño físico: en esta etapa se transforma la estructura obtenida en la etapa del diseño lógico, con el objetivo de conseguir una mayor eficiencia; además, se completa con aspectos de implementación física que dependerán del SGBD.

El resultado del diseño físico

El resultado de la etapa del diseño físico se sitúa en el mundo de las representaciones, al igual que el resultado de la etapa del diseño lógico. La diferencia con respecto a la etapa anterior es que ahora se tienen en cuenta aspectos de carácter más físico del mundo de las representaciones.

Por ejemplo, si se trata de una base de datos relacional, la transformación de la estructura puede consistir en lo siguiente: tener almacenada alguna relación que sea la combinación de varias relaciones que se han obtenido en la etapa del diseño lógico, partir una relación en varias, añadir algún atributo calculable a una relación, etc. Los aspectos de implementación física que hay que completar consisten normalmente en la elección de estructuras físicas de implementación de las relaciones, la selección del tamaño de las memorias intermedias (*buffers*) o de las páginas, etc.

En la etapa del diseño físico –con el objetivo de conseguir un buen rendimiento de la base de datos–, se deben tener en cuenta las características de los procesos que consultan y actualizan la base de datos, como por ejemplo los caminos de acceso que utilizan y las frecuencias de ejecución. También es necesario considerar los volúmenes que se espera tener de los diferentes datos que se quieren almacenar.

2. Diseño conceptual: el modelo ER

En este apartado trataremos el diseño conceptual de una base de datos mediante el modelo ER. Lo que explicaremos es aplicable al diseño de cualquier tipo de bases de datos –relacional, jerárquica, etc.–, porque, como ya hemos dicho, en la etapa del diseño conceptual todavía no se tiene en cuenta la tecnología concreta que se utilizará para implementar la base de datos.

El **modelo ER** es uno de los enfoques de modelización de datos que más se utiliza actualmente por su simplicidad y legibilidad. Su legibilidad se ve favorecida porque proporciona una notación diagramática muy comprensiva. Es una herramienta útil tanto para ayudar al diseñador a reflejar en un modelo conceptual los requisitos del mundo real de interés como para comunicarse con el usuario final sobre el modelo conceptual obtenido y, de este modo, poder verificar si satisface sus requisitos.

El modelo ER resulta fácil de aprender y de utilizar en la mayoría de las aplicaciones. Además, existen herramientas informáticas de ayuda al diseño (herramientas CASE*) que utilizan alguna variante del modelo ER para hacer el diseño de los datos.

*** La sigla CASE corresponde al término inglés *Computer Aided Software Engineering*.**

El nombre completo del modelo ER es *entity-relationship*, y proviene del hecho de que los principales elementos que incluye son las entidades y las interrelaciones (*entities* y *relationships*). Traduciremos este nombre por 'entidad-interrelación'.

El modelo entidad-interrelación

Algunos autores denominan *entidad-relación* al modelo ER, pero en nuestro caso hemos preferido traducir *relationship* por 'interrelación' y no por 'relación', con el objetivo de evitar confusiones entre este concepto y el de *relación* que se utiliza en el modelo relacional.

El origen del modelo ER se encuentra en trabajos efectuados por Peter Chen en 1976. Posteriormente, muchos otros autores han descrito variantes y/o extensiones de este modelo. Así pues, en la literatura se encuentran muchas formas diferentes del modelo ER que pueden variar simplemente en la notación diagramática o en algunos de los conceptos en que se basan para modelizar los datos.

Cuando se quiere utilizar el modelo ER para comunicarse con el usuario, es recomendable emplear una variante del modelo que incluya sólo sus elementos más simples –entidades, atributos e interrelaciones– y, tal vez, algunas construcciones adicionales, como por ejemplo entidades débiles y dependencias de existencia. Éstos eran los elementos incluidos en el modelo original propuesto por Chen. En cambio, para llevar a cabo la tarea de modelizar propiamente dicha, suele ser útil usar un modelo ER más completo que incluya construcciones más avanzadas que extienden el modelo original.

Según la noción de *modelo de datos* que hemos utilizado en los otros módulos, un modelo de datos tiene en cuenta tres aspectos de los datos: la estructura, la manipulación y la integridad. Sin embargo, el modelo ER habitualmente se

Recordad el modelo relacional, que se ha estudiado en la unidad "El modelo relacional y el álgebra relacional".

utiliza para reflejar aspectos de la estructura de los datos y de su integridad, pero no de su manipulación.

2.1. Construcciones básicas

2.1.1. Entidades, atributos e interrelaciones

Por **entidad** entendemos un objeto del mundo real que podemos distinguir del resto de objetos y del que nos interesan algunas propiedades.

Ejemplos de entidad

Algunos ejemplos de entidad son un empleado, un producto o un despacho. También son entidades otros elementos del mundo real de interés, menos tangibles pero igualmente diferenciables del resto de objetos; por ejemplo, una asignatura impartida en una universidad, un préstamo bancario, un pedido de un cliente, etc.

Las propiedades de los objetos que nos interesan se denominan **atributos**.

Ejemplos de atributo

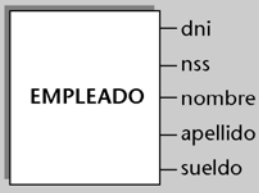
Sobre una entidad *empleado* nos puede interesar, por ejemplo, tener registrados su DNI, su NSS, su nombre, su apellido y su sueldo como atributos.

El término *entidad* se utiliza tanto para denominar objetos individuales como para hacer referencia a conjuntos de objetos similares de los que nos interesan los mismos atributos; es decir, que, por ejemplo, se utiliza para designar tanto a un empleado concreto de una empresa como al conjunto de todos los empleados de la empresa. Más concretamente, el término *entidad* se puede referir a **instancias u ocurrencias concretas** (empleados concretos) o a **tipos o clases de entidades** (el conjunto de todos los empleados).

El modelo ER proporciona una **notación diagramática** para representar gráficamente las entidades y sus atributos:

- Las **entidades** se representan con un rectángulo. El nombre de la entidad se escribe en mayúsculas dentro del rectángulo.

- Los **atributos** se representan mediante su nombre en minúsculas unido con un guión al rectángulo de la entidad a la que pertenecen. Muchas veces, dado que hay muchos atributos para cada entidad, se listan todos aparte del diagrama para no complicarlo.



Notación diagramática de entidades y atributos

La figura muestra la notación diagramática para el caso de una entidad *empleado* con los atributos *dni, nss, nombre, apellido* y *sueldo*.

Cada uno de los atributos de una entidad toma valores de un cierto dominio o conjunto de valores. Los valores de los dominios deben ser atómicos; es decir,

no deben poder ser descompuestos. Además, todos los atributos tienen que ser univaluados. Un atributo es univaluado si tiene un único valor para cada ocurrencia de una entidad.

Recordad que los valores de los atributos de las relaciones también deben ser atómicos, tal y como se ha explicado en la unidad "El modelo relacional y el álgebra relacional".

Ejemplo de atributo univaluado

El atributo *sueldo* de la entidad *empleado*, por ejemplo, toma valores del dominio de los reales y únicamente toma un valor para cada empleado concreto; por lo tanto, ningún empleado puede tener más de un valor para el sueldo.

Como ya hemos comentado anteriormente, una entidad debe ser distinguible del resto de objetos del mundo real. Esto hace que para toda entidad sea posible encontrar un conjunto de atributos que permitan identificarla. Este conjunto de atributos forma una **clave de la entidad**.

Ejemplo de clave

La entidad *empleado* tiene una clave que consta del atributo *dni* porque todos los empleados tienen números de DNI diferentes.

Una determinada entidad puede tener más de una clave; es decir, puede tener varias **claves candidatas**.

Los conceptos de *clave candidata* y *clave primaria* de una entidad son similares a los conceptos de *clave candidata* y *clave primaria* de una relación, que hemos estudiado en la unidad "El modelo relacional y el álgebra relacional".

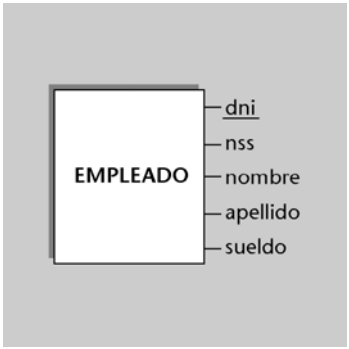
Ejemplo de clave candidata

La entidad *empleado* tiene dos claves candidatas, la que está formada por el atributo *dni* y la que está constituida por el atributo *nss*, teniendo en cuenta que el NSS también será diferente para cada uno de los empleados.

El diseñador elige una **clave primaria** entre todas las claves candidatas. En la notación diagramática, la clave primaria se subraya para distinguirla del resto de las claves.

Ejemplo de clave primaria

En el caso de la entidad *empleado*, podemos elegir *dni* como clave primaria. En la figura del margen vemos que la clave primaria se subraya para distinguirla del resto.



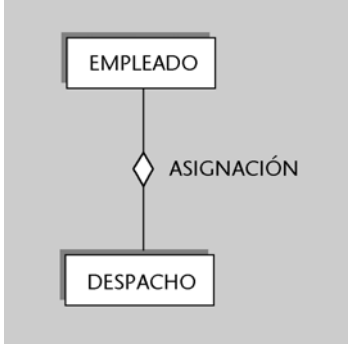
Se define **interrelación** como una asociación entre entidades.

Las interrelaciones se representan en los diagramas del modelo ER mediante un rombo. Junto al rombo se indica el nombre de la interrelación con letras mayúsculas.

Ejemplo de interrelación

Consideremos una entidad *empleado* y una entidad *despacho* y supongamos que a los empleados se les asignan despachos donde trabajar. Entonces hay una interrelación entre la entidad *empleado* y la entidad *despacho*.

Esta interrelación, que podríamos denominar *asignación*, asocia a los empleados con los despachos donde trabajan. La figura del margen muestra la interrelación *asignación* entre las entidades *empleado* y *despacho*.



El término *interrelación* se puede utilizar tanto para denominar asociaciones concretas u ocurrencias de asociaciones como para designar conjuntos o clases de asociaciones similares.

Ejemplo

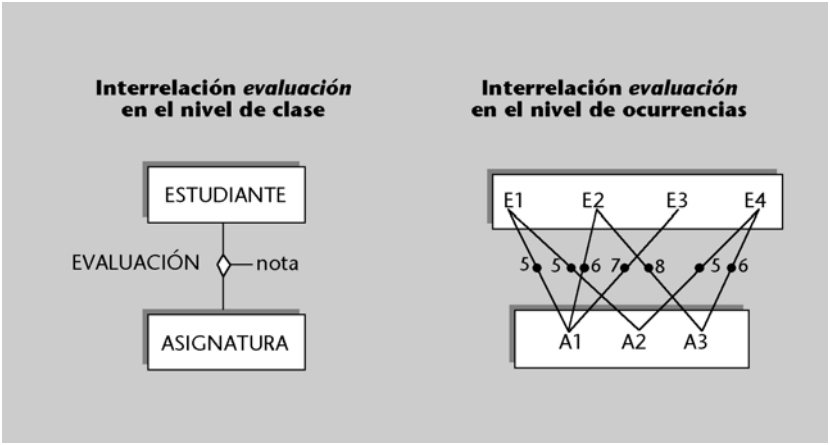
Una interrelación se aplica tanto a una asociación concreta entre el empleado de DNI '50.455.234' y el despacho 'Diagonal, 20' como a la asociación genérica entre la entidad *empleado* y la entidad *despacho*.

En ocasiones interesa reflejar algunas propiedades de las interrelaciones. Por este motivo, las interrelaciones pueden tener también atributos. Los **atributos de las interrelaciones**, igual que los de las entidades, tienen un cierto dominio, deben tomar valores atómicos y deben ser univaluados.

Los atributos de las interrelaciones se representan mediante su nombre en minúsculas unido con un guión al rombo de la interrelación a la que pertenecen.

Ejemplo de atributo de una interrelación

Observemos la entidad *estudiante* y la entidad *asignatura* que se muestran en la figura siguiente:



Entre estas dos entidades se establece la interrelación *evaluación* para indicar de qué asignaturas han sido evaluados los estudiantes. Esta interrelación tiene el atributo *nota*, que sirve para especificar qué nota han obtenido los estudiantes de las asignaturas evaluadas.

Conviene observar que el atributo *nota* deber ser forzosamente un atributo de la interrelación *evaluación*, y que no sería correcto considerarlo un atributo de la entidad *estudiante* o un atributo de la entidad *asignatura*. Lo explicaremos analizando las ocurrencias de la interrelación *evaluación* que se muestran en la figura anterior.

Si *nota* se considerase un atributo de *estudiante*, entonces para el estudiante 'E1' de la figura necesitaríamos dos valores del atributo, uno para cada asignatura que tiene el estudiante; por lo tanto, no sería univaluado. De forma similar, si *nota* fuese atributo de *asignatura* tampoco podría ser univaluado porque, por ejemplo, la asignatura 'A1' requeriría tres valores de nota, una para cada estudiante que se ha matriculado en ella. Podemos concluir que el atributo *nota* está relacionado al mismo tiempo con una asignatura y con un estudiante que la cursa y que, por ello, debe ser un atributo de la interrelación que asocia las dos entidades.

2.1.2. Grado de las interrelaciones

Una interrelación puede asociar dos o más entidades. El número de entidades que asocia una interrelación es el **grado de la interrelación**.

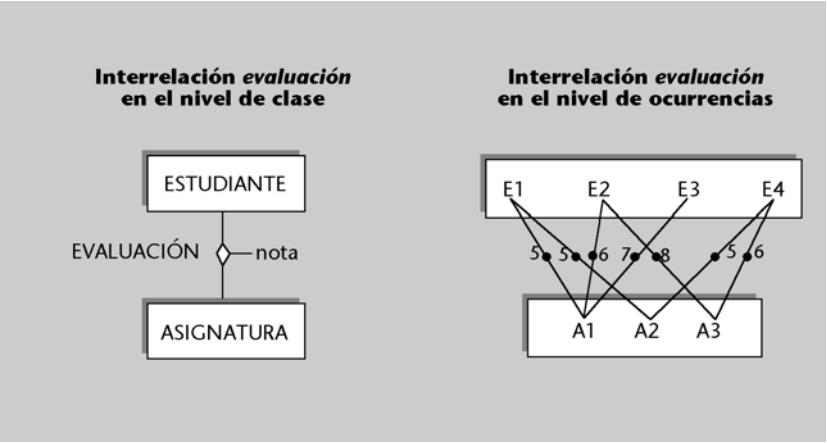
Interrelaciones de grado dos

Las interrelaciones *evaluación* y *asignación* de los ejemplos anteriores tienen grado dos:

- La interrelación *evaluación* asocia la entidad *estudiante* y la entidad *asignatura*; es decir, asocia dos entidades.
- De forma análoga, la interrelación *asignación* asocia *empleado* y *despacho*.

Las interrelaciones de grado dos se denominan también **interrelaciones binarias**. Todas las interrelaciones de grado mayor que dos se denominan, en conjunto, **interrelaciones *n*-arias**. Así pues, una interrelación *n*-aria puede tener grado tres y ser una interrelación ternaria, puede tener grado cuatro y ser una interrelación cuaternaria, etc.

A continuación presentaremos un ejemplo que nos ilustrará el hecho de que, en ocasiones, las interrelaciones binarias no nos permiten modelizar correctamente la realidad y es necesario utilizar interrelaciones de mayor grado.

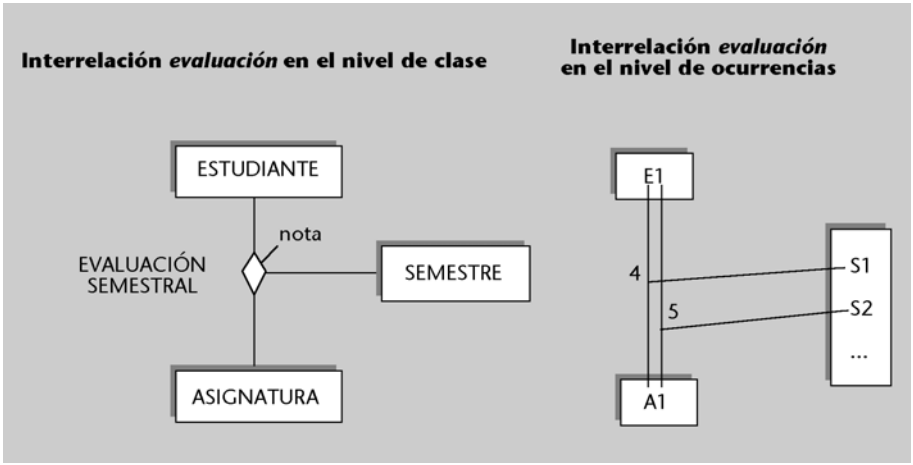


Consideremos la interrelación *evaluación* de la figura anterior, que tiene un atributo *nota*. Este atributo permite registrar la nota obtenida por cada estudiante en cada asignatura de la que ha sido evaluado. Una interrelación permite establecer una sola asociación entre unas entidades individuales determinadas. En otras palabras, sólo se puede interrelacionar una vez al estudiante 'E1' con la asignatura 'A1' vía la interrelación evaluación.

Observad que, si pudiese haber más de una interrelación entre el estudiante 'E1' y la asignatura 'A1', no podríamos distinguir estas diferentes ocurrencias de la interrelación. Esta restricción hace que se registre una sola nota por estudiante y asignatura.

Supongamos que deseamos registrar varias notas por cada asignatura y estudiante correspondientes a varios semestres en los que un mismo estudiante ha cursado una asignatura determinada (desgraciadamente, algunos estudiantes tienen que cursar una asignatura varias veces antes de aprobarla). La interrelación

anterior no nos permitiría reflejar este caso. Sería necesario aumentar el grado de la interrelación, tal y como se muestra en la figura siguiente:



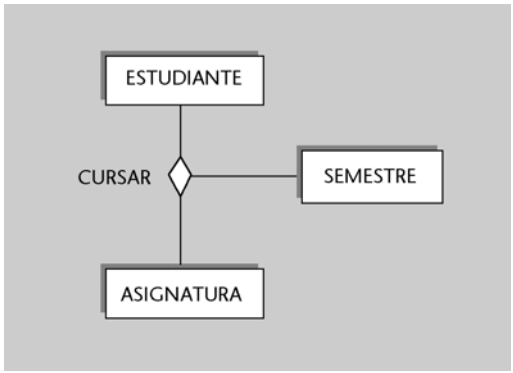
La interrelación ternaria *evaluación-semestral* asocia estudiantes, asignaturas y una tercera entidad que denominamos *semestre*. Su atributo *nota* nos permite reflejar todas las notas de una asignatura que tiene un estudiante correspondientes a diferentes semestres.

De hecho, lo que sucede en este caso es que, según los requisitos de los usuarios de esta BD, una nota pertenece al mismo tiempo a un estudiante, a una asignatura y a un semestre y, lógicamente, debe ser un atributo de una interrelación ternaria entre estas tres entidades.

Este ejemplo demuestra que una interrelación binaria puede no ser suficiente para satisfacer los requisitos de los usuarios, y puede ser necesario aplicar una interrelación de mayor grado. Conviene observar que esto también puede ocurrir en interrelaciones que no tienen atributos.

Ejemplo de interrelación ternaria sin atributos

Consideremos un caso en el que deseamos saber para cada estudiante qué asignaturas ha cursado cada semestre, a pesar de que no queremos registrar la nota que ha obtenido. Entonces aplicaríamos también una interrelación ternaria entre las entidades *estudiante*, *asignatura* y *semestre* que no tendría atributos, tal y como se muestra en la figura siguiente:



Hemos analizado casos en los que era necesario utilizar interrelaciones ternarias para poder modelizar correctamente ciertas situaciones de interés del mundo

real. Es preciso remarcar que, de forma similar, a veces puede ser necesario utilizar interrelaciones de grado todavía mayor: cuaternarias, etc.

En el subapartado siguiente analizaremos con detalle las interrelaciones binarias, y más adelante, las interrelaciones *n*-arias.

Las relaciones *n*-arias se analizan en el subapartado 2.1.4 de esta unidad didáctica.

2.1.3. Interrelaciones binarias

Conectividad de las interrelaciones binarias

La **conectividad de una interrelación** expresa el tipo de correspondencia que se establece entre las ocurrencias de entidades asociadas con la interrelación. En el caso de las interrelaciones binarias, expresa el número de ocurrencias de una de las entidades con las que una ocurrencia de la otra entidad puede estar asociada según la interrelación.

Una interrelación binaria entre dos entidades puede tener tres tipos de conectividad:

- **Conectividad uno a uno (1:1).** La conectividad 1:1 se denota poniendo un 1 a lado y lado de la interrelación.

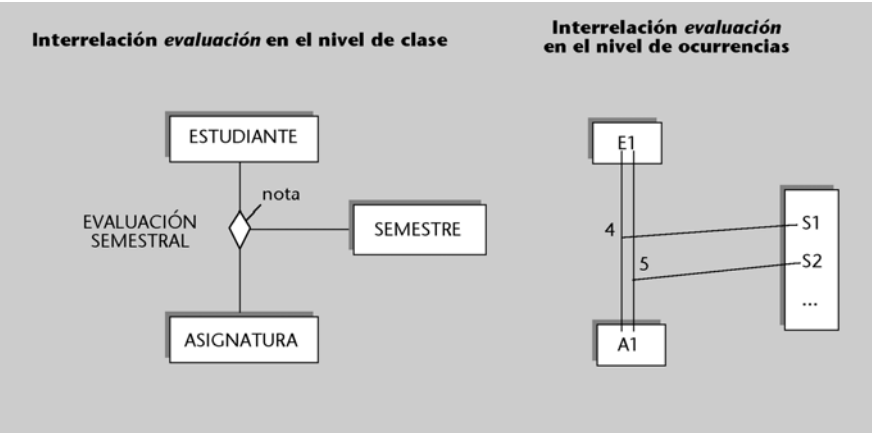
- **Conectividad uno a muchos (1:N).** La conectividad 1:N se denota poniendo un 1 en un lado de la interrelación y una N en el otro.

- **Conectividad muchos a muchos: (M:N).** La conectividad M:N se denota poniendo una M en uno de los lados de la interrelación, y una N en el otro.

Ejemplos de conectividad en una interrelación binaria

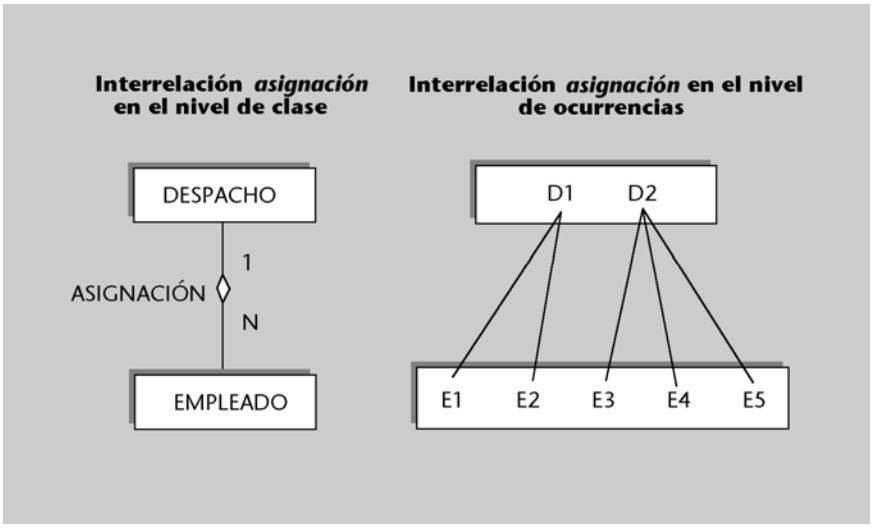
A continuación analizaremos un ejemplo de cada una de las conectividades posibles para una interrelación binaria:

a) Conectividad 1:1



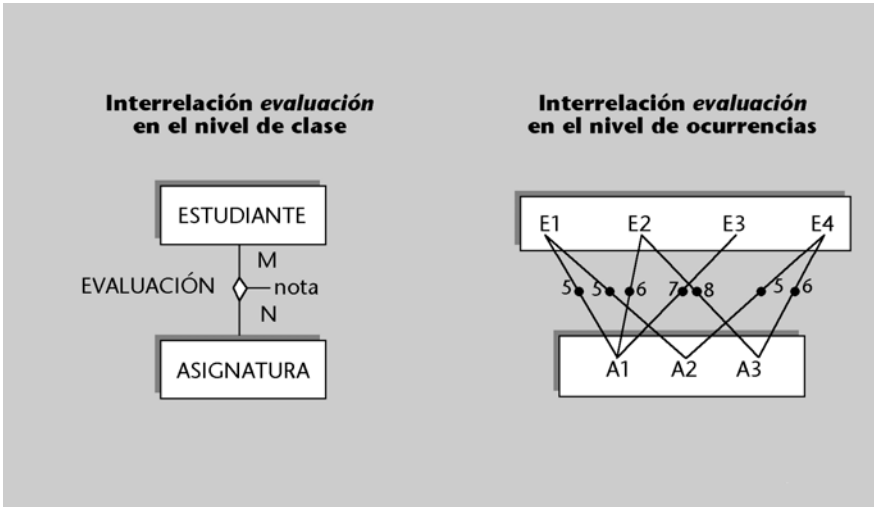
La interrelación anterior tiene conectividad 1:1. Esta interrelación asocia las delegaciones de una empresa con las ciudades donde están situadas. El hecho de que sea 1:1 indica que una ciudad tiene sólo una delegación, y que una delegación está situada en una única ciudad.

b) Conectividad 1:N



La interrelación *asignación* entre la entidad *empleado* y la entidad *despacho* tiene conectividad 1:N, y la N está en el lado de la entidad *empleado*. Esto significa que un empleado tiene un solo despacho asignado, pero que, en cambio, un despacho puede tener uno o más empleados asignados.

c) Conectividad M:N



Para analizar la conectividad M:N, consideramos la interrelación *evaluación* de la figura anterior. Nos indica que un estudiante puede ser evaluado de varias asignaturas y, al mismo tiempo, que una asignatura puede tener varios estudiantes por evaluar.

Es muy habitual que las interrelaciones binarias M:N y todas las *n*-arias tengan atributos. En cambio, las interrelaciones binarias 1:1 y 1:N no tienen por qué tenerlos. Siempre se pueden asignar estos atributos a la entidad del lado N, en el caso de las 1:N, y a cualquiera de las dos entidades interrelacionadas en el caso de las 1:1. Este cambio de situación del atributo se puede hacer porque no origina un atributo multivaluado.

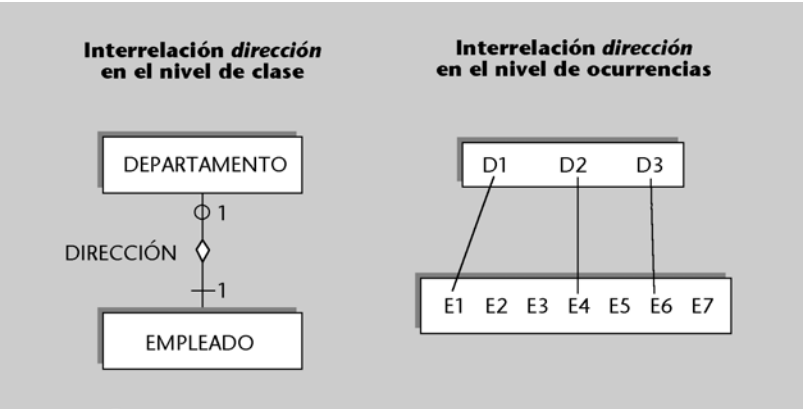
Dependencias de existencia en las interrelaciones binarias

En algunos casos, una entidad individual sólo puede existir si hay como mínimo otra entidad individual asociada con ella mediante una interrelación binaria determinada. En estos casos, se dice que esta última entidad es una **entidad obligatoria en la interrelación**. Cuando esto no sucede, se dice que es una **entidad opcional en la interrelación**.

En el modelo ER, un círculo en la línea de conexión entre una entidad y una interrelación indica que la entidad es opcional en la interrelación. La obligatoriedad de una entidad a una interrelación se indica con una línea perpendicular. Si no se consigna ni un círculo ni una línea perpendicular, se considera que la dependencia de existencia es desconocida.

Ejemplo de dependencias de existencia

La figura siguiente nos servirá para entender el significado práctico de la dependencia de existencia. La entidad *empleado* es obligatoria en la interrelación *dirección*. Esto indica que no puede existir un departamento que no tenga un empleado que actúa de director del departamento. La entidad *departamento*, en cambio, es opcional en la interrelación *dirección*. Es posible que haya un empleado que no está interrelacionado con ningún departamento: puede haber –y es el caso más frecuente– empleados que no son directores de departamento.



Aplicaremos la dependencia de existencia en las interrelaciones binarias, pero no en las *n*-arias.

2.1.4. Ejemplo: base de datos de casas de colonias

En este punto, y antes de continuar explicando construcciones más complejas del modelo ER, puede resultar muy ilustrativo ver la aplicación práctica de las construcciones que hemos estudiado hasta ahora. Por este motivo, analizaremos un caso práctico de diseño con el modelo ER que corresponde a una base de datos destinada a la gestión de las inscripciones en un conjunto de casas de colonias. El modelo ER de esta base de datos será bastante sencillo e incluirá sólo entidades, atributos e interrelaciones binarias (no incluirá interrelaciones *n*-arias ni otros tipos de estructuras).

La descripción siguiente explica con detalle los requisitos de los usuarios que hay que tener en cuenta al hacer el diseño conceptual de la futura base de datos:

a) Cada casa de colonias tiene un nombre que la identifica. Se desea saber de cada una, aparte del nombre, la capacidad (el número de niños que se pueden alojar en cada una como máximo), la comarca donde está situada y las ofertas de actividades que proporciona. Una casa puede ofrecer actividades como por ejemplo natación, esquí, remo, pintura, fotografía, música, etc.

b) Es necesario tener en cuenta que en una casa de colonias se pueden practicar varias actividades (de hecho, cada casa debe ofrecer como mínimo una), y también puede ocurrir que una misma actividad se pueda llevar a cabo en varias casas. Sin embargo, toda actividad que se registre en la base de datos debe ser ofertada como mínimo en una de las casas.

c) Interesa tener una evaluación de las ofertas de actividades que proporcionan las casas. Se asigna una calificación numérica que indica el nivel de calidad que tiene cada una de las actividades ofertadas.

Es posible, ...

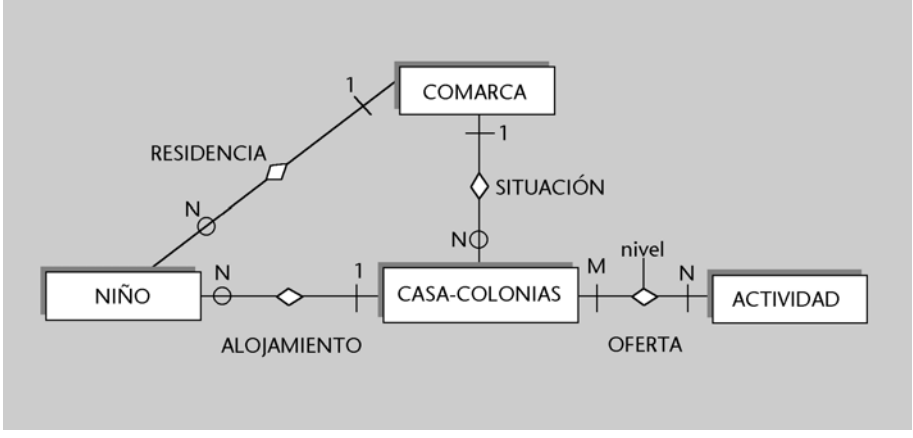
... por ejemplo, que una actividad como por ejemplo el esquí tenga una calificación de 10 en la oferta de la casa Grévol, y que la misma actividad tenga una calificación de 8 en la casa Ardilla.

d) Las casas de colonias alojan niños que se han inscrito para pasar en ellas unas pequeñas vacaciones. Se quiere tener constancia de los niños que se alojan en cada una de las casas en el momento actual. Se debe suponer que hay casas que están vacías (en las que no se aloja ningún niño) durante algunas temporadas.

e) De los niños que se alojan actualmente en alguna de las casas, interesa conocer un código que se les asigna para identificarlos, su nombre, su apellido, el número de teléfono de sus padres y su comarca de residencia.

f) De las comarcas donde hay casas o bien donde residen niños, se quiere tener registrados la superficie y el número de habitantes. Se debe considerar que puede haber comarcas donde no reside ninguno de los niños que se alojan en un momento determinado en las casas de colonias, y comarcas que no disponen de ninguna casa.

La figura siguiente muestra un diagrama ER que satisface los requisitos anteriores. Los atributos de las entidades no figuran en el diagrama y se listan aparte.



Los atributos de las entidades que figuran en el diagrama son los siguientes (las claves primarias están subrayadas):

CASA-COLONIAS
nombre-casa, capacidad
ACTIVIDAD
nombre-actividad
NIÑO
código-niño, nombre, apellido, teléfono
COMARCA
nombre-comarca, superficie, número-habitantes

A continuación comentamos los aspectos más relevantes de este modelo ER:

1) Una de las dificultades que en ocasiones se presenta durante la modelización conceptual es decidir si una información determinada debe ser una entidad o un atributo. En nuestro ejemplo, puede resultar difícil decidir si *comarca* se debe modelizar como una entidad o como un atributo.

A primera vista, podría parecer que *comarca* debe ser un atributo de la entidad *casa-colonias* para indicar dónde está situada una casa de colonias, y también un atributo de la entidad *niño* para indicar la residencia del niño. Sin embargo, esta solución no sería adecuada, porque se quieren tener informaciones adicionales asociadas a la comarca: la superficie y el número de habitantes. Es preciso que *comarca* sea una entidad para poder reflejar estas informaciones adicionales como atributos de la entidad.

La entidad *comarca* tendrá que estar, evidentemente, interrelacionada con las entidades *niño* y *casa-colonias*. Observad que de este modo, además, se hace patente que las comarcas de residencia de los niños y las comarcas de situación de las casas son informaciones de un mismo tipo.

2) Otra decisión que hay que tomar es si el concepto *actividad* se debe modelizar como una entidad o como un atributo. *Actividad* no tiene informaciones adicionales asociadas; no tiene, por lo tanto, más atributos que los que forman la clave. Aun así, es necesario que *actividad* sea una entidad para que, mediante la interrelación *oferta*, se pueda indicar que una casa de colonias ofrece actividades.

Observad que las actividades ofertadas no se pueden expresar como un atributo de *casa-colonias*, porque una casa puede ofrecer muchas actividades y, en este caso, el atributo no podría tomar un valor único.

3) Otra elección difícil, que con frecuencia se presenta al diseñar un modelo ER, consiste en modelizar una información determinada como una entidad o

como una interrelación. Por ejemplo, podríamos haber establecido que *oferta*, en lugar de ser una interrelación, fuese una entidad; lo habríamos hecho así:

La entidad *oferta* representada en la figura anterior tiene los atributos que presentamos a continuación:

OFERTA
nombre-casa, nombre-actividad, nivel

Esta solución no acaba de reflejar adecuadamente la realidad. Si analizamos la clave de *oferta*, podemos ver que se identifica con *nombre-casa*, que es la clave de la entidad *casa-colonias*, y con *nombre-actividad*, que es la clave de la entidad *actividad*. Esto nos debe hacer sospechar que oferta, de hecho, corresponde a una asociación o interrelación entre *casas* y *actividades*. En consecuencia, reflejaremos la realidad con más exactitud si modelizamos *oferta* como una interrelación entre estas entidades.

4) Finalmente, un aspecto que hay que cuidar durante el diseño conceptual es el de evitar las redundancias. Por ejemplo, si hubiésemos interrelacionado *comarca* con *actividad* para saber qué actividades se realizan en las casas de cada una de las comarcas, habríamos tenido información redundante. La interrelación *oferta* junto con la interrelación *situación* ya permiten saber, de forma indirecta, qué actividades se hacen en las comarcas.

2.1.5. Interrelaciones *n*-arias

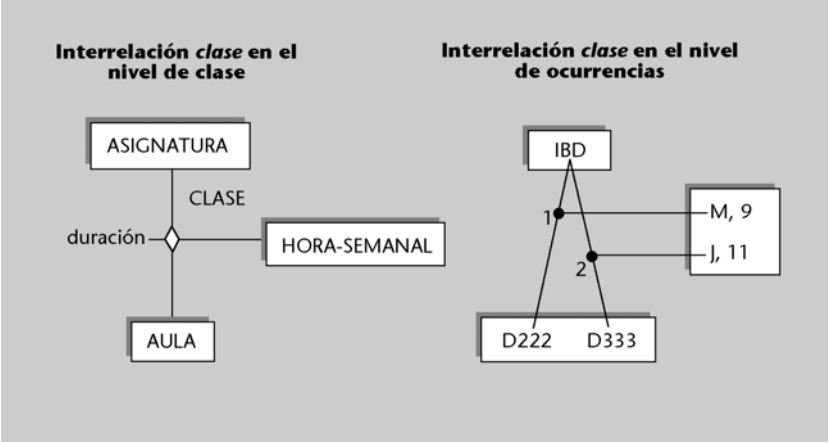
Las interrelaciones *n*-arias, igual que las binarias, pueden tener diferentes tipos de conectividad. En este subapartado analizaremos primero el caso particular de las interrelaciones ternarias y, a continuación, trataremos las conectividades de las interrelaciones *n*-arias en general.

Conectividad de las interrelaciones ternarias

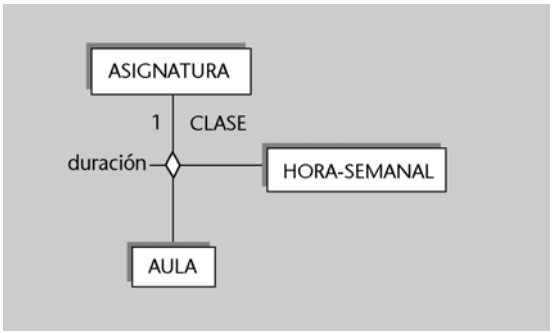
Cada una de las tres entidades asociadas con una interrelación ternaria puede estar conectada con conectividad "uno" o bien con conectividad "muchos". En consecuencia, las interrelaciones ternarias pueden tener cuatro tipos de conectividad: **M:N:P**, **M:M:1**, **N:1:1** y **1:1:1**.

Observad que usamos M, N y P para representar "muchos", y 1 para representar "uno".

Analizaremos cómo se decide cuál es la conectividad adecuada de una interrelación ternaria mediante el siguiente ejemplo. Consideremos una interrelación que denominamos *clase* y que asocia las entidades *asignatura*, *aula* y *hora-semanal*. Esta interrelación permite registrar clases presenciales. Una clase corresponde a una asignatura determinada, se imparte en un aula determinada y a una hora de la semana determinada. Por ejemplo, podemos registrar que se hace clase de la asignatura IBD en el aula D222 el martes a las 9, tal y como se muestra en la figura de la página siguiente. El atributo *duración* nos permite saber cuántas horas dura la clase.



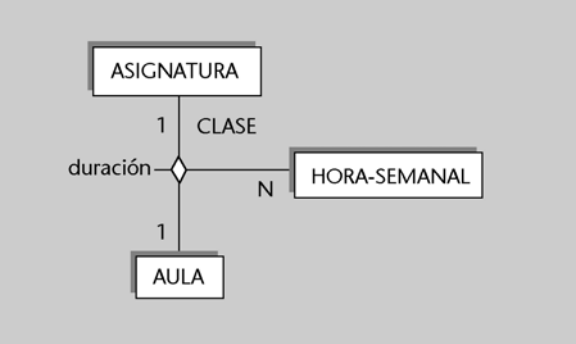
Para decidir si el lado de la entidad *asignatura* se conecta con "uno" o con "muchos" , es necesario preguntarse si, dadas un *aula* y una *hora-semanal*, se puede hacer clase de sólo una o bien de muchas asignaturas en aquellas aula y hora. La respuesta sería que sólo se puede hacer clase de una asignatura en una misma aula y hora. Esto nos indica que *asignatura* se conecta con "uno", tal y como reflejamos en la figura siguiente:



Como nos indica este ejemplo, para decidir cómo se debe conectar una de las entidades, es necesario preguntarse si, ya fijadas ocurrencias concretas de las otras dos, es posible conectar sólo "una" o bien "muchas" ocurrencias de la primera entidad.

Utilizaremos el mismo procedimiento para determinar cómo se conectan las otras dos entidades del ejemplo. Una vez fijadas una asignatura y un aula, es posible que se haga clase de aquella asignatura en aquella aula, en varias horas de la semana; entonces, *hora-semana* se conecta con "muchos". Finalmente, la

entidad *aula* se conecta con "uno", teniendo en cuenta que, fijadas una asignatura y una hora de la semana, sólo se puede hacer una clase de aquella asignatura a aquella hora. La conectividad resultante, de este modo, es N:1:1.



Caso general: conectividad de las interrelaciones *n*-arias

Lo que hemos explicado sobre la conectividad para las interrelaciones ternarias es fácilmente generalizable a interrelaciones *n*-arias.

Una interrelación *n*-aria puede tener $n + 1$ tipos de conectividad, teniendo en cuenta que cada una de las *n* entidades puede estar conectada con "uno" o con "muchos" en la interrelación*.

*** Recordad que para las interrelaciones ternarias hay cuatro tipos posibles de conectividad.**

Para decidir si una de las entidades se conecta con "uno" o con "muchos", es necesario preguntarse si, fijadas ocurrencias concretas de las otras $n - 1$ entidades, es posible conectar sólo una o bien muchas ocurrencias de la primera entidad:

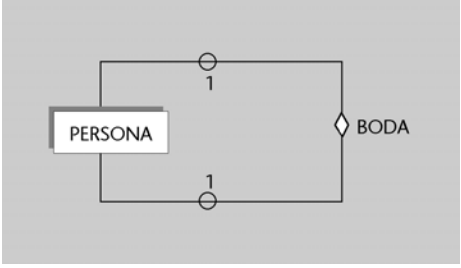
- Si la respuesta es que sólo una, entonces se conecta con "uno".
- Si la respuesta es que muchas, la entidad se conecta con "muchos".

2.1.6. Interrelaciones recursivas

Una **interrelación recursiva** es una interrelación en la que alguna entidad está asociada más de una vez.

Ejemplo de interrelación recursiva

Si, para una entidad *persona*, queremos tener constancia de qué personas están actualmente casadas entre ellas, será necesario definir la siguiente interrelación, que asocia dos veces la entidad *persona*:



Una interrelación recursiva puede ser tanto binaria como *n*-aria:

1) Interrelación recursiva binaria: interrelación en la que las ocurrencias asocian dos instancias de la misma entidad*. Las interrelaciones binarias recursivas pueden tener conectividad 1:1, 1:N o M:N, como todas las binarias. En esta interrelación también es posible expresar la dependencia de existencia igual que en el resto de las interrelaciones binarias.

*** Éste es el caso de la interrelación *boda* anterior.**

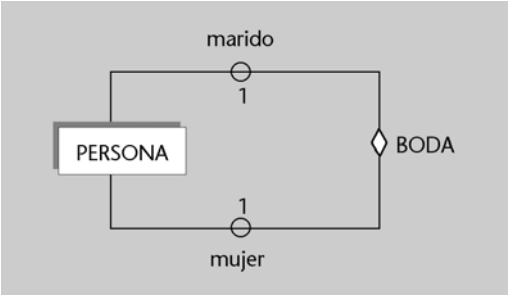
Ejemplo de interrelación recursiva binaria

La interrelación *boda* tiene conectividad 1:1 porque un marido está casado con una sola mujer y una mujer está casada con un solo marido. También tiene un círculo en los dos lados (según la dependencia de existencia), porque puede haber personas que no estén casadas.

En una interrelación recursiva, puede interesar distinguir los diferentes papeles que una misma entidad tiene en la interrelación. Con este objetivo, se puede etiquetar cada línea de la interrelación con un rol. En las interrelaciones no recursivas normalmente no se especifica el rol; puesto que todas las entidades interrelacionadas son de clases diferentes, sus diferencias de rol se sobreentienden.

Roles diferentes

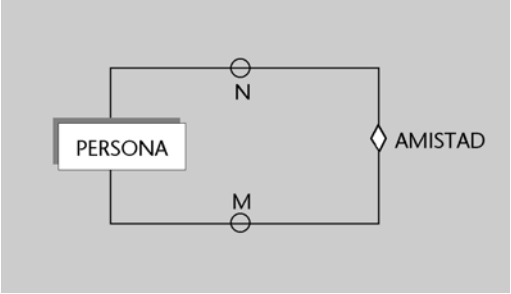
Una ocurrencia de la interrelación *boda* asocia a dos personas concretas. Para reflejar el papel diferente que tiene cada una de ellas en la interrelación, una de las personas tendrá el rol de marido y la otra tendrá el rol de mujer.



Algunas interrelaciones recursivas no presentan diferenciación de roles; entonces, las líneas de la interrelación no se etiquetan.

No-diferencia de roles

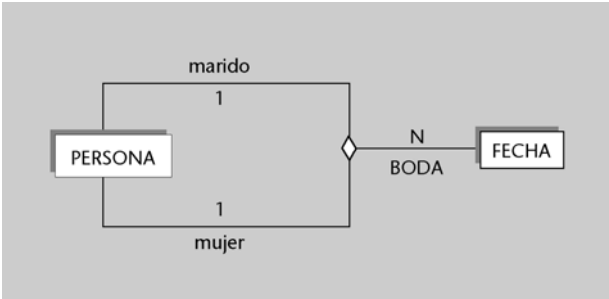
Consideremos una interrelación *amistad* que asocia a personas concretas que son amigas. A diferencia de lo que sucedía en la interrelación *boda*, donde una de las personas es el marido y la otra la mujer, en este caso no hay diferenciación de roles entre las dos personas interrelacionadas. A continuación se muestra esta interrelación. Observad que su conectividad es M:N, teniendo en cuenta que una persona puede tener muchos amigos y, al mismo tiempo, puede haber muchas personas que la consideran amiga.



2) Interrelación recursiva *n*-aria: interrelación recursiva en la que las ocurrencias asocian más de dos instancias.

Ejemplo de interrelación recursiva ternaria

Consideremos una interrelación que registra todas las bodas que se han producido a lo largo del tiempo entre un conjunto de personas determinado. Esta interrelación permite tener constancia no sólo de las bodas vigentes, sino de todas las bodas realizadas en un cierto periodo de tiempo.



Esta interrelación es recursiva y ternaria. Una ocurrencia de la interrelación asocia a una persona que es el marido, a otra que es la mujer y la fecha de su boda. La conectividad es N:1:1. A los lados del marido y de la mujer les corresponde un 1, porque un marido o una mujer, en una fecha determinada, se casa con una sola persona. Al lado de la entidad fecha le corresponde una N, porque se podría dar el caso de que hubiese, en fechas diferentes, más de una boda entre las mismas personas.

2.1.7. Entidades débiles

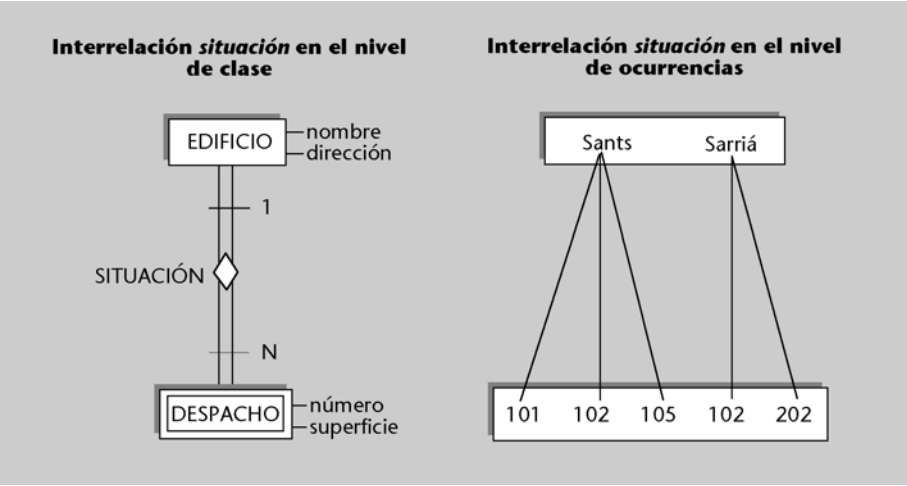
Las entidades que hemos considerado hasta ahora tienen un conjunto de atributos que forman su claves primarias y que permiten identificarlas completamente. Estas entidades se denominan, de forma más específica, **entidades fuertes**. En este subapartado consideraremos otro tipo de entidades que denominaremos *entidades débiles*.

Una **entidad débil** es una entidad cuyos atributos no la identifican completamente, sino que sólo la identifican de forma parcial. Esta entidad debe participar en una interrelación que ayuda a identificarla.

Una entidad débil se representa con un rectángulo doble, y la interrelación que ayuda a identificarla se representa con una doble línea.

Ejemplo de entidad débil

Consideremos las entidades *edificio* y *despacho* de la figura siguiente. Supongamos que puede haber despachos con el mismo número en edificios diferentes. Entonces, su número no identifica completamente un despacho. Para identificar completamente un despacho, es necesario tener en cuenta en qué edificio está situado. De hecho, podemos identificar un despacho mediante la interrelación *situación*, que lo asocia a un único edificio. El nombre del edificio donde está situado junto con el número de despacho lo identifican completamente.



En el ejemplo anterior, la interrelación *situación* nos ha permitido completar la identificación de los despachos. Para toda entidad débil, siempre debe haber una única interrelación que permita completar su identificación. Esta interrelación debe ser binaria con conectividad 1:N, y la entidad débil debe estar en el lado N. De este modo, una ocurrencia de la entidad débil está asociada con una sola ocurrencia de la entidad del lado 1, y será posible completar su identificación de forma única. Además, la entidad del lado 1 debe ser obligatoria en la interrelación porque, si no fuese así, alguna ocurrencia de la entidad débil podría no estar interrelacionada con ninguna de sus ocurrencias y no se podría identificar completamente.

2.2. Extensiones del modelo ER

En este subapartado estudiaremos algunas construcciones avanzadas que extienden el modelo ER estudiado hasta ahora.

2.2.1. Generalización/especialización

En algunos casos, hay ocurrencias de una entidad que tienen características propias específicas que nos interesa modelizar. Por ejemplo, puede ocurrir que se quiera tener constancia de qué coche de la empresa tienen asignado los empleados que son directivos; también que, de los empleados técnicos, interese tener una interrelación con una entidad *proyecto* que indique en qué proyectos trabajan y se desee registrar su titulación. Finalmente, que convenga conocer la antigüedad de los empleados administrativos. Asímismo, habrá algunas características comunes a todos los empleados: todos se identifican por un DNI, tienen un nombre, un apellido, una dirección y un número de teléfono.

La **generalización/especialización** permite reflejar el hecho de que hay una entidad general, que denominamos *entidad superclase*, que se puede especializar en entidades subclase:

a) La **entidad superclase** nos permite modelizar las características comunes de la entidad vista de una forma genérica.

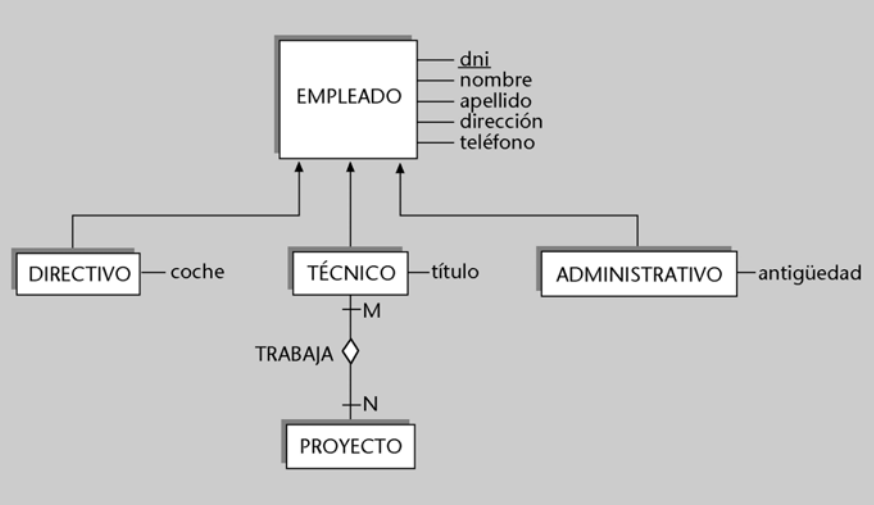
b) Las **entidades subclase** nos permiten modelizar las características propias de sus especializaciones.

Es necesario que se cumpla que toda ocurrencia de una entidad subclase sea también una ocurrencia de su entidad superclase.

Denotamos la generalización/especialización con una flecha que parte de las entidades subclase y que se dirige a la entidad superclase.

Ejemplo de entidades superclase y subclase

En la figura siguiente están representadas la entidad superclase, que corresponde al empleado del ejemplo anterior, y las entidades subclase, que corresponden al directivo, al técnico y al administrativo del mismo ejemplo.



En la generalización/especialización, las características (atributos o interrelaciones) de la entidad superclase se propagan hacia las entidades subclase. Es lo que se denomina **herencia de propiedades**.

En el diseño de una generalización/especialización, se puede seguir uno de los dos procesos siguientes:

1) Puede ocurrir que el diseñador primero identifique la necesidad de la entidad superclase y, posteriormente, reconozca las características específicas que hacen necesarias las entidades subclase. En estos casos se dice que ha seguido un **proceso de especialización**.

2) La alternativa es que el diseñador modelice en primer lugar las entidades subclase y, después, se dé cuenta de sus características comunes e identifique la entidad superclase. Entonces se dice que ha seguido un **proceso de generalización**.

La generalización/especialización puede ser de dos tipos:

a) Disjunta. En este caso no puede suceder que una misma ocurrencia aparezca en dos entidades subclase diferentes. Se denota gráficamente con la etiqueta D.

b) Solapada. En este caso no tiene lugar la restricción anterior. Se denota gráficamente con la etiqueta S.

Nuestro ejemplo de los empleados...

... corresponde a una generalización/especialización disjunta porque ningún empleado puede ser de más de un tipo. Se denota con la etiqueta D.

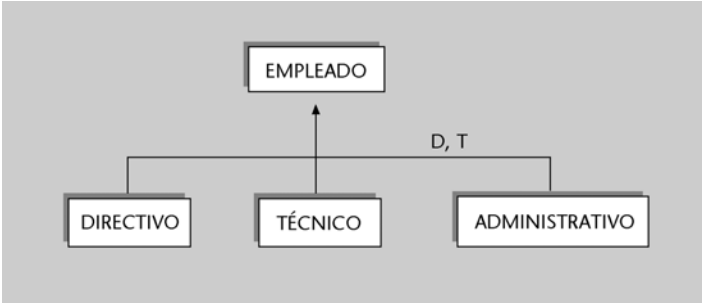
Además, una generalización/especialización también puede ser:

1) Total. En este caso, toda ocurrencia de la entidad superclase debe pertenecer a alguna de las entidades subclase. Esto se denota con la etiqueta T.

2) Parcial. En este caso no es necesario que se cumpla la condición anterior. Se denota con la etiqueta P.

La generalización/especialización de los empleados

La generalización/especialización de los empleados es total porque suponemos que todo empleado debe ser directivo, técnico o administrativo. Se denota con la etiqueta T.



2.2.2. Entidades asociativas

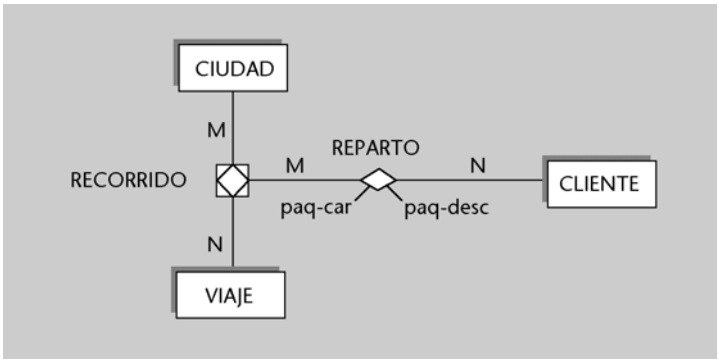
En este subapartado veremos un mecanismo que nos permite considerar una interrelación entre entidades como si fuese una entidad.

La entidad que resulta de considerar una interrelación entre entidades como si fuese una entidad es una **entidad asociativa**, y tendrá el mismo nombre que la interrelación sobre la que se define.

La utilidad de una entidad asociativa consiste en que se puede interrelacionar con otras entidades y, de forma indirecta, nos permite tener interrelaciones en las que intervienen interrelaciones. Una entidad asociativa se denota recuadrando el rombo de la interrelación de la que proviene.

Ejemplo de entidad asociativa

La figura siguiente muestra un ejemplo de entidad asociativa:

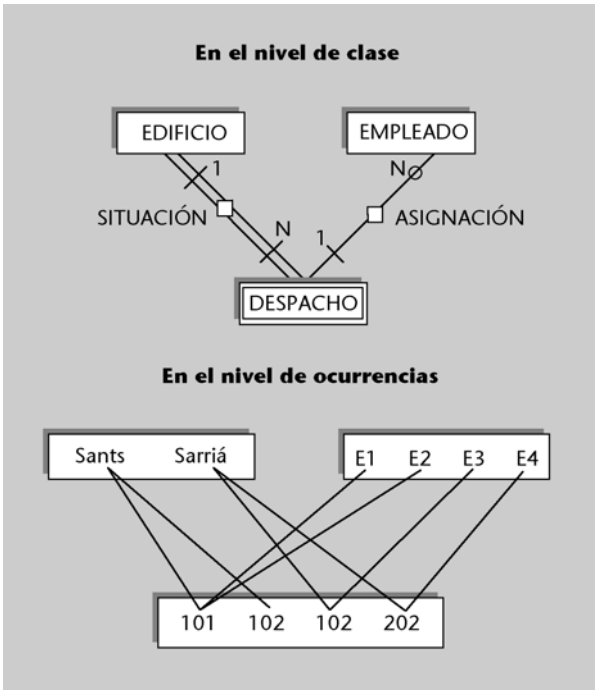


Recorrido es una interrelación de conectividad M:N que registra las ciudades por donde han pasado los diferentes viajes organizados por una empresa de reparto de paquetes. Consideramos *recorrido* una entidad asociativa con el fin de interrelacionarla con la entidad *cliente*; de este modo nos será posible reflejar por orden de qué clientes se han hecho repartos en una ciudad del recorrido de un viaje, así como el número de paquetes cargados y descargados siguiendo sus indicaciones.

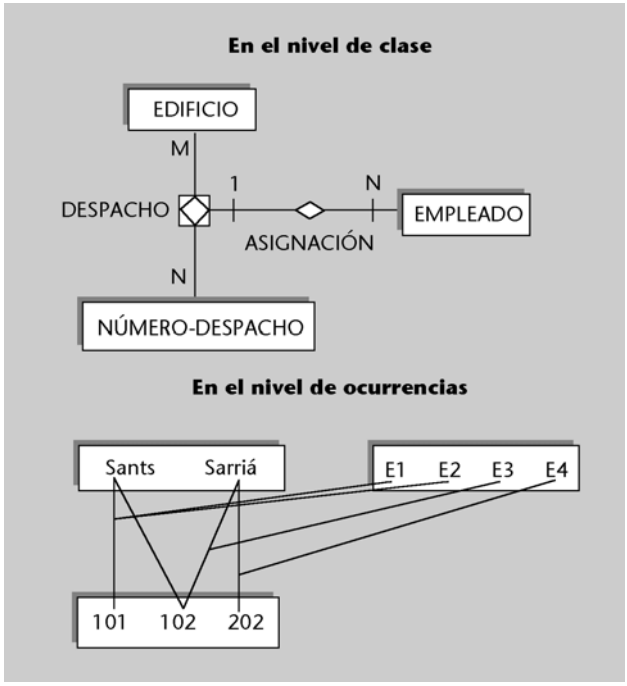
El mecanismo de las entidades asociativas subsume el de las entidades débiles y resulta todavía más potente. Es decir, siempre que utilicemos una entidad débil podremos sustituirla por una entidad asociativa, pero no al revés.

Ejemplo de sustitución de una entidad débil por una asociativa

A continuación se muestra la entidad débil *despacho*, que tiene la interrelación *asignación* con la entidad *empleado*.



Podríamos modelizar este caso haciendo que *despacho* fuese una entidad asociativa si consideramos una nueva entidad *número-despacho* que contiene simplemente números de despachos. Entonces, la entidad asociativa *despacho* se obtiene de la interrelación entre *edificio* y *número-despacho*.



Aunque las entidades débiles se puedan sustituir por el mecanismo de las entidades asociativas, es adecuado mantenerlas en el modelo ER porque resultan menos complejas y son suficientes para modelizar muchas de las situaciones que se producen en el mundo real.

2.3. Ejemplo: base de datos del personal de una entidad bancaria

En este subapartado veremos un ejemplo de diseño conceptual de una base de datos mediante el modelo ER.

Se trata de diseñar una base de datos para la gestión del personal de una entidad bancaria determinada que dispone de muchos empleados y de una amplia red de agencias. La siguiente descripción resume los requisitos de los usuarios de la futura base de datos:

a) Los empleados se identifican por un código de empleado, y también deseamos conocer su DNI, su NSS, su nombre y su apellido. Será importante registrar su ciudad de residencia, considerando que hay ciudades donde no reside ningún empleado.

b) Interesa saber en qué ciudades están ubicadas las diversas agencias de la entidad bancaria. Estas agencias bancarias se identifican por la ciudad donde están y por un nombre que permite distinguir las agencias de una misma ciudad. Se quiere tener constancia del número de habitantes de las ciudades, así como de la dirección y el número de teléfono de las agencias. Se debe consi-

derar que la base de datos también incluye ciudades donde no hay ninguna agencia.

c) Un empleado, en un momento determinado, trabaja en una sola agencia, lo cual no impide que pueda ser trasladado a otra o, incluso, que vuelva a trabajar en una agencia donde ya había trabajado anteriormente. Se quiere tener constancia del historial del paso de los empleados por las agencias.

d) Los empleados pueden tener títulos académicos (aunque no todos los tienen). Se quiere saber qué títulos tienen los empleados.

e) Cada empleado tiene una categoría laboral determinada (auxiliar, oficial de segunda, oficial de primera, etc.). A cada categoría le corresponde un sueldo base determinado y un precio por hora extra también determinado. Se quiere tener constancia de la categoría actual de cada empleado, y del sueldo base y el precio de la hora extra de cada categoría.

f) Algunos empleados (no todos) están afiliados a alguna central sindical. Se ha llegado al pacto de descontar de la nómina mensual la cuota sindical a los afiliados a cada central. Esta cuota es única para todos los afiliados a una central determinada. Es necesario almacenar las afiliaciones a una central de los empleados y las cuotas correspondientes a las diferentes centrales sindicales.

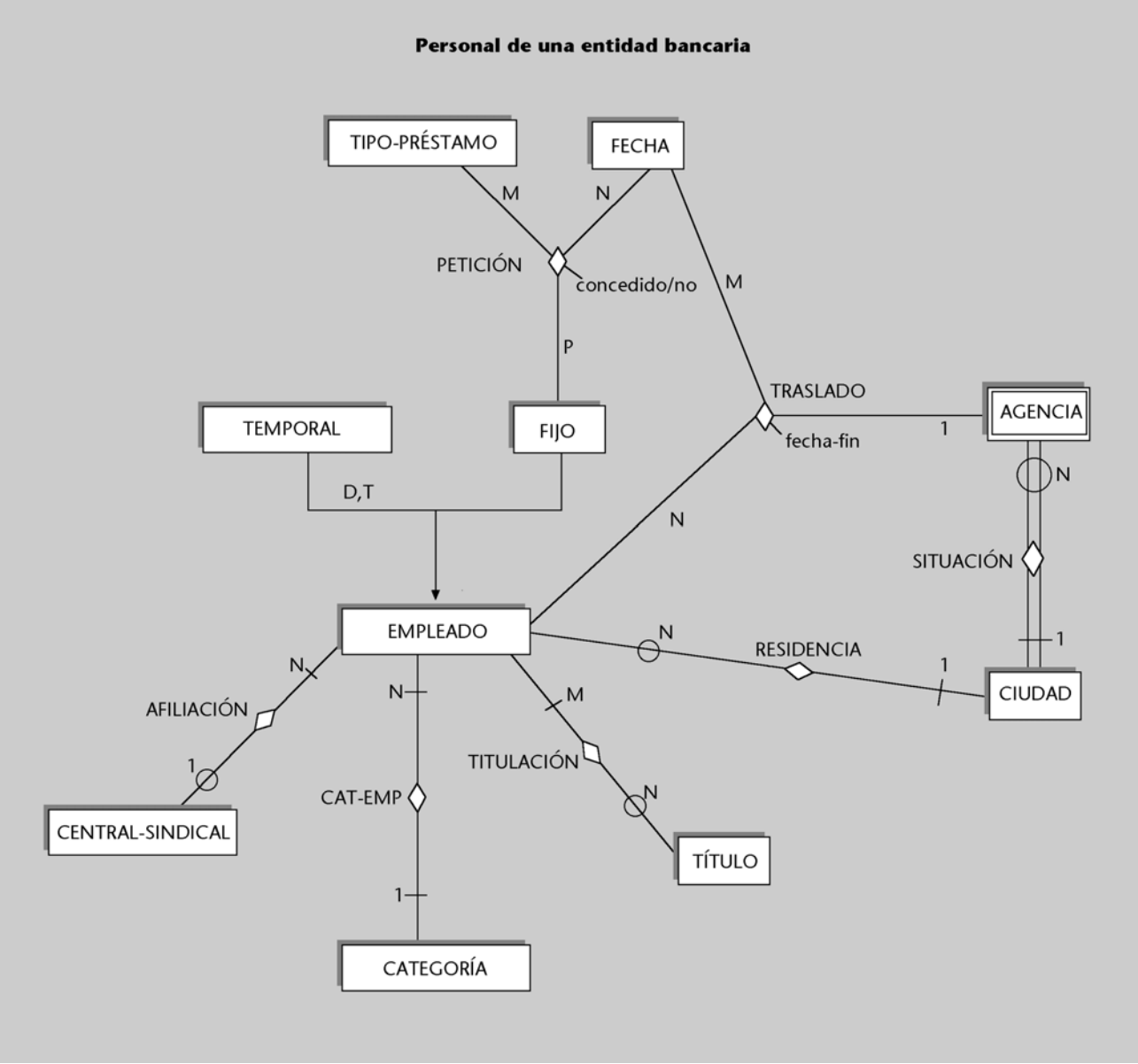
g) Hay dos tipos de empleados diferentes:

- Los que tienen contrato fijo, cuya antigüedad queremos conocer.
- Los que tienen contrato temporal, de los cuales nos interesa saber las fechas de inicio y finalización de su último contrato.

Si un empleado temporal pasa a ser fijo, se le asigna un nuevo código de empleado; consideraremos que un empleado fijo nunca pasa a ser temporal. Todo lo que se ha indicado hasta ahora (traslados, categorías, afiliación sindical, etc.) es aplicable tanto a empleados fijos como a temporales.

h) Los empleados fijos tienen la posibilidad de pedir diferentes tipos preestablecidos de préstamos (por matrimonio, por adquisición de vivienda, por estudios, etc.), que pueden ser concedidos o no. En principio, no hay ninguna limitación a la hora de pedir varios préstamos a la vez, siempre que no se pida más de uno del mismo tipo al mismo tiempo. Se quiere registrar los préstamos pedidos por los empleados, y hacer constar si han sido concedidos o no. Cada tipo de préstamo tiene establecidas diferentes condiciones; de estas condiciones, en particular, nos interesará saber el tipo de interés y el periodo de vigencia del préstamo.

La siguiente figura muestra un diagrama ER que satisface los requisitos anteriores:



Los atributos de las entidades que figuran en el diagrama son los siguientes (las claves primarias se han subrayado):

EMPLEADO
    código-emplado, dni, nss, nombre, apellido
FIJO (entidad subclase de empleado)
    código-empleado, antigüedad
TEMPORAL (entidad subclase de empleado)
    código-empleado, fecha-inicio-cont, fecha-final-cont
CIUDAD
    nombre-ciudad, número-hab

AGENCIA (entidad débil: nombre-agencia la identifica parcialmente,
se identifica completamente con la ciudad de situación)
nombre-agencia, dirección, teléfono
TÍTULO
nombre-título
CATEGORÍA
nombre-categ, sueldo-base, hora-extra
CENTRAL-SINDICAL
central, cuota
TIPO-PRÉSTAMO
código-préstamo, tipo-interés, período-vigencia
FECHA
fecha

A continuación, comentaremos los aspectos que pueden resultar más complejos de este modelo ER:

1) La entidad *agencia* se ha considerado una entidad débil porque su atributo *nombre-agencia* sólo permite distinguir las agencias situadas en una misma ciudad, pero para identificar de forma total una agencia, es necesario saber en qué ciudad está situada. De este modo, la interrelación *situación* es la que nos permite completar la identificación de la entidad *agencia*.

2) La interrelación *petición* es ternaria y asocia a empleados fijos que hacen peticiones de préstamos, tipos de préstamos pedidos por los empleados y fechas en las que se hacen estas peticiones.

3) El lado de la entidad *fecha* se conecta con "muchos" porque un mismo empleado puede pedir un mismo tipo de préstamo varias veces en fechas distintas. La entidad *fijo* se conecta con "muchos" porque un tipo de préstamo determinado puede ser pedido en una misma fecha por varios empleados. También la entidad *tipo-préstamo* se conecta con "muchos" porque es posible que un empleado en una fecha determinada pida más de un préstamo de tipo diferente.

4) El atributo *concedido/no* indica si el préstamo se ha concedido o no. Es un atributo de la interrelación porque su valor depende al mismo tiempo del empleado fijo que hace la petición, del tipo de préstamo pedido y de la fecha de petición.

5) La interrelación *traslado* también es una interrelación ternaria que permite registrar el paso de los empleados por las distintas agencias. Un traslado concreto asocia a un empleado, una agencia donde él trabajará y una fecha inicial en la que empieza a trabajar en la agencia. El atributo de la interrelación *fecha-fin* indica en qué fecha finaliza su asignación a la agencia (*fecha-fin* tendrá el valor nulo cuando

un empleado trabaja en una agencia en el momento actual y no se sabe cuándo se le trasladará). Observad que *fecha-fin* debe ser un atributo de la interrelación. Si se colocase en una de las tres entidades interrelacionadas, no podría ser un atributo univaluado.

Conviene observar que esta interrelación no registra todas y cada una de las fechas en las que un empleado está asignado a una agencia, sino sólo la fecha inicial y la fecha final de la asignación. Es muy habitual que, para informaciones que son ciertas durante todo un periodo de tiempo, se registre en la base de datos únicamente el inicio y el final del periodo.

Notad que la entidad *agencia* se ha conectado con "uno" en la interrelación *traslado*, porque no puede ocurrir que, en una fecha, un empleado determinado sea trasladado a más de una agencia.

6) Finalmente, comentaremos la generalización/especialización de la entidad *empleado*. Los empleados pueden ser de dos tipos; se quieren registrar propiedades diferentes para cada uno de los tipos y también se requieren algunas propiedades comunes a todos los empleados. Por este motivo, es adecuado utilizar una generalización/especialización.

3. Diseño lógico: la transformación del modelo ER al modelo relacional

En este apartado trataremos el diseño lógico de una base de datos relacional. Partiremos del resultado de la etapa del diseño conceptual expresado mediante el modelo ER y veremos cómo se puede transformar en una estructura de datos del modelo relacional.

3.1. Introducción a la transformación de entidades e interrelaciones

Los elementos básicos del modelo ER son las entidades y las interrelaciones:

a) Las entidades, cuando se traducen al modelo relacional, originan **relaciones**.

b) Las interrelaciones, en cambio, cuando se transforman, pueden dar lugar a **claves foráneas** de alguna relación ya obtenida o pueden dar lugar a una **nueva relación**.

En el caso de las interrelaciones, es necesario tener en cuenta su grado y su conectividad para poder decidir cuál es la transformación adecuada:

1) Las interrelaciones binarias 1:1 y 1:N dan lugar a claves foráneas.

2) Las interrelaciones binarias M:N y todas las *n*-arias se traducen en nuevas relaciones.

En los subapartados siguientes explicaremos de forma más concreta las transformaciones necesarias para obtener un esquema relacional a partir de un modelo ER. Más adelante proporcionamos una tabla que resume los aspectos más importantes de cada una de las transformaciones para dar una visión global sobre ello. Finalmente, describimos su aplicación en un ejemplo.

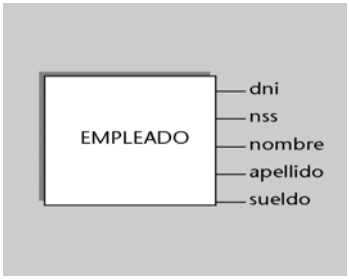
Encontraréis la tabla de las transformaciones en el subapartado 3.10. de esta unidad; en el subapartado 3.11. veremos el ejemplo de aplicación.

3.2. Transformación de entidades

Empezaremos el proceso transformando todas las entidades de un modelo ER adecuadamente. Cada entidad del modelo ER se transforma en una relación del modelo relacional. Los atributos de la entidad serán atributos de la relación y, de forma análoga, la clave primaria de la entidad será la clave primaria de la relación.

Ejemplo de transformación de una entidad

Según esto, la entidad de la figura del margen se transforma en la relación que tenemos a continuación:



EMPLEADO(DNI, NSS, nombre, apellido, sueldo)

Una vez transformadas todas las entidades en relaciones, es preciso transformar todas las interrelaciones en las que intervienen estas entidades.

Si una entidad interviene en alguna interrelación binaria 1:1 o 1:N, puede ser necesario añadir nuevos atributos a la relación obtenida a partir de la entidad. Estos atributos formarán claves foráneas de la relación.

Veremos las transformaciones de las interrelaciones binarias en el siguiente subapartado.

3.3. Transformación de interrelaciones binarias

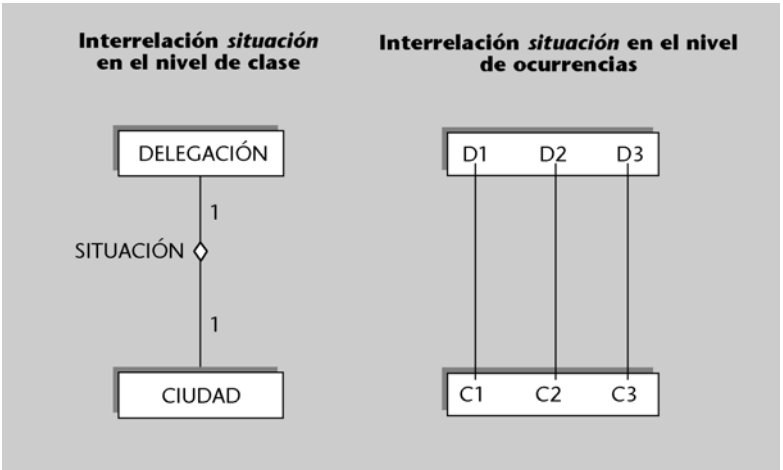
Para transformar una interrelación binaria es necesario tener en cuenta su conectividad, y si las entidades son obligatorias u opcionales en la interrelación.

3.3.1. Conectividad 1:1

Nuestro punto de partida es que las entidades que intervienen en la interrelación 1:1 ya se han transformado en relaciones con sus correspondientes atributos.

Entonces sólo será necesario añadir a cualquiera de estas dos relaciones una clave foránea que referencie a la otra relación.

Ejemplo de transformación de una interrelación binaria 1:1



Para la interrelación de la figura anterior, tenemos dos opciones de transformación:

- Primera opción:

DELEGACIÓN(nombre-del, ..., nombre-ciudad)
donde {nombre-ciudad} referencia CIUDAD
CIUDAD(nombre-ciudad, ...)

- Segunda opción:

DELEGACIÓN(nombre-del, ...)
CIUDAD(nombre-ciudad, ..., nombre-del)
donde {nombre-del} referencia DELEGACIÓN

Ambas transformaciones nos permiten saber en qué ciudad hay una delegación, y qué delegación tiene una ciudad. De este modo, reflejan correctamente el significado de la interrelación *situación* del modelo ER.

En la primera transformación, dado que una delegación está situada en una sola ciudad, el atributo *nombre-ciudad* tiene un único valor para cada valor de la clave primaria {*nombre-del*}. Observad que, si pudiese tener varios valores, la solución no sería correcta según la teoría relacional.

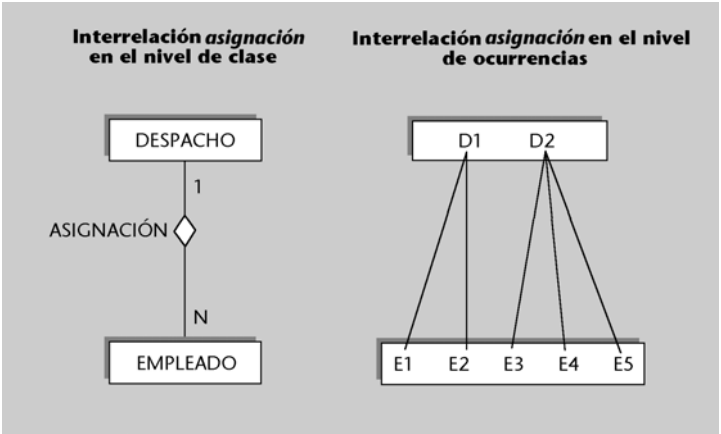
En la segunda transformación, teniendo en cuenta que una ciudad tiene una sola delegación, el atributo *nombre-del* también toma un solo valor para cada valor de la clave primaria {*nombre-ciudad*}.

También es necesario tener en cuenta que, en las dos transformaciones, la clave foránea que se les añade se convierte en una clave alternativa de la relación porque no admite valores repetidos. Por ejemplo, en la segunda transformación no puede haber más de una ciudad con la misma delegación; de este modo, *nombre-del* debe ser diferente para todas las tuplas de *CIUDAD*.

3.3.2. Conectividad 1:N

Partimos del hecho de que las entidades que intervienen en la interrelación 1:N ya se han trasformado en relaciones con sus correspondientes atributos. En este caso sólo es necesario añadir en la relación correspondiente a la entidad del lado N, una clave foránea que referencie la otra relación.

Ejemplo de transformación de una interrelación binaria 1:N



La interrelación de la figura anterior se transforma en:

DESPACHO(desp, ...)
EMPLEADO(emp, ..., desp)
donde {desp}referencia DESPACHO

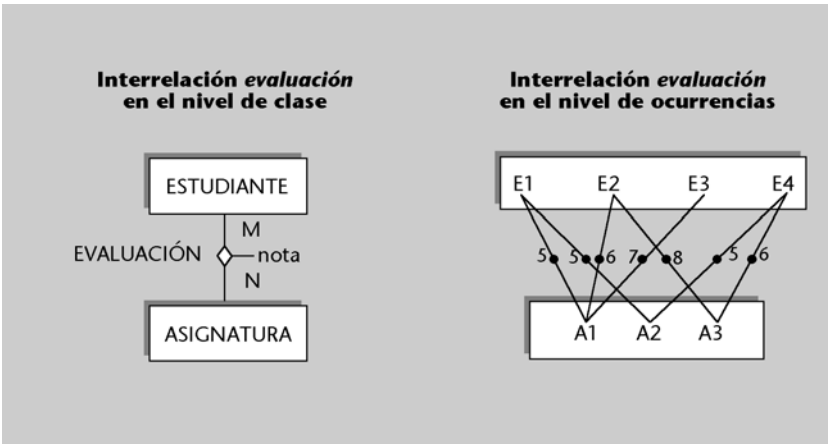
Esta solución nos permite saber en qué despacho está asignado cada empleado, y también nos permite consultar, para cada despacho, qué empleados hay. Es decir, refleja correctamente el significado de la interrelación *asignación*.

Teniendo en cuenta que un empleado está asignado a un único despacho, el atributo *desp* tiene un valor único para cada valor de la clave primaria {*emp*}. Si hubiésemos puesto la clave foránea {*emp*} en la relación *DESPACHO*, la solución habría sido incorrecta, porque *emp* habría tomado varios valores, uno para cada uno de los distintos empleados que pueden estar asignados a un despacho.

3.3.3. Conectividad M:N

Una interrelación M:N se transforma en una relación. Su clave primaria estará formada por los atributos de la clave primaria de las dos entidades interrelacionadas. Los atributos de la interrelación serán atributos de la nueva relación.

Ejemplo de transformación de una interrelación binaria M:N



La interrelación de la figura anterior se transforma en:

ESTUDIANTE(est, ...)
ASIGNATURA(asig, ...)
EVALUACIÓN(est,asig, nota)
donde {est} referencia ESTUDIANTE
y {asig} referencia ASIGNATURA

Observad que la clave de *evaluacion* debe constar tanto de la clave de *estudiante* como de la clave de *asignatura* para identificar completamente la relación.

La solución que hemos presentado refleja correctamente la interrelación *evaluación* y su atributo *nota*. Permite saber, para cada estudiante, qué notas obtiene de las varias asignaturas y, para cada asignatura, qué notas tienen los diferentes estudiantes de aquella asignatura.

En el caso M:N no podemos utilizar claves foráneas para transformar la interrelación, porque obtendríamos atributos que necesitarían tomar varios valores, y esto no se permite en el modelo relacional.

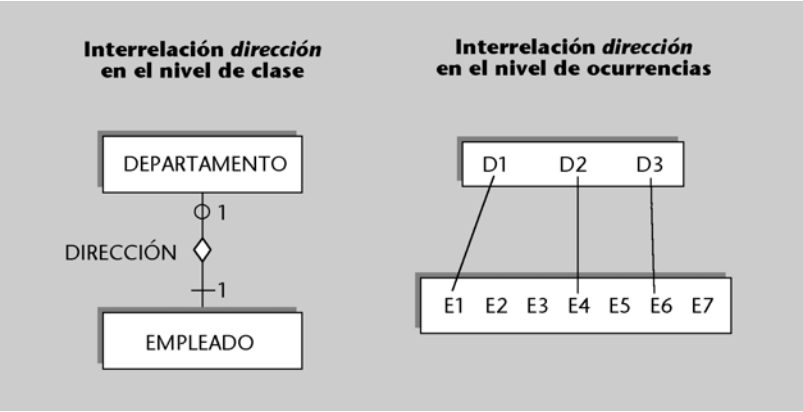
3.3.4. Influencia de la dependencia de existencia en la transformación de las interrelaciones binarias

La dependencia de existencia, o más concretamente, el hecho de que alguna de las entidades sea opcional en una interrelación se debe tener en cuenta al hacer la transformación de algunas relaciones binarias 1:1 y 1:N.

Si una de las entidades es opcional en la interrelación, y la transformación ha consistido en poner una clave foránea en la relación que corresponde a la otra entidad, entonces esta clave foránea puede tomar valores nulos.

Ejemplo de transformación de una entidad opcional en la interrelación

En el ejemplo siguiente, la entidad *departamento* es opcional en *dirección* y, por lo tanto, puede haber empleados que no sean directores de ningún departamento.



En principio, hay dos opciones de transformación:

- Primera opción:

```
DEPARTAMENTO(dep, ..., emp-dir)
    donde {emp-dir} referencia EMPLEADO
EMPLEADO(emp, ...)
```

- Segunda opción:

```
DEPARTAMENTO(dep, ...)
EMPLEADO(emp, ..., dep)
    donde {dep} referencia DEPARTAMENTO
    y dep puede tomar valores nulos
```

La segunda transformación da lugar a una clave foránea que puede tomar valores nulos (porque puede haber empleados que no son directores de ningún departamento). Entonces será preferible la primera transformación, porque no provoca la aparición de valores nulos en la clave foránea y, de este modo, nos ahorra espacio de almacenamiento.

En las interrelaciones 1:N, el hecho de que la entidad del lado 1 sea opcional también provoca que la clave foránea de la transformación pueda tener valores nulos. En este caso, sin embargo, no se pueden evitar estos valores nulos porque hay una única transformación posible.

3.4. Transformación de interrelaciones ternarias

La transformación de las interrelaciones ternarias presenta similitudes importantes con la transformación de las binarias M:N. No es posible representar la interrelación mediante claves foráneas, sino que es necesario usar una nueva relación. Para que la nueva relación refleje toda la información que modeliza la interrelación, es necesario que contenga las claves primarias de las tres entidades interrelacionadas y los atributos de la interrelación.

Así pues, la transformación de una interrelación ternaria siempre da lugar a una nueva relación, que tendrá como atributos las claves primarias de las tres entidades interrelacionadas y todos los atributos que tenga la interrelación. La clave primaria de la nueva relación depende de la conectividad de la interrelación.

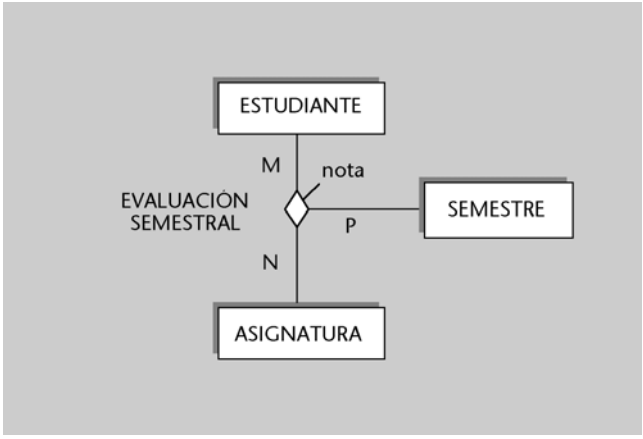
A continuación analizaremos cuál debe ser la clave primaria de la nueva relación según la conectividad. Empezaremos por el caso M:N:P y acabaremos con el caso 1:1:1.

3.4.1. Conectividad M:N:P

Cuando la conectividad de la interrelación es M:N:P, la relación que se obtiene de su transformación tiene como clave primaria todos los atributos que forman las claves primarias de las tres entidades interrelacionadas.

Ejemplo de transformación de una interrelación ternaria M:N:P

Analizaremos la transformación con un ejemplo:



La interrelación anterior se transforma en:

ESTUDIANTE(est, ...)
ASIGNATURA(asig, ...)
SEMESTRE(sem, ...)
EVALUACIÓN-SEMESTRAL(est, asig, sem, nota)
donde {est} referencia ESTUDIANTE,
{asig} referencia ASIGNATURA
y {sem} referencia SEMESTRE

Para identificar completamente la relación, la clave debe constar de la clave de *estudiante*, de la clave de *asignatura* y de la clave de *semestre*. Si nos faltase una de las tres, la clave de la relación podría tener valores repetidos. Consideremos, por ejemplo, que no tuviésemos la clave de *semestre*. Dado que *semestre* está conectada con "muchos" en la interrelación, puede haber estudiantes que han sido evaluados de una misma asignatura en más de un semestre. Entonces, para estos casos habría valores repetidos en la clave de la relación *EVALUACION-SEMESTRAL*.

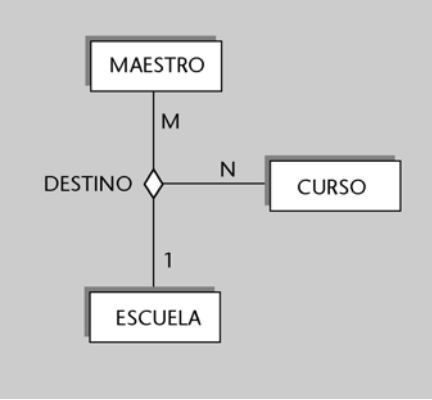
Observad que, del mismo modo que es necesaria la clave de semestre, también lo son la de *estudiante* y la de *asignatura*.

3.4.2. Conectividad M:N:1

Cuando la conectividad de la interrelación es M:N:1, la relación que se obtiene de su transformación tiene como clave primaria todos los atributos que forman las claves primarias de las dos entidades de los lados de la interrelación etiquetados con M y con N.

Ejemplo de transformación de una interrelación ternaria M:N:1

Lo ilustraremos con un ejemplo:



Esta interrelación refleja los destinos que se dan a los maestros de escuela en los diferentes cursos. El 1 que figura en el lado de *escuela* significa que un maestro no puede ser destinado a más de una escuela en un mismo curso.

El ejemplo de la figura se transforma en:

MAESTRO(código-maestro, ...)
CURSO(código-curso, ...)
ESCUELA(código-esc, ...)
DESTINO(código-maestro, código-curso, código-esc)
donde {código-maestro} referencia MAESTRO
{código-curso} referencia CURSO
y {código-esc} referencia ESCUELA

No es necesario que la clave incluya *código-esc* para identificar completamente la relación. Si se fijan un maestro y un curso, no puede haber más de una escuela de destino y, por lo tanto, no habrá claves repetidas.

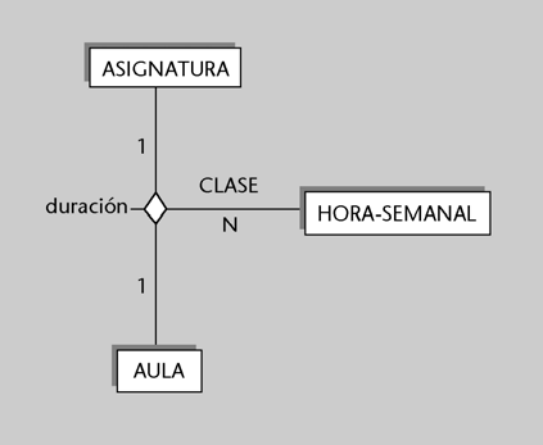
3.4.3. Conectividad N:1:1

Cuando la conectividad de la interrelación es N:1:1, la relación que se consigue de su transformación tiene como clave primaria los atributos que forman la clave primaria de la entidad del lado N y los atributos que forman la clave primaria de cualquiera de las dos entidades que están conectadas con 1.

Así pues, hay dos posibles claves para la relación que se obtiene. Son dos claves candidatas entre las cuales el diseñador deberá escoger la primaria.

Ejemplo de transformación de una interrelación ternaria N:1:1

Veamos un ejemplo de ello:



1) Una posible transformación es la siguiente:

HORA-SEMANAL(código-hora, ...)
AULA(código-aula, ...)
ASIGNATURA(asig, ...)
CLASE (código-hora, código-aula, asig, duración)
donde {código-hora} referencia HORA-SEMANAL,
{código-aula} referencia AULA
y {asig} referencia ASIGNATURA

En este caso, la clave, a pesar de no incluir el atributo *asig*, identifica completamente la relación porque para una hora-semanal y un aula determinadas hay una única asignatura de la que se hace clase a esa hora y en esa aula.

2) La segunda transformación posible es esta otra:

HORA-SEMANAL(código-hora, ...)
AULA(código-aula, ...)
ASIGNATURA(asig, ...)
CLASE (código-hora, código-aula, asig, duración)
donde {código-hora} referencia HORA-SEMANAL,
{código-aula} referencia AULA
y {asig} referencia ASIGNATURA

Ahora la clave incluye el atributo *asig* y, en cambio, no incluye el atributo *código-aula*. La relación también queda completamente identificada porque, para una asignatura y hora-semanal determinadas, de aquella asignatura se da clase en una sola aula a aquella hora.

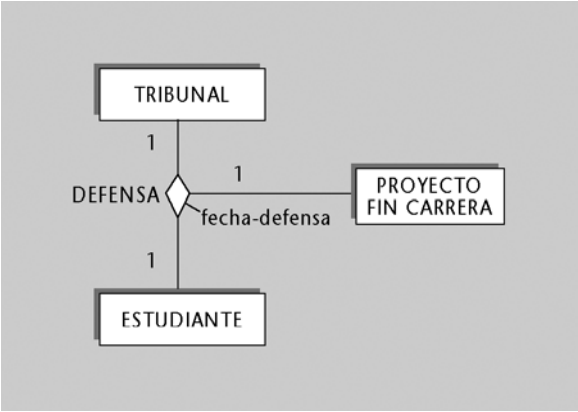
3.4.4. Conectividad 1:1:1

Cuando la conectividad de la interrelación es 1:1:1, la relación que se obtiene de su transformación tiene como clave primaria los atributos que forman la clave primaria de dos entidades cualesquiera de las tres interrelacionadas.

Así pues, hay tres claves candidatas para la relación.

Ejemplo de transformación de una interrelación ternaria 1:1:1

Veamos un ejemplo de ello:



Esta interrelación registra información de defensas de proyectos de fin de carrera. Intervienen en ella el estudiante que presenta el proyecto, el proyecto presentado y el tribunal evaluador.

Hemos considerado que, ...

... si dos estudiantes presentan un mismo proyecto de fin de carrera, el tribunal será necesariamente diferente.

La transformación del ejemplo anterior se muestra a continuación:

TRIBUNAL(trib, ...)
ESTUDIANTE(est, ...)
PROYECTO-FIN-CARRERA(pro, ...)

Para la nueva relación DEFENSA, tenemos las tres posibilidades siguientes:

- Primera opción:

DEFENSA(trib, est, pro, fecha-defensa)
donde {trib} referencia TRIBUNAL,
{est} referencia ESTUDIANTE
y {pro} referencia PROYECTO-FIN-CARRERA

- Segunda opción:

DEFENSA(trib, pro, est, fecha-defensa)
donde {trib} referencia TRIBUNAL,
{est} referencia ESTUDIANTE
y {pro} referencia PROYECTO-FIN-CARRERA

- Tercera opción:

DEFENSA(est, pro, trib, fecha-defensa)
donde {trib} referencia TRIBUNAL,
{est} referencia ESTUDIANTE
y {pro} referencia PROYECTO-FIN-CARRERA

En los tres casos, es posible comprobar que la clave identifica completamente la relación si se tiene en cuenta la conectividad de la interrelación *defensa*.

3.5. Transformación de interrelaciones *n*-arias

La transformación de las interrelaciones *n*-arias se puede ver como una generalización de lo que hemos explicado para las ternarias.

En todos los casos, la transformación de una interrelación *n*-aria consistirá en la obtención de una nueva relación que contiene todos los atributos que forman las claves primarias de las *n* entidades interrelacionadas y todos los atributos de la interrelación.

Podemos distinguir los casos siguientes:

a) Si todas las entidades están conectadas con "muchos", la clave primaria de la nueva relación estará formada por todos los atributos que forman las claves de las *n* entidades interrelacionadas.

b) Si una o más entidades están conectadas con "uno", la clave primaria de la nueva relación estará formada por las claves de $n - 1$ de las entidades interrelacionadas, con la condición de que la entidad, cuya clave no se ha incluido, debe ser una de las que está conectada con "uno".

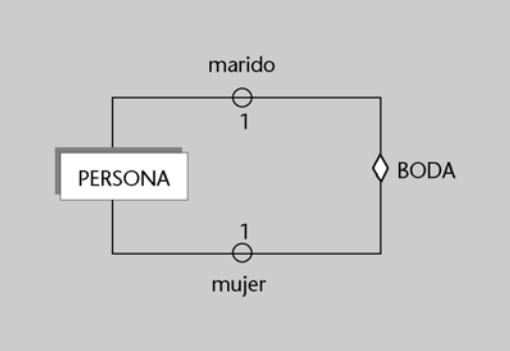
3.6. Transformación de interrelaciones recursivas

Las transformaciones de las interrelaciones recursivas son similares a las que hemos visto para el resto de las interrelaciones.

De este modo, si una interrelación recursiva tiene conectividad 1:1 o 1:N, da lugar a una clave foránea, y si tiene conectividad M:N o es *n*-aria, origina una nueva relación.

Mostraremos la transformación de algunos ejemplos concretos de interrelaciones recursivas para ilustrar los detalles de la afirmación anterior.

Ejemplo de transformación de una interrelación recursiva binaria 1:1



La interrelación de la figura anterior es recursiva, binaria y tiene conectividad 1:1. Las interrelaciones 1:1 originan una clave foránea que se pone en la relación correspondiente a una de las entidades interrelacionadas. En nuestro ejemplo, sólo hay una entidad interrelacionada, la entidad *persona*. Entonces, la clave foránea deberá estar en la relación *PERSONA*. Esta clave foránea deberá referenciar a la misma relación para que refleje una interrelación entre una ocurrencia de persona y otra ocurrencia de persona. Así, obtendremos:

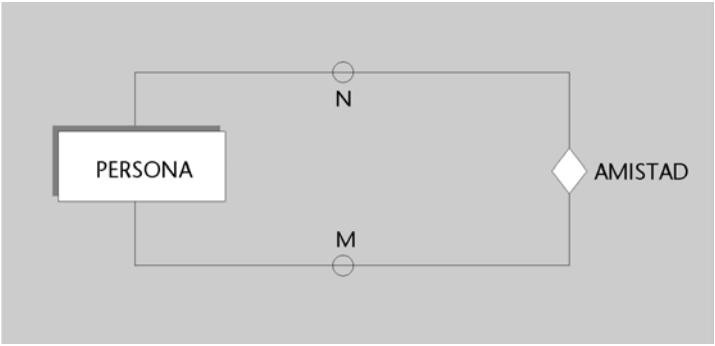
PERSONA (<u>código-per</u>, ..., código-conyuge)
donde {código-conyuge} referencia PERSONA
y código-conyuge admite valores nulos

La clave foránea {*código-conyuge*} referencia la relación *PERSONA* a la que pertenece.

Conviene tener en cuenta que, en casos como éste, los atributos de la clave foránea no pueden tener los mismos nombres que los atributos de la clave primaria que referencian porque, según la teoría relacional, una relación no puede tener nombres de atributos repetidos.

Ejemplo de transformación de una interrelación recursiva M:N

Veamos a continuación un ejemplo en el que interviene una interrelación recursiva y con conectividad M:N.



Las interrelaciones M:N se traducen en nuevas relaciones que tienen como clave primaria las claves de las entidades interrelacionadas.

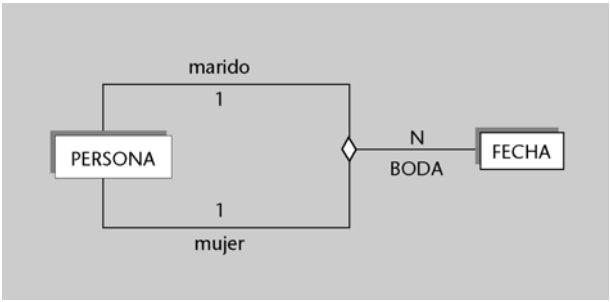
En nuestro ejemplo, la interrelación vincula ocurrencias de persona con otras ocurrencias de *persona*. En este caso, la clave primaria de la nueva relación estará formada por la clave de la

entidad *persona* dos veces. Convendrá dar nombres diferentes a todos los atributos de la nueva relación. De este modo, la traducción del ejemplo anterior será:

PERSONA (código-per, ...)
AMISTAD (código-per, código-per-amiga)
donde {código-per} referencia PERSONA
y {código-per-amiga} referencia PERSONA

Ejemplo de transformación de una interrelación recursiva *n*-aria N:1:1

Finalmente, analizaremos un ejemplo en el que la interrelación recursiva es *n*-aria:



La anterior interrelación *boda* es recursiva, ternaria y tiene conectividad N:1:1. Las interrelaciones N:1:1 originan siempre una nueva relación que contiene, además de los atributos de la interrelación, todos los atributos que forman la clave primaria de las tres entidades interrelacionadas.

En nuestro ejemplo, la interrelación asocia ocurrencias de *persona* con otras ocurrencias de *persona* y con ocurrencias de *fecha*. Entonces, la clave de *persona* tendrá que figurar dos veces en la nueva relación, y la clave de *fecha*, solo una.

La clave primaria de la relación que se obtiene para interrelaciones N:1:1 está formada por la clave de la entidad del lado N y por la clave de una de las entidades de los lados 1.

En nuestro ejemplo, en los dos lados 1 de la interrelación tenemos la misma entidad: *persona*. La clave primaria estará formada por la clave de la entidad *fecha* y por la clave de la entidad *persona*.

Según todo esto, la transformación será la siguiente:

PERSONA(código-per, ...)
FECHA(fecha-bod, ...)
BODA (fecha-bod, código-per, código-conyuge)
donde {fecha-bod} referencia FECHA,
{código-per} referencia PERSONA
y {código-conyuge} referencia PERSONA

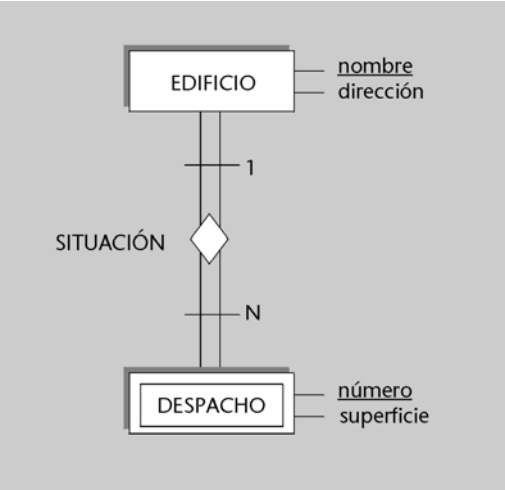
3.7. Transformación de entidades débiles

Las entidades débiles se traducen al modelo relacional igual que el resto de entidades, con una pequeña diferencia. Estas entidades siempre están en el lado N de una interrelación 1:N que completa su identificación.

Así pues, la clave foránea originada por esta interrelación 1:N debe formar parte de la clave primaria de la relación correspondiente a la entidad débil.

Ejemplo de transformación de entidad débil

Lo explicaremos con un ejemplo:



Este ejemplo se transforma tal y como se muestra a continuación:

EDIFICIO(nombre, dirección)
DESPACHO(nombre, número, superficie)
donde {nombre} referencia EDIFICIO

Observad que la clave foránea {*nombre*} forma parte también de la clave primaria de *DESPACHO*. Si no fuese así, y la clave primaria contuviese sólo el atributo *número*, los despachos no quedarían totalmente identificados, teniendo en cuenta que puede haber despachos situados en edificios diferentes que tengan el mismo número.

3.8. Transformación de la generalización/especialización

Cada una de las entidades superclase y subclase que forman parte de una generalización/especialización se transforma en una relación:

a) La relación de la entidad superclase tiene como clave primaria la clave de la entidad superclase y contiene todos los atributos comunes.

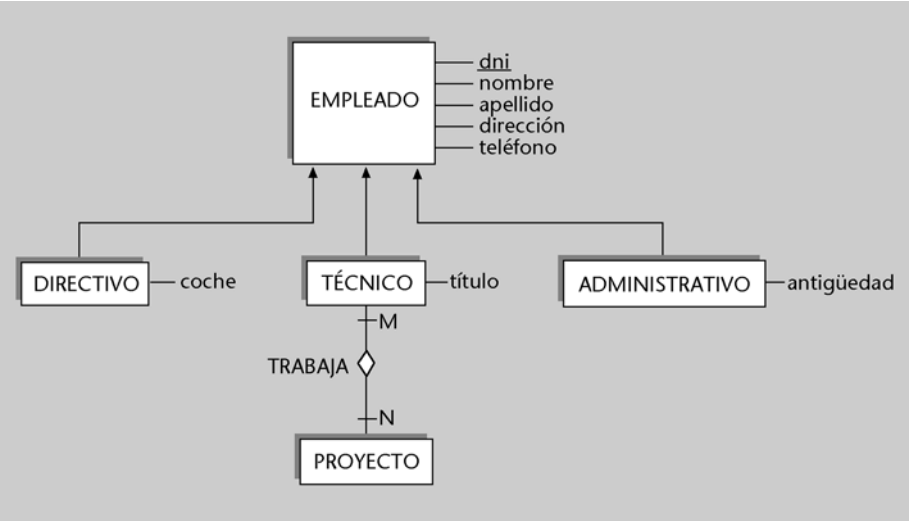
b) Las relaciones de las entidades subclase tienen como clave primaria la clave de la entidad superclase y contienen los atributos específicos de la subclase.

Ejemplo de transformación de la generalización/especialización

Veamos un ejemplo (consultad el gráfico en la página siguiente) que contiene una generalización/especialización y, también, una interrelación M:N en la que interviene una de las entidades subclase. Este ejemplo se traduce al modelo relacional como se indica a continuación:

EMPLEADO(DNI, nombre, dirección, teléfono)
DIRECTIVO(DNI, coche)
donde {DNI} referencia EMPLEADO

ADMINISTRATIVO (DNI, antigüedad)
donde {DNI} referencia EMPLEADO
TÉCNICO (DNI, título)
donde {DNI} referencia EMPLEADO
PROYECTO(pro, ...)
TRABAJA(DNI, pro, superficie)
donde {DNI} referencia TÉCNICO
y {pro} referencia PROYECTO



Conviene observar que los atributos comunes se han situado en la relación *EMPLEADO* y que los atributos específicos se han situado en la relación de su entidad subclase. De este modo, *coche* está en *DIRECTIVO*, *título* en *TÉCNICO* y *antigüedad* en *ADMINISTRATIVO*.

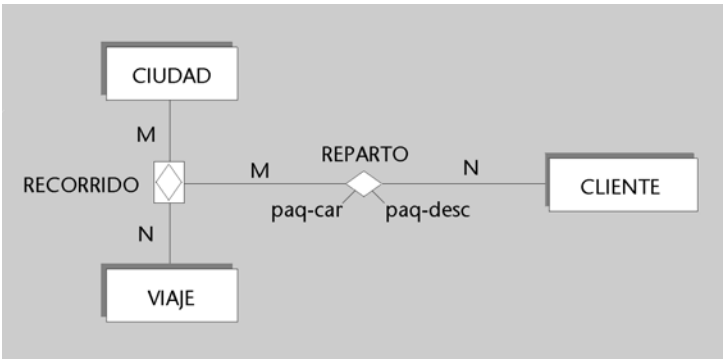
Por otro lado, la interrelación específica para los empleados técnicos denominada *trabaja* se transforma en la relación *TRABAJA*. Observad que esta relación tiene una clave foránea que referencia sólo a los empleados técnicos, y no a los empleados directivos o administrativos.

3.9. Transformación de entidades asociativas

Una entidad asociativa tiene su origen en una interrelación. En consecuencia, sucede que la transformación de la interrelación originaria es, al mismo tiempo, la transformación de la entidad asociativa.

Ejemplo de transformación de una entidad asociativa

Veamos un ejemplo, que incluye una entidad asociativa interrelacionada con otra entidad:



La transformación del ejemplo anterior será:

```
CIUDAD(nombre-ciudad, ...)
VIAJE(id-viaje, ...)
RECORRIDO (nombre-ciudad, id-viaje)
    donde {nombre-ciudad} referencia CIUDAD
    e {id-viaje} referencia VIAJE
CLIENTE {código-cliente, ...}
REPARTO(nombre-ciudad, id-viaje, código-cliente, paq-car, paq-desc)
    donde {nombre-ciudad, id-viaje} referencia RECORRIDO
    y {código-cliente} referencia CLIENTE
```

Tal y como se puede observar, la traducción de la interrelación *recorrido* es, al mismo tiempo, la traducción de su entidad asociativa.

La relación *REPARTO* nos ilustra la transformación de una interrelación en la que participa una entidad asociativa. Puesto que se trata de una interrelación M:N entre *recorrido* y *ciudad*, una parte de la clave primaria de *REPARTO* referencia la clave de *RECORRIDO*, y el resto, la clave de *CIUDAD*.

3.10. Resumen de la transformación del modelo ER al modelo relacional

La tabla que mostramos a continuación resume los aspectos más básicos de las transformaciones que hemos descrito en las secciones anteriores, con el objetivo de presentar una visión rápida de los mismos:

Elemento del modelo ER	Transformación al modelo relacional
Entidad	Relación
Interrelación 1:1	Clave foránea
Interrelación 1:1	Clave foránea
Interrelación M:N	Relación
Interrelación *n*-aria	Relación
Interrelación recursiva	Como en las interrelaciones no recursivas: • Clave foránea para binarias 1:1 y 1:N • Relación para binarias M:N y *n*-arias
Entidad débil	La clave foránea de la interrelación identificadora forma parte de la clave primaria
Generalización/especialización	• Relación para la entidad superclase • Relación para cada una de las entidades subclase
Entidad asociativa	La transformación de la interrelación que la origina es a la vez su transformación

3.11. Ejemplo: base de datos del personal de una entidad bancaria

En este apartado aplicaremos las transformaciones que hemos explicado en el caso práctico de la base de datos del personal de una entidad bancaria. Antes hemos presentado el diseño conceptual de esta base de datos. A continuación, veremos su transformación al modelo relacional.

Hemos presentado el diseño conceptual de la base de datos del personal de la entidad bancaria en el subapartado 2.3 de esta unidad didáctica.

Empezaremos por transformar todas las entidades en relaciones y todas las interrelaciones 1:1 y 1:N en claves foráneas de estas relaciones.

EMPLEADO(código-empleado, DNI, NSS, nombre, apellido, nombre-categ, central, ciudad-res)
donde {nombre-categ} referencia CATEGORÍA,
{central} referencia CENTRAL-SINDICAL,
el atributo central admite valores nulos
y {ciudad-res} referencia CIUDAD
FIJO(código-empleado, antigüedad)
donde {código-empleado} referencia EMPLEADO
TEMPORAL(código-empleado, fecha-inicio-cont, fecha-final-cont)
donde {código-empleado} referencia EMPLEADO
CIUDAD(nombre-ciudad, número-hab)
AGENCIA(nombre-ciudad, nombre-agencia, dirección, teléfono)
donde {nombre-ciudad} referencia CIUDAD
TÍTULO(nombre-título)
CATEGORÍA(nombre-categoría, sueldo-base, hora-extra)
CENTRAL-SINDICAL(central, cuota)
TIPO-PRÉSTAMO(código-préstamo, tipo-interés, período-vigencia)
FECHA(fecha)

Observad que, en la transformación de la generalización/especialización correspondiente a la entidad *empleado*, hemos situado los atributos comunes a la relación *EMPLEADO* y los atributos específicos se han situado en las relaciones *FIJO* y *TEMPORAL*.

En la relación *AGENCIA*, el atributo *nombre-ciudad* es una clave foránea y al mismo tiempo forma parte de la clave primaria porque *agencia* es una entidad débil que requiere la interrelación *situacion* para ser identificada.

Veamos ahora las relaciones que se obtienen a partir de la transformación de las interrelaciones binarias y *n*-arias:

TITULACIÓN(código-empleado, nombre-título)
donde {código-empleado} referencia EMPLEADO
y {nombre-título} referencia TÍTULO
TRASLADO(código-empleado, fecha, nombre-ciudad, nombre-agencia, fecha-fin)
donde {código-empleado} referencia EMPLEADO,
{nombre-ciudad, nombre-agencia} referencia AGENCIA
y {fecha} referencia FECHA
PETICIÓN(código-empleado, código-préstamo, fecha, concedido/no)
donde {código-empleado) referencia FIJO
{código-préstamo} referencia TIPO-PRÉSTAMO
y {fecha} referencia FECHA

Para elegir las claves primarias adecuadas, se ha tenido en cuenta la conectividad de las interrelaciones.

Resumen

Esta unidad es una introducción a un tema de gran interés: el **diseño de bases de datos**.

En primer lugar, hemos explicado qué se entiende por *diseñar una base de datos* y hemos analizado las etapas en las que se puede descomponer el proceso de diseño:

- la **etapa del diseño conceptual**,
- la **etapa del diseño lógico**,
- la **etapa del diseño físico**.

En el resto de la unidad hemos tratado el diseño conceptual y el diseño lógico de la base de datos. No hemos estudiado el diseño físico porque requiere unos conocimientos previos de estructuras de implementación física que hacen inadecuado explicarlo en este curso. 

Para el diseño conceptual hemos adoptado el enfoque del **modelo ER**, un modelo de datos muy utilizado y comprensible. Hemos descrito las diversas construcciones que proporciona y hemos dado ejemplos de aplicación a casos prácticos.

En lo que respecta al diseño lógico, lo hemos centrado en el caso de utilización de la **tecnología relacional**. De este modo, hemos explicado cómo se puede transformar un modelo conceptual expresado mediante el modelo ER en una estructura de datos del modelo relacional.

Ejercicios de autoevaluación

1. Haced un diseño conceptual de una base de datos mediante el modelo ER que satisfaga los requisitos que se resumen a continuación:
a) Un directivo de un club de fútbol quiere disponer de una base de datos que le permita controlar datos que le interesan sobre competiciones, clubes, jugadores, entrenadores, etc. de ámbito estatal.
b) Los clubes disputan cada temporada varias competiciones (liga, copa, etc.) entre sí. Nuestro directivo desea información histórica de las clasificaciones obtenidas por los clubes en las diferentes competiciones a lo largo de todas las temporadas. La clasificación se especificará mediante un número de posición: 1 significa campeón, 2 significa subcampeón, etc.
c) Los distintos clubes están agrupados en las federaciones regionales correspondientes. Toda federación tiene como mínimo un club. Quiere saber el nombre y la fecha de creación de las federaciones así como el nombre y el número de socios de los clubes.
d) Es muy importante la información sobre jugadores y entrenadores. Se identificarán por un código, y quiere saber el nombre, la dirección, el número de teléfono y la fecha de nacimiento de todos. Es necesario mencionar que algunos entrenadores pueden haber sido jugadores en su juventud. De los jugadores, además, quiere saber el peso, la altura, la especialidad o las especialidades y qué dominio tienen de ellas (grado de especialidad). Todo jugador debe tener como mínimo una especialidad, pero puede haber especialidades en las que no haya ningún jugador. De los entrenadores le interesa la fecha en que iniciaron su carrera como entrenadores de fútbol.
e) De todas las personas que figuran en la base de datos (jugadores y entrenadores), quiere conocer el historial de contrataciones por parte de los diferentes clubes, incluyendo el importe y la fecha de baja de cada contratación. En un momento determinado, una persona puede estar contratada por un único club, pero puede cambiar de club posteriormente e, incluso, puede volver a un club en el que ya había trabajado.
f) También quiere registrar las ofertas que las personas que figuran en la base de datos han recibido de los clubes durante su vida deportiva (y de las que se ha enterado). Considera básico tener constancia del importe de las ofertas. Se debe tener en cuenta que, en un momento determinado, una persona puede recibir muchas ofertas, siempre que provengan de clubes distintos.

2. Haced un diseño conceptual de una base de datos mediante el modelo ER que satisfaga los requisitos que se resumen a continuación:
a) Se quiere diseñar una base de datos para facilitar la gestión de una empresa dedicada al transporte internacional de mercancías que opera en todo el ámbito europeo.
b) La empresa dispone de varias delegaciones repartidas por toda la geografía europea. Las delegaciones se identifican por un nombre, y se quiere registrar también su número de teléfono. En una determinada ciudad no hay nunca más de una delegación. Se desea conocer la ciudad donde está situada cada delegación. Se debe suponer que no hay ciudades con el nombre repetido (por lo menos en el ámbito de esta base de datos).
c) El personal de la empresa se puede separar en dos grandes grupos:
- Administrativos, sobre los cuales interesa saber su nivel de estudios.
- Conductores, sobre los que interesa saber el año en el que obtuvieron el carnet de conducir y el tipo de carnet que tienen.

De todo el personal de la empresa, se quiere conocer el código de empleado (que lo identifica), su nombre, su número de teléfono y el año de nacimiento. Todos los empleados están asignados a una delegación determinada. Se quiere tener constancia histórica de este hecho teniendo en cuenta que pueden ir cambiando de delegación (incluso pueden volver a una delegación donde ya habían estado anteriormente).
d) La actividad de la empresa consiste en efectuar los viajes pertinentes para transportar las mercancías según las peticiones de sus clientes. Todos los clientes se identifican por un código de cliente. Se quiere conocer, además, el nombre y el teléfono de contacto de cada uno de ellos.
e) La empresa, para llevar a cabo su actividad, dispone de muchos camiones identificados por un código de camión. Se quiere tener constancia de la matrícula, la marca y la tara de los camiones.
f) Los viajes los organiza siempre una delegación, y se identifican mediante un código de viaje, que es interno de cada delegación (y que se puede repetir en delegaciones diferentes). Para cada uno de los viajes que se han hecho, es necesario saber:
- Qué camión se ha utilizado (ya que cada viaje se hace con un solo camión).
- Qué conductor o conductores han ido (considerando que en viajes largos pueden ir varios conductores). Se quiere saber también el importe de las dietas pagadas a cada conductor (teniendo en cuenta que las dietas pueden ser diferentes para los diferentes conductores de un mismo viaje).
- El recorrido del viaje; es decir, la fecha y la hora en que el camión llega a cada una de las ciudades donde deberá cargar o descargar. Supondremos que un viaje no pasa nunca dos veces por una misma ciudad.

- El número de paquetes cargados y de paquetes descargados en cada ciudad, y para cada uno de los clientes. En un mismo viaje se pueden dejar y/o recoger paquetes en diferentes ciudades por encargo de un mismo cliente. También, en un mismo viaje, se pueden dejar y/o recoger paquetes en una misma ciudad por encargo de diferentes clientes.

3. Haced un diseño conceptual de una base de datos mediante el modelo ER que satisfaga los requisitos que se resumen a continuación:
a) Es necesario diseñar una base de datos para una empresa inmobiliaria con el objetivo de gestionar la información relativa a su cartera de pisos en venta.
b) Cada uno de los pisos que tienen pendientes de vender tiene asignado un código de piso que lo identifica. Además de este código, se quiere conocer la dirección del piso, la superficie, el número de habitaciones y el precio. Tienen estos pisos clasificados por zonas (porque a sus clientes, en ocasiones, sólo les interesan los pisos de una zona determinada) y se quiere saber en qué zona está situado cada piso. Las zonas tienen un nombre de zona que es diferente para cada una de una misma población, pero que pueden coincidir en zonas de poblaciones diferentes. En ocasiones sucede que en algunas de las zonas no tienen ningún piso pendiente de vender.
c) Se quiere tener el número de habitantes de las poblaciones. Se quiere saber qué zonas son limítrofes, (porque, en caso de no disponer de pisos en una zona que desea un cliente, se le puedan ofrecer los que tengan en otras zonas limítrofes). Es necesario considerar que pueden existir zonas sin ninguna zona limítrofe en algunas poblaciones pequeñas que constan de una sola zona.
d) Se disponen de diferentes características codificadas de los pisos, como por ejemplo tener ascensor, ser exterior, tener terraza, etc. Cada característica se identifica mediante un código y tiene una descripción. Para cada característica y cada piso se quiere saber si el piso satisface la característica o no. Además, quieren tener constancia del propietario o los propietarios de cada piso.
e) También necesitan disponer de información relativa a sus clientes actuales que buscan piso (si dos o más personas buscan piso conjuntamente, sólo se guarda información de una de ellas como cliente de la empresa). En particular, interesa saber las zonas donde busca piso cada cliente (sólo en caso de que tenga alguna zona de preferencia).
f) A cada uno de estos clientes le asignan un vendedor de la empresa para que se ocupe de atenderlo. A veces, estas asignaciones varían con el tiempo y se cambia al vendedor asignado a un determinado cliente. También es posible que a un cliente se le vuelva a asignar un vendedor que ya había tenido con anterioridad. Se quiere tener constancia de las asignaciones de los clientes actuales de la empresa.
g) Los vendedores, clientes y propietarios se identifican por un código de persona. Se quiere registrar, de todos, su nombre, dirección y número de teléfono. Además, se quiere disponer del número de Seguridad Social y el sueldo de los vendedores, y del NIF de los propietarios. Puede haber personas que sean al mismo tiempo clientes y propietarios, o bien vendedores y propietarios, etc.
h) Finalmente, para ayudar a programar y consultar las visitas que los clientes hacen a los pisos en venta, se quiere guardar información de todas las visitas correspondientes a los clientes y a los pisos actuales de la empresa. De cada visita hay que saber el cliente que la hace, el piso que se va a ver y la hora concreta en que se inicia la visita. Entendemos que la hora de la visita está formada por la fecha, la hora del día y el minuto del día (por ejemplo, 25-FEB-98, 18:30). Hay que considerar que un mismo cliente puede visitar un mismo piso varias veces para asegurarse de si le gusta o no, y también que para evitar conflictos no se programan nunca visitas de clientes diferentes a un mismo piso y a la misma hora.

4. Transformad a relacional el diseño conceptual que habéis obtenido en el ejercicio 1.

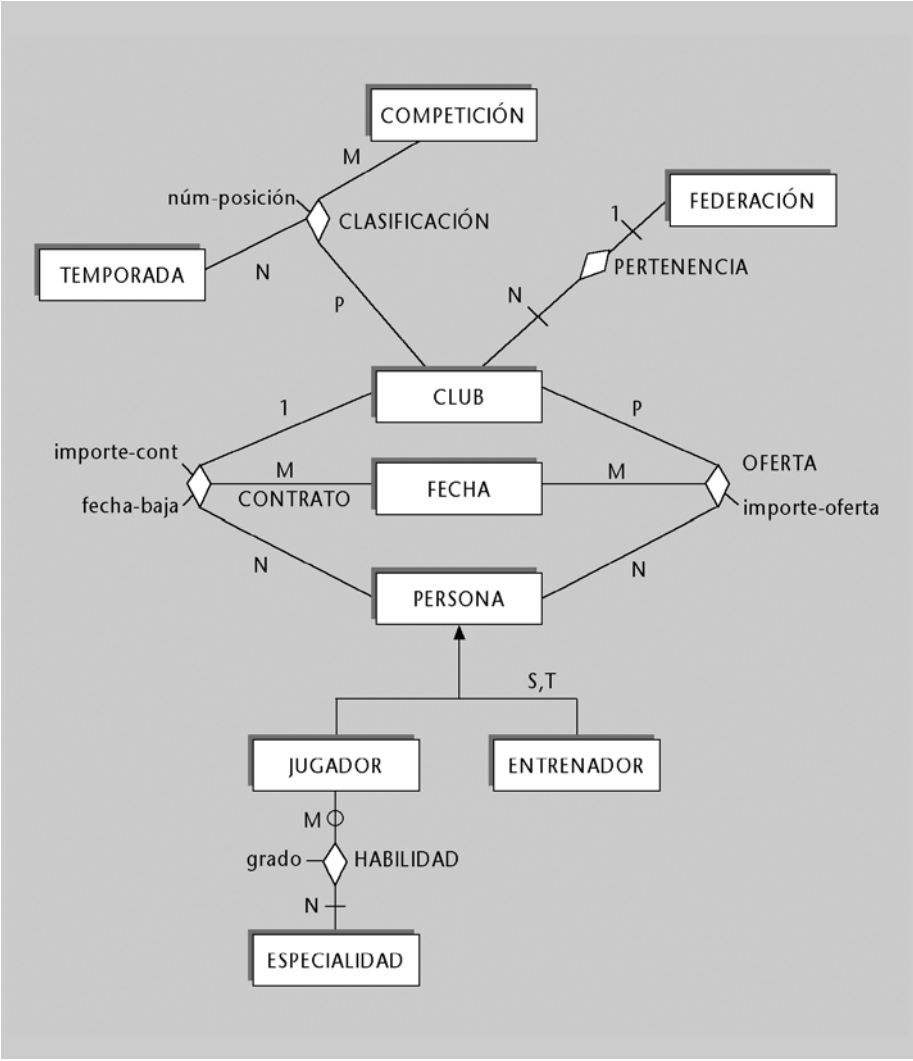
5. Transformad a relacional el diseño conceptual que habéis obtenido en el ejercicio 2.

6. Transformad a relacional el diseño conceptual que habéis obtenido en el ejercicio 3.

Solucionario

Ejercicios de autoevaluación

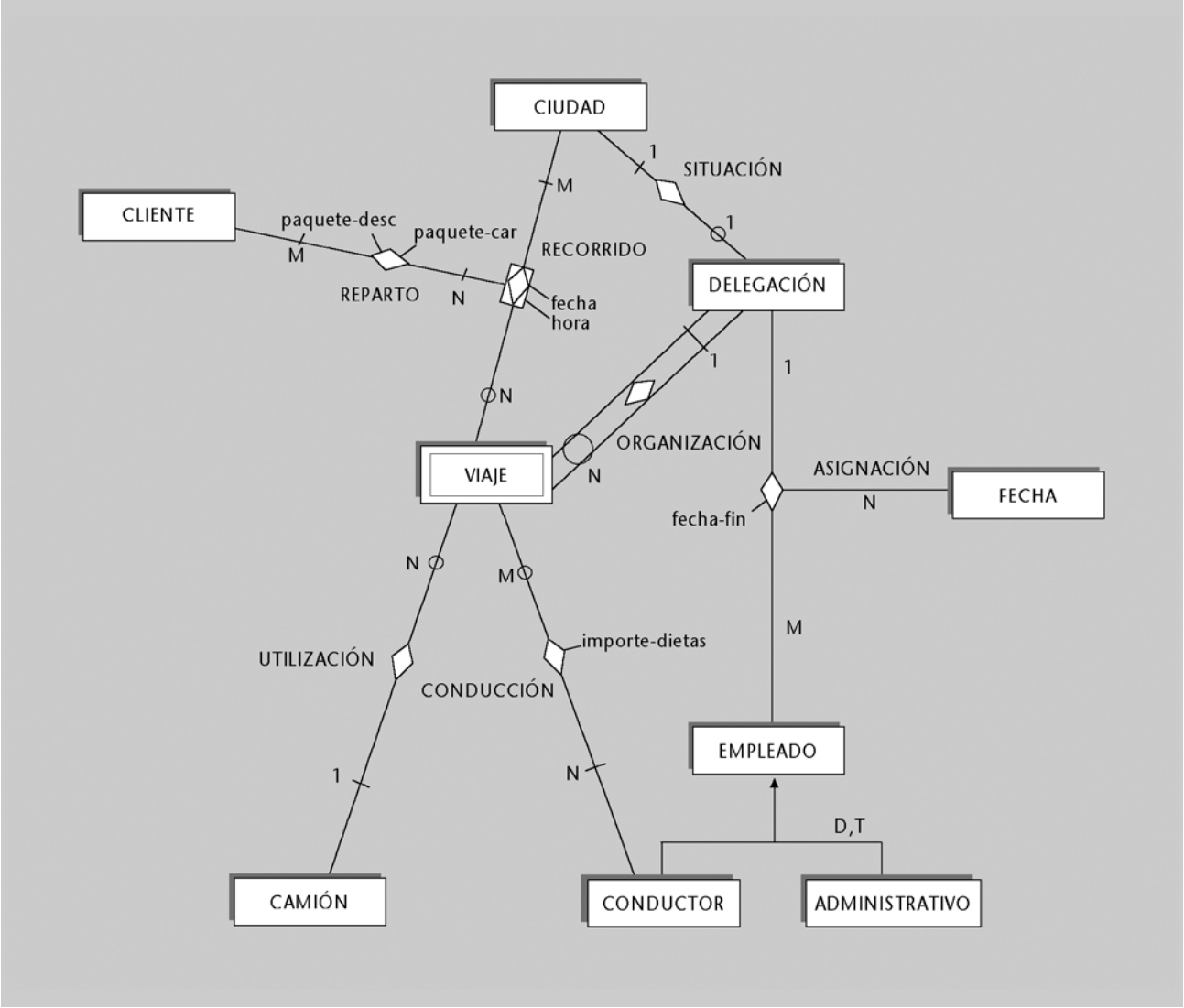
1. La siguiente figura muestra un diagrama ER que satisface los requisitos que se han descrito:



Los atributos de las entidades que figuran en el diagrama son los siguientes (las claves primarias se han subrayado):

COMPETICIÓN
    nombre-comp
TEMPORADA
    código-temp
FEDERACIÓN
    nombre-fed, fecha-creación
CLUB
    nombre-club, número-socios
PERSONA
    código-persona, nombre, dirección, teléfono, fecha-nacimiento
JUGADOR (entidad subclase de persona)
    código-persona, peso, altura
ENTRENADOR (entidad subclase de persona)
    código-persona, fecha-inicio-carrera
ESPECIALIDAD
    nombre-esp
FECHA
    fecha

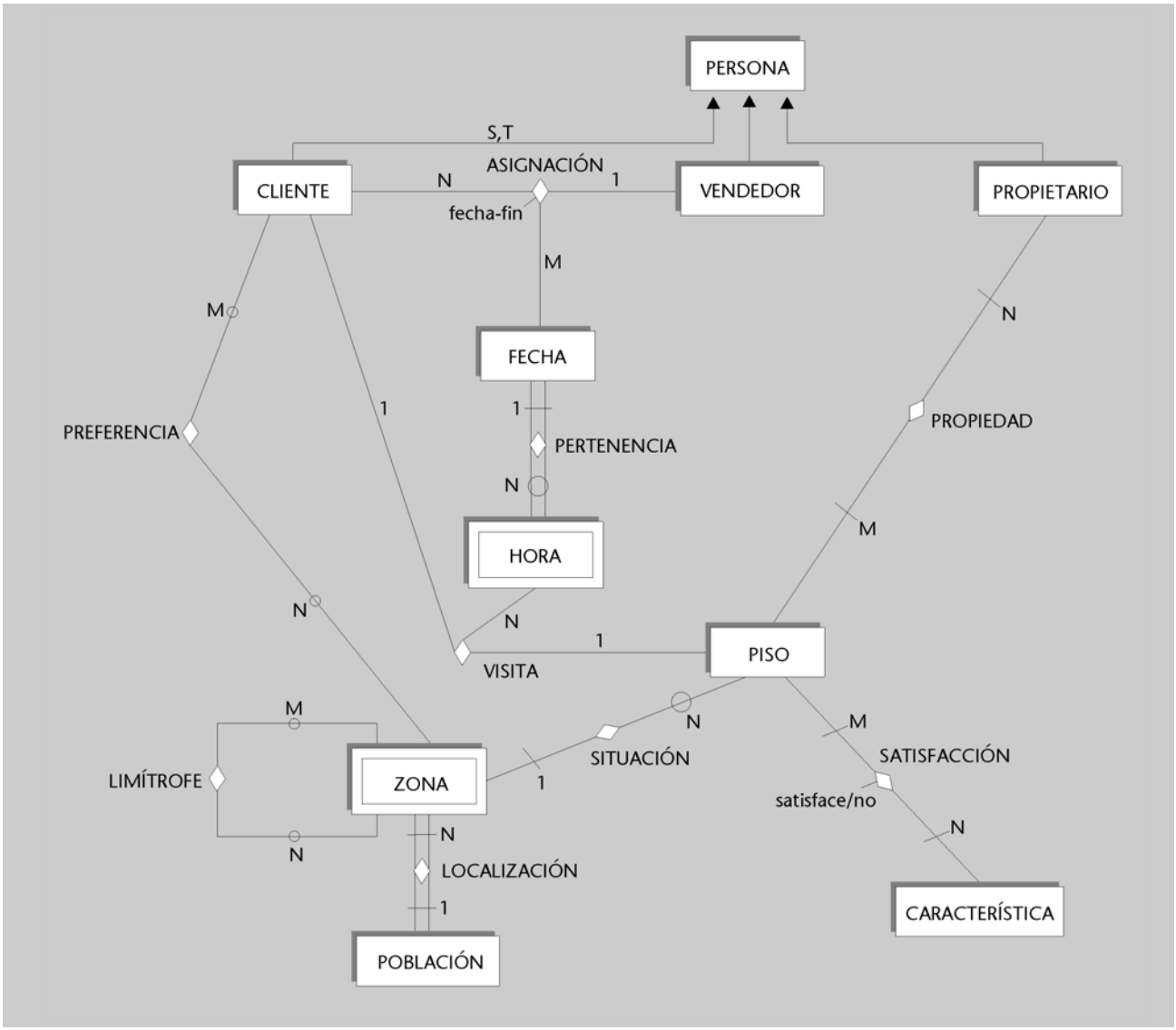
2. La siguiente figura muestra un diagrama ER que satisface los requisitos que se han descrito:



Los atributos de las entidades que figuran en el diagrama son los siguientes (las claves primarias se han subrayado):

CIUDAD
    nombre-ciudad
DELEGACIÓN
    nombre-del, teléfono
EMPLEADO
    código-empleado, nombre, teléfono, año-nacimiento
CONDUCTOR (subclase de empleado)
    código-empleado, año-carnet, tipo-carnet
ADMINISTRATIVO (subclase de empleado)
    código-empleado, nivel-estudios
FECHA
    fecha
VIAJE (entidad débil: código-viaje la identifica parcialmente, se identifica completamente con la delegación de organización).
    código-viaje
CAMION
    código-camion, matrícula, marca, tara
CLIENTE
    código-cliente, nombre, teléfono-contacto

3. La figura que podéis ver a continuación muestra un diagrama ER que satisface los requisitos que se han descrito:



Los atributos de las entidades que figuran en el diagrama son los siguientes (las claves primarias se han subrayado):

POBLACIÓN
nombre-pobl, número-hab
ZONA (entidad débil: nombre-zona la identifica parcialmente, se identifica completamente con la población de localización)
nombre-zona
PISO
código-piso, dirección, superficie, número-habitaciones, precio
CARACTERISTICA
código-car, descripción
PERSONA
código-persona, nombre, dirección, teléfono
VENDEDOR (entidad subclase de persona)
código-persona, nss, sueldo
CLIENTE (entidad subclase de persona)
código-persona
PROPIETARIO (entidad subclase de persona)
código-persona, nif
FECHA
fecha
HORA (entidad débil: hora-minuto la identifica parcialmente, se identifica completamente con la fecha de pertenencia)
hora-minuto

4. El resultado de la transformación a relacional del modelo ER propuesto como solución del ejercicio 1 consta de las siguientes relaciones:

COMPETICIÓN(nombre-comp)
TEMPORADA(código-temp)
FEDERACIÓN(nombre-fed, fecha-creación)
CLUB(nombre-club, número-socios, nombre-fed)
donde {nombre-fed} referencia FEDERACIÓN
PERSONA(código-persona, nombre, dirección, teléfono, fecha-nacimiento)
JUGADOR(código-persona, peso, altura)
donde {código-persona} referencia PERSONA
ENTRENADOR(código-persona, fecha-inicio-carrera)
donde {código-persona} referencia PERSONA
ESPECIALIDAD(nombre-esp)
FECHA(fecha)
HABILIDAD(código-persona, nombre-esp, grado)
donde {código-persona} referencia JUGADOR
y {nombre-esp} referencia ESPECIALIDAD
CLASIFICACIÓN(nombre-comp, código-temp, nombre-club, num-posición)
donde {nombre-comp} referencia COMPETICIÓN,
{código-temp} referencia TEMPORADA
y {nombre-club} referencia CLUB
CONTRATO(código-persona, fecha, nombre-club, importe-cont, fecha-baja)
donde {código-persona} referencia PERSONA,
{fecha} referencia FECHA
y {nombre-club} referencia CLUB
OFERTA(código-persona, fecha, nombre-club, importe-oferta)
donde {código-persona} referencia PERSONA,
{fecha} referencia FECHA
y {nombre-club} referencia CLUB

5. El resultado de la transformación a relacional del modelo ER propuesto como solución del ejercicio 2 consta de las siguientes relaciones:

CIUDAD(nombre-ciudad)
DELEGACIÓN(nombre-del, teléfono, nombre-ciudad)
donde {nombre-ciudad} referencia CIUDAD
EMPLEADO(código-empleado, nombre, teléfono, año-nacimiento)
CONDUCTOR(código-empleado, año-carnet, tipo-carnet)
donde {código-empleado} referencia EMPLEADO
ADMINISTRATIVO (código-empleado, nivel-estudios)
donde {código-empleado} referencia EMPLEADO
FECHA(fecha)
VIAJE(nombre-del, código-viaje, código-camión)
donde {nombre-del} referencia DELEGACIÓN
y {código-camión} referencia CAMIÓN
CAMIÓN(código-camión, matrícula, marca, tara)
CLIENTE(código-cliente, nombre, teléfono-contacto)
ASIGNACION(código-empleado, fecha, nombre-del, fecha-fin)
donde {código-empleado} referencia EMPLEADO,
{nombre-del} referencia DELEGACIÓN
y {fecha} referencia FECHA
CONDUCCIÓN(código-empleado, nombre-del, código-viaje, importe-dietas)
donde {código-empleado} referencia CONDUCTOR
y {nombre-del, código-viaje} referencia VIAJE
RECORRIDO(nombre-ciudad, nombre-del, código-viaje, fecha, hora)
donde {nombre-ciudad} referencia CIUDAD
y {nombre-del, código-viaje} referencia VIAJE
REPARTO (nombre-ciudad, nombre-del, código-viaje, código-cliente,
paquetes-car, paquetes-desc)
donde {nombre-ciudad, nombre-del, código-viaje} referencia RECORRIDO
y {código-cliente} referencia CLIENTE

6. El resultado de la transformación a relacional del modelo ER propuesto como solución del ejercicio 3 consta de las siguientes relaciones:

POBLACIÓN(nombre-pobl, nombre-hab)
ZONA(nombre-pobl, nom-zona)
  donde {nombre-pobl} referencia POBLACIÓN
PISO(código-piso, dirección, superficie, número-habitaciones, precio,
    nombre-pobl, nombre-zona)
  donde {nombre-pobl, nombre-zona} referencia ZONA
CARACTERÍSTICA(código-car, descripción)
PERSONA(código-persona, nombre, dirección, teléfono)
VENDEDOR(código-persona, nss, sueldo)
  donde {código-persona} referencia PERSONA
CLIENTE(código-persona)
  donde {código-persona} referencia PERSONA
PROPIETARIO(código-persona, nif)
  donde {código-persona} referencia PERSONA
FECHA(fecha)
HORA(fecha, hora-minuto)
  donde {fecha} referencia FECHA
ASIGNACIÓN(código-cliente, fecha, código-vendedor, fecha-fin)
  donde {código-cliente} referencia CLIENTE,
  {fecha} referencia FECHA
  y {código-vendedor} referencia VENDEDOR
PREFERENCIA(código-persona, nombre-pobl, nombre-zona)
  donde {código-personal} referencia CLIENTE
  y {nombre-pobl, nom-zona} referencia ZONA
SATISFACCIÓN(código-piso, código-car, satisface/no)
  donde {código-piso} referencia PISO
  y {código-car} referencia CARACTERÍSTICA
LIMÍTROFE(nombre-pobl, nombre-zona, nombre-pobl-lim, nombre-zona-lim)
  donde {nombre-pobl, nombre-zona} referencia ZONA
  y {nombre-pobl-lim, nombre-zona-lim} referencia ZONA
PROPIEDAD(código-piso, código-persona)
  donde {código-piso} referencia PISO
  y {código-persona} referencia PROPIETARIO

Para la interrelación *visita*, hay dos transformaciones posibles:

1)

VISITA(fecha, hora-minuto, código-piso, código-persona)
  donde {fecha, hora-minuto} referencia HORA,
  {código-persona} referencia CLIENTE
  y {código-piso} referencia PISO

2)

VISITA(fecha, hora-minuto, código-persona, código-piso)
  donde {fecha, hora-minuto} referencia HORA,
  {código-persona} referencia CLIENTE
  y {código-piso} referencia PISO

Glosario

atributo de una entidad
Propiedad que interesa de una entidad.

atributo de una interrelación
Propiedad que interesa de una interrelación.

conectividad de una interrelación
Expresión del tipo de correspondencia entre las ocurrencias de entidades asociadas con la interrelación.

diseño conceptual
Etapa del diseño de una base de datos que obtiene una estructura de la información de la futura BD independiente de la tecnología que se quiere utilizar.

diseño físico
Etapa del diseño de una base de datos que transforma la estructura obtenida en la etapa del diseño lógico con el objetivo de conseguir una mayor eficiencia y que, además, la completa con aspectos de implementación física que dependerán del SGBD que se debe utilizar.

diseño lógico
Etapa del diseño de una base de datos que parte del resultado del diseño conceptual y lo transforma de modo que se adapte al modelo del SGBD con el que se desea implementar la base de datos.

entidad
Objeto del mundo real que podemos distinguir del resto de los objetos y del cual nos interesan algunas propiedades.

entidad asociativa
Entidad resultante de considerar una interrelación entre entidades como una nueva entidad.

entidad débil
Entidad cuyos atributos no la identifican completamente, sino que sólo la identifican de forma parcial.

entidad obligatoria en una interrelación binaria
Entidad tal que una ocurrencia de la otra entidad que interviene en la interrelación sólo puede existir si se da como mínimo una ocurrencia de la entidad obligatoria a la que está asociada.

entidad opcional en una interrelación binaria
Entidad tal que una ocurrencia de la otra entidad que interviene en la interrelación puede existir aunque no haya ninguna ocurrencia de la entidad opcional a la que está asociada.

generalización/especialización
Construcción que permite reflejar que existe una entidad general, denominada *entidad superclase*, que se puede especializar en entidades subclase. La entidad superclase nos permite modelizar las características comunes de la entidad vista a un nivel genérico, y con las entidades subclase podemos modelizar las características propias de sus especializaciones.

grado de una interrelación
Número de entidades que asocia la interrelación.

interrelación
Asociación entre entidades.

interrelación recursiva
Interrelación en la que alguna entidad está asociada más de una vez.

Bibliografía

Batini, C.; Ceri, S.; Navathe, S.B. (1992). *Conceptual Database Design: An Entity-Relationship Approach*. Reading, Massachusetts: Addison Wesley.

Teorey, T.J. (1999). *Database Modeling & Design. The Fundamental Principles* (3ª ed.). San Francisco: Morgan Kaufmann Publishers, Inc.

Bases de datos en MySQL

Luis Alberto Casillas Santillán
Marc Gibert Ginestà
Óscar Pérez Mora

UOC
Formación de Posgrado

Índice

Introducción

MySQL es un sistema gestor de bases de datos (SGBD, DBMS por sus siglas en inglés) muy conocido y ampliamente usado por su simplicidad y notable rendimiento. Aunque carece de algunas características avanzadas disponibles en otros SGBD del mercado, es una opción atractiva tanto para aplicaciones comerciales, como de entretenimiento precisamente por su facilidad de uso y tiempo reducido de puesta en marcha. Esto y su libre distribución en Internet bajo licencia GPL le otorgan como beneficios adicionales (no menos importantes) contar con un alto grado de estabilidad y un rápido desarrollo.

Nota

Podremos utilizar la licencia GPL de MySQL siempre que el programa que lo use también lo sea, en caso contrario se debe adquirir la "licencia comercial", entre 250 y 500 €, en el momento de escribir este material.

MySQL está disponible para múltiples plataformas, la seleccionada para los ejemplos de este libro es GNU/Linux. Sin embargo, las diferencias con cualquier otra plataforma son prácticamente nulas, ya que la herramienta utilizada en este caso es el cliente *mysql-client,* que permite interactuar con un servidor MySQL (local o remoto) en modo texto. De este modo es posible realizar todos los ejercicios sobre un servidor instalado localmente o, a través de Internet, sobre un servidor remoto.

Para la realización de todas las actividades, es imprescindible que dispongamos de los datos de acceso del usuario administrador de la base de datos. Aunque en algunos de ellos los privilegios necesarios serán menores, para los capítulos que tratan la administración del SGBD será imprescindible disponer de las credenciales de administrador.

Nota

Las sentencias o comandos escritos por el usuario estarán en `fuente monoespaciada`, y las palabras que tienen un significado especial en MySQL estarán en **negrita**. Es importante hacer notar que estas últimas no siempre son palabras reservadas, sino comandos o sentencias de *mysql-client*.

La versión de MySQL que se ha utilizado durante la redacción de este material, y en los ejemplos, es la 4.1, la última versión estable en ese momento, aunque no habrá ningún problema en ejecutarlos en versiones anteriores, hasta la 3.23.

Objetivos

Adquirir las habilidades y conocimientos de MySQL necesarios para utilizar y administrar este SGBD (sistema gestor de bases de datos).

1. Características de MySQL

En este apartado enumeraremos las prestaciones que caracterizan a este SGBD, así como las deficiencias de diseño, limitaciones o partes del estándar aún no implementadas.

1.1. Prestaciones

MySQL es un SGBD que ha ganado popularidad por una serie de atractivas características:

- Está desarrollado en C/C++.
- Se distribuyen ejecutables para cerca de diecinueve plataformas diferentes.
- La API se encuentra disponible en C, C++, Eiffel , Java, Perl, PHP, Python, Ruby y TCL.
- Está optimizado para equipos de múltiples procesadores.
- Es muy destacable su velocidad de respuesta.
- Se puede utilizar como cliente-servidor o incrustado en aplicaciones.
- Cuenta con un rico conjunto de tipos de datos.
- Soporta múltiples métodos de almacenamiento de las tablas, con prestaciones y rendimiento diferentes para poder optimizar el SGBD a cada caso concreto.
- Su administración se basa en usuarios y privilegios.
- Se tiene constancia de casos en los que maneja cincuenta millones de registros, sesenta mil tablas y cinco millones de columnas.
- Sus opciones de conectividad abarcan TCP/IP, *sockets* UNIX y *sockets* NT, además de soportar completamente ODBC.
- Los mensajes de error pueden estar en español y hacer ordenaciones correctas con palabras acentuadas o con la letra 'ñ'.
- Es altamente confiable en cuanto a estabilidad se refiere.

Para todos aquellos que son adeptos a la filosofía de UNIX y del lenguaje C/C++, el uso de MySQL les será muy familiar, ya que su diseño y sus interfaces son acordes a esa filosofía: "crear herramientas que hagan una sola cosa y que la hagan bien". MySQL tiene como principal objetivo ser una base de datos fiable y eficiente. Ninguna característica es implementada en MySQL si antes no se tiene la certeza que funcionará con la mejor velocidad de respuesta y, por supuesto, sin causar problemas de estabilidad.

La influencia de C/C++ y UNIX se puede observar de igual manera en su sintaxis. Por ejemplo, la utilización de expresiones regulares, la diferenciación de funciones por los paréntesis, los valores lógicos como 0 y 1, la utilización del tabulador para completar sentencias, por mencionar algunos.

1.2. Limitaciones

Al comprender sus principios de diseño, se puede explicar mejor las razones de algunas de sus carencias. Por ejemplo, el soporte de transacciones o la integridad referencial (la gestión de claves foráneas) en MySQL está condicionado a un esquema de almacenamiento de tabla concreto, de forma que si el usuario no va a usar transacciones, puede usar el esquema de almacenamiento "tradicional" (MyISAM) y obtendrá mayor rendimiento, mientras que si su aplicación requiere transacciones, deberá usar el esquema que lo permite (InnoDB), sin ninguna otra restricción o implicación.

Nota

El esquema de tabla que hay que usar se decide para cada una en el momento de su creación, aunque puede cambiarse posteriormente. Actualmente, MySQL soporta varios esquemas y permite la incorporación de esquemas definidos por el usuario.

Otras limitaciones son las siguientes:

- No soporta procedimientos almacenados (se incluirán en la próxima versión 5.0).
- No incluye disparadores (se incluirán en la próxima versión 5.0).
- No incluye vistas (se incluirán en la próxima versión 5.0).
- No incluye características de objetos como tipos de datos estructurados definidos por el usuario, herencia etc.

2. Acceso a un servidor MySQL

En este apartado veremos las distintas formas de acceso a un servidor MySQL existente que nos proporciona el propio SGBD. El acceso desde lenguajes de programación o herramientas en modo gráfico se tratará en otros apartados.

2.1. Conectándose con el servidor

Para conectarse con el servidor deberemos asegurarnos de que éste está funcionando y de que admite conexiones, sean éstas locales (el SGBD se está ejecutando en la misma máquina que intenta la conexión) o remotas.

Adicionalmente, deberemos disponer de las credenciales necesarias para la conexión. Distintos tipos de credenciales nos permitirán distintos niveles de acceso. Para simplificar, supondremos que disponemos de las credenciales (usuario y contraseña) del administrador de la base de datos (normalmente, usuario *root* y su contraseña). En el apartado que concierne a la administración de MySQL, se comenta detalladamente los aspectos relacionados con el sistema de usuarios, contraseñas y privilegios del SGBD.

2.1.1. Servidores y clientes

El servidor MySQL es el servicio *mysqld*, que puede recibir solicitudes de clientes locales o remotos a través TCP/IP, *sockets* o *pipes* en forma de ficheros locales a la máquina en que se está ejecutando. En la distribución se incluye un cliente llamado *mysql-client*, al que en adelante nos referiremos simplemente como *mysql* (así es como se llama el programa ejecutable). Si se invoca sin parámetros, *mysql* realiza una conexión al servidor local utilizando el nombre del usuario UNIX que lo ha invocado, y supone que este usuario no requiere contraseña. La conexión a un servidor remoto y un nombre de usuario específicos requiere de al menos dos argumentos:

Nota

El servidor MySQL es *mysqld*. A él se pueden conectar múltiples clientes. *mysql* es el cliente en modo texto que proporciona el propio SGBD.

- -h para especificar el nombre del servidor.
- -u para el nombre del usuario.

Para que el programa cliente pregunte la contraseña de conexión al usuario, deberemos proporcionar adicionalmente el parámetro -p.

```
$ mysql -h servidor.misitio.org -u <usuario> -p
```

2.1.2. Conectarse y desconectarse

Si se tiene algún problema para realizar la conexión, es necesario consultar con el administrador del sistema, que nos proporcionará un nombre de usuario, contraseña y el nombre del servidor, según sea necesario, y nos informará de las restricciones que tiene nuestra cuenta.

Nota

Los usuarios del sistema operativo y los de MySQL no son los mismos, aunque el administrador de MySQL (con fines prácticos) pueda utilizar los mismos nombres para las cuentas de los usuarios MySQL.

La administración y seguridad de MySQL está diseñada sobre un esquema de usuarios y privilegios. Los usuarios deben ser creados por el administrador con sus respectivos privilegios y restricciones. Es el administrador quien decide si los nombres de los usuarios de MySQL se corresponden o no a los del sistema operativo.

Apariencia de *mysql* al ingresar en el modo interactivo:

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 5 to server version: 3.23.49-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Con el comando *help* obtenemos una serie de opciones (veremos las más utilizadas).

Para salir del cliente podemos escribir '*q*' o '*quit*':

```
mysql> quit;
```

Tanto para el comando *quit* como para el comando *help*, el punto y coma al final es opcional.

2.2. Introducción de sentencias

El cliente de MySQL en modo interactivo nos permite tanto la introducción de sentencias SQL para trabajar con la base de datos (crear tablas, hacer consultas y ver sus resultados, etc.) como la ejecución de comandos propios del SGBD para obtener información sobre las tablas, índices, etc. o ejecutar operaciones de administración.

Sentencias

Las sentencias en *mysql* pueden abarcar múltiples líneas y terminan con punto y coma.

2.2.1. Sentencias

A continuación presentamos una ejecución de la sentencia *select* con cuatro columnas de datos:

```
mysql> select user(), connection_id(), version(), database();
+--------------+-----------------+-------------+------------+
|   user()     | CONNECTION_ID() | version()   | database()  |
+--------------+-----------------+-------------+------------+
| yo@localhost |    4            | 3.23.49-log |            |
+--------------+-----------------+-------------+------------+
1 row in set (0.00 sec)
mysql>
```

En esta consulta se solicita, a través de funciones incorporadas en el SGBD, el nombre del usuario actual de MySQL, el número de conexión al servidor, la versión del servidor y la base de datos en uso. Las funciones se reconocen por los paréntesis al final. *mysql* entrega sus resultados en tablas, en la que el primer renglón son los encabezados de las columnas. Es importante no dejar espacio entre el nombre de una función y los paréntesis, de otro modo, *mysql* marcará un mensaje de error.

La última línea entregada por *mysql* informa sobre el número de filas encontrado como resultado de la consulta y el tiempo estimado que llevó su realización. Esta medida de tiempo no se debe considerar muy precisa para medir el rendimiento del servidor, se trata simplemente de un valor aproximado que puede verse alterado por múltiples factores.

Observamos que la columna con el nombre de la base de datos actual esta vacía. Esto es natural, ya que no hemos creado aún ninguna base de datos ni le hemos indicado al gestor sobre cuál queremos trabajar.

2.2.2. Comandos en múltiples líneas

Los comandos pueden expandirse en varias líneas por comodidad, sobre todo al escribir largas sentencias SQL. El cliente no enviará la sentencia SQL al servidor hasta encontrar el punto y coma, de este modo, el comando anterior puede escribirse así:

```
mysql> select user(),
-> connection_id(),
-> version(),
-> database();
+--------------+-----------------+-------------+------------+
|   user()     | CONNECTION_ID() | version()   | database()  |
+--------------+-----------------+-------------+------------+
| yo@localhost |    4            | 3.23.49-log |            |
+--------------+-----------------+-------------+------------+
1 row in set (0.00 sec)
mysql>
```

Obsérvese el indicador de *mysql* que se transforma en ->, signo que significa que el comando aún no está completo. También pueden escribirse varios comandos en una sola línea, cada uno debe llevar su respectivo punto y coma:

```
mysql> select now(); select user();
+---------------------+
| CONNECTION_ID()     |
+---------------------+
|    4                |
+---------------------+
1 row in set (0.00 sec)
+---------------------+
|   user()            |
+---------------------+
|  yo@localhost       |
+---------------------+
1 row in set (0.01 sec)
mysql>
```

Se ejecutarán en el orden que están escritos. Los comandos se pueden cancelar con la combinación \c, con lo que el cliente nos volverá a mostrar el indicador para que escribamos de nuevo la sentencia.

```
mysql> select now(),
-> uso
-> ver \c
mysql>
```

Indicadores de *mysql*

Indicador	Significado
mysql>	Espera una nueva sentencia
->	La sentencia aún no se ha terminado con ;
">	Una cadena en comillas dobles no se ha cerrado
'>	Una cadena en comillas simples no se ha cerrado

2.2.3. Cadenas de caracteres

Las cadenas de caracteres pueden delimitarse mediante comillas dobles o simples. Evidentemente, deben cerrarse con el mismo delimitador con el que se han abierto.

```
mysql> select "Hola mundo",'Felicidades';
```

y pueden escribirse en diversas líneas:

```
mysql> select "Éste es un texto
      "> en dos renglones";
```

Al principio, es común olvidar el punto y coma al introducir un comando y, también, olvidar cerrar las comillas. Si éste es el caso, hay que recordar que *mysql* no interpreta lo que está entre comillas, de tal modo que para utilizar el comando de cancelación '\c' es preciso antes cerrar las comillas abiertas:

```
mysql> select "Éste es un texto
"> \c
"> " \c
mysql>
```

2.2.4. Expresiones y variables

MySQL dispone de variables de sesión, visibles únicamente durante la conexión actual. Éstas pueden almacenar valores de tipos enteros, flotantes o cadenas, pero no tablas. Se definen como en el siguiente ejemplo:

```
mysql> select @x := 1;
```

La variable local @x tiene ahora el valor 1 y puede utilizarse en expresiones:

```
mysql> select @x, sqrt(@x), sin(@x), @x + 10, @x > 10;
+------+----------+----------+---------+---------+
| @x   | sqrt(@x) | sin(@x)  | @x + 10 | @x > 10 |
+------+----------+----------+---------+---------+
| 1    | 1.000000 | 0.841471 |  11     |  0      |
+------+----------+----------+---------+---------+
1 row in set (0.01 sec)
```

Las variables locales permiten almacenar datos entre consultas y, en la práctica, es recomendable utilizarlas exclusivamente con este fin, por ejemplo:

```
mysql> select @hora_ingreso := now();
mysql> mysql> select now() - @ingreso;
+------------------+
| now() - @ingreso  |
+------------------+
|  20040124138051   |
+------------------+
1 row in set (0.00 sec)
```

2.2.5. Expresiones

Hay que tener cuidado con el uso de las variables locales por los motivos siguientes:

- Se evalúan en el servidor al ser enviadas por el cliente.
- Se realizan conversiones de tipo implícitas.

```
mysql> do @ingreso := now();
```

Nota

El comando **do** evalúa expresiones sin mostrar los resultados en pantalla. Se puede evaluar cualquier expresión que admite el comando **select**.

Las variables no requieren declaración y, por omisión, contienen el valor NULL que significa "ausencia de valor", observad en la siguiente consulta los resultados de utilizar valores nulos:

```
mysql> select @y,
-> sqrt( @y ),
-> @y + 10,
-> @y < 1 ;
+------+------------+---------+--------+
| @y   | sqrt( @y ) | @y + 10 | @y < 1 |
+------+------------+---------+--------+
| NULL |  NULL      |  NULL   | NULL   |
+------+------------+---------+--------+
1 row in set (0.00 sec)
```

La razón de este comportamiento es que no es posible realizar ninguna operación cuando se desconoce algún valor. La entrada de valores NULL siempre significará salida de valores NULL.

Nota

En una expresión donde cualquiera de sus elementos sea NULL, automáticamente entregará como resultado el valor NULL.

2.3. Proceso por lotes

MySQL puede procesar por lotes las sentencias contenidas en un archivo de texto. Cada sentencia deberá terminar en ';' igual que si la escribiéramos en el cliente. La sintaxis es la siguiente:

```
$ mysql -u juan -h servidor.misitio.org -p < demo.sql
```

En este caso, se realizará una conexión con el servidor, nos pedirá la contraseña del usuario 'juan' y, si ésta es correcta, ejecutará los comandos incluidos en el archivo *demo.sql*, uno a uno y por el mismo orden. Imprimirá los resultados (o errores) en la salida estándar (o de error) y terminará. De este modo evitaremos la molestia de procesarlos uno por uno de forma interactiva.

Otra forma de procesar un archivo es mediante el comando `source` desde el indicador interactivo de MySQL:

```
mysql> source demo.sql
```

El archivo *demo.sql* crea una nueva base de datos.

El usuario debe tener permisos para crear bases de datos si quiere que sea procesado el archivo *demo.sql*. Si el administrador crea la base de datos por nosotros, será necesario editarlo, comentando la línea donde se crea la base de datos con el símbolo '#' al inicio:

```
# create database demo;
```

Es necesario procesar el contenido del fichero *demo.sql* tal como los transcribimos aquí, con el fin de poder realizar los ejemplos del resto del apartado. Si se observa su contenido, posiblemente muchas cosas se expliquen por sí mismas, de cualquier manera, serán explicadas en este apartado. También pueden ejecutarse sus órdenes en el cliente directamente.

Contenido del fichero demo.sql

```
#drop database demo;
create database demo;
use demo;
---
--- Estructura de la tabla productos
---

create table productos (
 parte    varchar(20),
 tipo     varchar(20) ,
 especificación varchar(20) ,
 psugerido   float(6,2),
 clave int(3)  zerofill not null auto_increment,
 primary key (clave)
);
insert into productos (parte,tipo,especificación,psugerido) values
    ('Procesador','2 GHz','32 bits',null),
    ('Procesador','2.4 GHz','32 bits',35),
    ('Procesador','1.7 GHz','64 bits',205),
    ('Procesador','3 GHz','64 bits',560),
    ('RAM','128MB','333 MHz',10),
    ('RAM','256MB','400 MHz',35),
    ('Disco Duro','80 GB','7200 rpm',60),
```

```
    ('Disco Duro','120 GB','7200 rpm',78),
    ('Disco Duro','200 GB','7200 rpm',110),
    ('Disco Duro','40 GB','4200 rpm',null),
    ('Monitor','1024x876','75 Hz',80),
    ('Monitor','1024x876','60 Hz',67)
    ;
--
-- Estructura de la tabla 'proveedor'
--
create table proveedores (
 empresa   varchar(20) not null,
 pago    set('crédito','efectivo'),
 primary key (empresa)
);
--
-- Valores de la tabla 'proveedor'
--
insert into proveedores (empresa,pago) values
   ('Tecno-k','crédito'),
   ('Patito','efectivo'),
   ('Nacional','crédito,efectivo')
;
create table ganancia(
  venta  enum('Por mayor','Por menor'),
   factor  decimal(2,2)
);
insert into ganancia values
  ('Por mayor',1.05),
    ('Por menor',1.12)
;
create table precios (
   empresa  varchar(20) not null,
   clave  int(3) zerofill not null,
   precio  float(6,2),

   foreign key (empresa)  references proveedores,
   foreign key (clave)   references productos
);

insert into precios values
   ('Nacional',001,30.82),
   ('Nacional',002,32.73),
   ('Nacional',003,202.25),
   ('Nacional',005,9.76),
   ('Nacional',006,31.52),
   ('Nacional',007,58.41),
   ('Nacional',010,64.38),
   ('Patito',001,30.40),
   ('Patito',002,33.63),
   ('Patito',003,195.59),
   ('Patito',005,9.78),
   ('Patito',006,32.44),
```

```
   ('Patito',007,59.99),
   ('Patito',010,62.02),
   ('Tecno-k',003,198.34),
   ('Tecno-k',005,9.27),
   ('Tecno-k',006,34.85),
   ('Tecno-k',007,59.95),
   ('Tecno-k',010,61.22),
   ('Tecno-k',012,62.29)
;
```

Si se desea llevar un registro de todas las operaciones de una sesión, se puede utilizar la expresión siguiente; de este modo se guardarán todos los comandos y sus resultados en *archivo_registro.txt*:

```
mysql> tee archivo_registro.txt
```

Para cancelar la captura, basta con teclear lo siguiente:

```
mysql> notee
```

2.4. Usar bases de datos

La siguiente consulta informa sobre la base de datos actualmente en uso.

```
mysql> select database();
+------------+
| database() |
+------------+
|            |
+------------+
1 row in set (0.13 sec)
```

El campo esta vacío porque no estamos haciendo uso de ninguna base de datos. Para ver las bases de datos existentes en el sistema, se debe efectuar la siguiente consulta:

```
mysql> show databases;
+-------------+
|  Database   |
+-------------+
|   demo      |
|  mysql      |
|   test      |
+-------------+
3 rows in set (0.01 sec)
```

MySQL nos muestra el listado de las bases de datos definidas en el servidor. Debe aparecer la base de datos *demo* que creamos con el archivo *demo.sql*. Para poder trabajar con ella, tenemos que abrirla:

```
mysql> use demo;
```

use

El comando *use base_de_datos* permite abrir una base de datos para su uso.

Nota

Es posible realizar consultas en una base de datos sin utilizar el comando **use**, en ese caso, todos los nombres de las tablas deben llevar el nombre de la base de datos a que pertenecen de la forma: *demo.productos*.

Otra posibilidad consiste en proporcionar el nombre de la base de datos al iniciar una sesión interactiva con *mysql:*

```
$ mysql demo -u juan -p
```

La consulta de las tablas que contiene la base de datos `demo` se realiza con la sentencia `show` de la siguiente manera:

```
mysql> show tables;
+----------------+
| Tables_in_demo |
+----------------+
|   partes       |
|  proveedores   |
+----------------+
2 rows in set (0.00 sec)
```

Nota

El comando *show* es útil para mostrar información sobre las bases de datos, tablas, variables y otra información sobre el SGBD. Podemos utilizar `help show` en el intérprete de comandos para obtener todas las variantes de esta sentencia.

Asimismo, podemos consultar las columnas de cada una de las tablas:

```
mysql> describe productos;
+---------------+----------------------+------+------+----------+-------------------+
| Field         |       Type           | Null | Key  | Default  | Extra             |
+---------------+----------------------+------+------+----------+-------------------+
| parte         | varchar(20)          | YES  |      | NULL     |                   |
| tipo          | varchar(20)          | YES  |      | NULL     |                   |
| especificicación | varchar(20)          | YES  |      | NULL     |                   |
| Field         | float(6,2)           | YES  |      | NULL     |                   |
| Field         | int(3) unsigned zerofill | YES  | PRI  | NULL     | auto_increment    |
+---------------+----------------------+------+------+----------+-------------------+
5 rows in set (0.00 sec)
```

Para crear una nueva base de datos usaremos la sentencia `create database`:

```
mysql> create database prueba;
```

Para eliminar una base de datos, usaremos la sentencia `drop database`:

```
mysql> drop database prueba;
```

MySQL es sensible al uso de mayúsculas y minúsculas, tanto en la definición de bases de datos, como de tablas o columnas.

3. Creación y manipulación de tablas

3.1. Crear tablas

Una vez realizada la conexión con el servidor MySQL y después de abrir una base de datos, podemos crear tablas en ella de la siguiente manera:

```
mysql> create table personas (
-> nombre char(30),
-> dirección char(40),
-> teléfono char(15)
-> );
Query OK, 0 rows affected (0.02 sec)
```

En este caso, la sentencia `create table` construye una nueva tabla en la base de datos en uso. La tabla contiene tres columnas, *nombre*, *dirección* y *teléfono*, todas de tipo carácter y de longitudes 30, 40 y 15 respectivamente. Si se intenta guardar en ellas valores que sobrepasen esos límites, serán truncados para poderlos almacenar. Por ese motivo, es importante reservar espacio suficiente para cada columna. Si se prevé que muchos registros ocuparán sólo una fracción del espacio reservado, se puede utilizar el tipo **varchar**, similar a **char**, con la diferencia de que el valor ocupará un espacio menor al especificado si la cadena es más corta que el máximo indicado, ahorrando así espacio de almacenamiento.

Los nombres de las columnas admiten caracteres acentuados. 

Las tablas pueden eliminarse con **drop table**:

```
mysql> drop table personas;
Query OK, 0 rows affected (0.01 sec)
```

Alternativamente, se puede utilizar la sintaxis siguiente:

```
mysql> drop table if exists personas;
```

Atributos de columna

Atributo	Significado
null	Se permiten valores nulos, atributo por omisión si no se especifica lo contrario.
not null	No se permiten valores nulos.
default valor	Valor por omisión que se asigna a la columna.
auto_increment	El valor se asigna automáticamente incrementando en uno el máximo valor registrado hasta ahora. Se aplica sólo a las columnas marcadas como clave primaria.
primary key	Señala al campo como clave primaria, implícitamente también lo declara como **not null**.

Veámoslo con un ejemplo:

```
mysql> create table personas (
-> nombre varchar(40) not null,
-> dirección varchar(50) null,
-> edo_civil char(13) default 'Soltero',
-> num_registro int primary key auto_increment,
-> );
Query OK, 0 rows affected (0.01 sec)
```

En este caso la tabla contiene cuatro columnas, de las cuales *nombre* y *edo_civil* permiten valores nulos, en *edo_civil* está implícito al no declarar lo contrario. La columna *num_registro* no acepta valores nulos porque está definida como clave primaria.

Nota

La definición de columnas tiene el siguiente formato:

nombre_columna **tipo atributos**.

Aunque la creación de una clave primaria puede declararse como atributo de columna, es conveniente definirla como restricción de tabla, como se verá enseguida.

También es posible indicar restricciones sobre la tabla y no sobre columnas especificas:

```
mysql> create table personas (
-> nombre varchar(40) not null,
-> nacimiento date not null,
-> pareja varchar(40),
-> proveedor int not null,
->
-> primary key (nombre,nacimiento),
-> unique (pareja),
-> foreign key (proveedor) references proveedores
-> );
Query OK, 0 rows affected (0.01 sec)
```

Restricciones de tabla

Restricción	Significado
primary key	Define la o las columnas que servirán como clave primaria. Las columnas que forman parte de la clave primaria deben de ser **not null**.
unique	Define las columnas en las que no pueden duplicarse valores. Serán las claves candidatas del modelo relacional.
foreign key (*columna*) **references** ***tabla*** (*columna2*)	Define que los valores de *columna* se permitirán sólo si existen en *tabla(columna2).* Es decir, *columna* hace referencia a los registros de *tabla*, esto asegura que no se realicen referencias a registros que no existen.

Se definen tres restricciones sobre la tabla después de la definición de cuatro columnas:

- La primera restricción se refiere a la clave primaria, compuesta por las columnas *nombre* y *nacimiento*: no puede haber dos personas que se llamen igual y que hayan nacido en la misma fecha. La clave primaria permite identificar de manera unívoca cada registro de la tabla.
- La segunda restricción define que la pareja de una persona debe ser única: dos personas no pueden tener la misma pareja. Todo intento de insertar un nuevo registro donde el nombre de la pareja ya exista, será rechazado. Cuando se restringe una columna con **unique**, los valores **null** reciben un trato especial, pues se permiten múltiples valores nulos.
- La tercera restricción afecta a la columna *proveedor*, sólo puede tomar valores que existan en la clave primaria de la tabla *proveedores*.

Claves foráneas

Las restricciones de tabla **foreign key** no tienen efecto alguno en MySQL 4.0 y anteriores, ya que esta característica no está implementada. Se admite en la sintaxis por compatibilidad, ya que será implementada en una versión posterior. En la versión 4.1, está soportada si se utiliza el tipo de tabla InnoDB.

Las restricciones de tabla pueden definirse con un identificador útil para hacer referencias posteriores a la restricción:

```
mysql> create table personas (
-> nombre varchar(40) not null,
-> nacimiento date not null,
-> pareja varchar(40),
-> proveedor int not null,
->
-> constraint clave primary key (nombre,nacimiento),
-> constraint monogamo unique (pareja),
-> constraint trabaja_en foreign key (proveedor) references
proveedores
-> );
```

key / index

La definición de índices puede hacerse también en el momento de creación de la tabla, mediante la palabra clave key (o index), a la que deberemos proporcionar el nombre que vamos a asignar a esta clave y las columnas que la forman, entre paréntesis. Existen modificadores opcionales sobre el índice que nos permiten especificar si se trata de un índice único o múltiple (según puedan existir o no varios valores iguales del índice en la tabla).

En versiones recientes de MySQL existen otros tipos de índices (espaciales, de texto completo, etc.) para tipos de datos concretos y que ofrecen prestaciones adicionales.

3.2. Tipos de datos

MySQL cuenta con un rico conjunto de tipos de datos para las columnas, que es necesario conocer para elegir mejor cómo definir las tablas. Los tipos de datos se pueden clasificar en tres grupos:

- Numéricos.
- Cadenas de caracteres
- Fechas y horas

El valor **null** es un caso especial de dato, ya que al significar *ausencia de valor* se aplica a todos los tipos de columna. Los siguientes símbolos se utilizan en la definición y descripción de los tipos de datos en MySQL:

- **M** - El ancho de la columna en número de caracteres.
- **D** - Número de decimales que hay que mostrar.
- **L** - Longitud o tamaño real de una cadena.
- **[ ]** - Lo que se escriba entre ellos es opcional.

3.2.1. Tipos de datos numéricos

Los tipos de datos numéricos comprenden dos categorías, los enteros y los números con punto flotante.

Números enteros

La principal diferencia entre cada uno de los tipos de enteros es su tamaño, que va desde 1 byte de almacenamiento hasta los 8 bytes. Las columnas de tipo entero pueden recibir dos atributos adicionales, que deben especificarse inmediatamente después del nombre del tipo:

- **unsigned**. Indica que el entero no podrá almacenar valores negativos. Es responsabilidad del usuario verificar, en este caso, que los resultados de las restas no sean negativos, porque MySQL los convierte en positivos.

- **zerofill**. Indica que la columna, al ser mostrada, rellenará con ceros a la izquierda los espacios vacíos. Esto de acuerdo al valor especificado por **M** en la declaración del tipo. Una columna con el atributo **zerofill** es al mismo tiempo **unsigned** aunque no se especifique.

Ejemplo

```
create table números (
x int(4) zerofill not null,
y int(5) unsigned
);
```

El comando anterior crea una tabla con dos columnas. Ambas ocuparán un espacio de 4 bytes, pero al mostrarse, la columna *x* ocupará un espacio de 4 dígitos y la columna *y*, de 5.

Tanto **zerofill** como **unsigned** deben escribirse siempre antes que cualquier otro atributo de columna.

Tipos enteros

Tipo	Espacio de almacenamiento	Significado
tinyint[(M)]	1 byte	Entero muy pequeño
smallint[(M)]	2 bytes	Entero pequeño
mediumint[(M)]	3 bytes	Entero mediano
int[(M)]	4 bytes	Entero
bigint[(M)]	8 bytes	Entero grande

Números con punto flotante

MySQL cuenta con los tipos **float** y **double**, de 4 y 8 bytes de almacenamiento. Además incluye el tipo decimal, que se almacena como una cadena de caracteres y no en formato binario.

Números de punto flotante

Tipo	Espacio de almacenamiento	Significado
float	4 bytes	Simple precisión
double	8 bytes	Doble precisión
decimal	M + 2 bytes	Cadena de caracteres representando un número flotante

3.2.2. Cadenas de caracteres

Cadenas de caracteres

Tipo	Equivalente	Tamaño máximo	Espacio de almacenamiento
char[(M)]		M bytes	M bytes
varchar[(M)]		M bytes	L+1 bytes
tinytext	tinyblob	2^8–1 bytes	L+1 bytes
text	blob	2^{16}–1 bytes	L+2 bytes
mediumtext	mediumblob	2^{24}–1 bytes	L+3 bytes
longtext	longblob	2^{32}–1 bytes	L+4 bytes
enum('v1','v2',. . .)		65535 valores	1 o 2 bytes
set('v1','v2',. . .)		64 valores	1 a 8 bytes

Si observamos la tabla, vemos que el único tipo de dato que siempre utiliza el tamaño especificado por M es el tipo **char**. Por este motivo, se ofrece el tipo **varchar** que ocupa sólo el espacio requerido por el valor de la columna.

Ejemplo

```
create table persona(
comentario char(250),
recado varchar(250)
);
```

La columna *comentario* ocupará 250 bytes de espacio de almacenamiento, sin importar el valor almacenado. Por el contrario, la columna *recado* ocupará sólo el espacio necesario según el valor asignado; por ejemplo, la cadena "*Instalar MySQL*" tiene 14 bytes de longitud, y el campo *recado* ocuparía 15 bytes para almacenarla.

Los tipos text y blob son equivalentes, pero text respeta las mayúsculas, minúsculas y caracteres acentuados en la ordenación.

Ejemplo del uso de los tipos enumerados o enum

```
create table persona(
edo_civil enum('soltero','casado','viudo','divorciado')
);
```

La columna *edo_civil* de la tabla en la sentencia anterior, solo podrá almacenar los valores '*soltero*', '*casado*', '*viudo*', '*divorciado*', que son especificados por el tipo **enum**. La columna ocupará el espacio de un byte, ya que los valores enum son representados internamente por números.

3.2.3. Fechas y horas

Fechas y horas

Tipo	Espacio de almacenamiento	Rango
date	3 bytes	'1000-01-01' al '9999-12-31'
time	3 bytes	'-838:59:59' a '838:59:59'
datetime	8 bytes	'1000-01-01 00:00:00' a '9999-12-31 23:59:59'
timestamp[(M)]	4 bytes	19700101000000 al año 2037
year[(M)]	1 byte	1901 a 2155

3.3. Modificar tablas

3.3.1. Agregar y eliminar columnas

Alterar la estructura de una tabla es una tarea más frecuente de lo que uno puede imaginar en un principio. La sentencia ***alter table*** permite una amplia gama de formas de modificar una tabla. La siguiente sentencia nos recuerda un poco a la estructura de la sentencia ***create table***, en donde modificamos la tabla *personal* creada en la sección anterior.

```
mysql> alter table personal add (
-> mascota char(30) default 'perro',
-> pasatiempo char (20) not null
-> );
```

Nota

Siempre es posible consultar la estructura de una tabla con el comando ***describe*** *tabla*.

Después de ejecutar la sentencia anterior, aparecen dos nuevas columnas en la tabla. Si queremos agregar una sola columna, podemos usar la sintaxis siguiente:

```
mysql> alter table personal add capital int not null
-> after nom;
```

Este formato de **alter table** permite, además, insertar las columnas antes (**before**) o después (**after**) de una columna en cuestión.

Las columnas no deseadas pueden eliminarse con la opción **drop**.

```
mysql> alter table personal drop pasatiempo;
```

3.3.2. Modificar columnas

La modificación de una columna con la opción **modify** es parecida a volver a definirla.

```
mysql> alter table personal modify
-> mascota char (14) default 'gato';
```

Después de la sentencia anterior, los atributos y tipo de la columna han cambiado por los especificados. Lo que no se puede cambiar con esta sintaxis es el nombre de la columna. Para ello, se debe utilizar la opción **change:**

```
mysql> alter table personal change nom
-> nombre char(20);
```

La columna que se llamaba *nom* cambia a *nombre*.

Con el mismo comando **alter table** podemos incluso realizar la ordenación física de una tabla bajo una columna específica:

```
mysql> alter table personal order by nom;
Query OK, 0 rows affected (0.06 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Nota

En general, una tabla no puede durar mucho tiempo con un `order` respecto a una columna, ya que las inserciones no se realizarán respetando el orden establecido. Solamente en tablas que no van a ser actualizadas es útil aplicar este comando.

Finalmente, podemos cambiar de nombre la tabla:

```
mysql> alter table personal rename gente;
```

rename table

El comando **rename table** *viejo_nombre* to *nuevo_nombre* es una forma alternativa de cambiar el nombre a una tabla.

3.4. Otras opciones

3.4.1. Copiar tablas

Aunque no existe un comando explícito para copiar tablas de una base de datos a otra, es posible utilizar el comando **rename table** para este propósito; basta con especificar la base de datos a la que pertenece una tabla:

```
mysql> rename table base_uno.tabla to base_dos.tabla;
```

También es posible crear una tabla nueva con el contenido de otra ya existente (copiando los datos):

```
mysql> create table nueva_tabla select * from otra_tabla;
```

La siguiente sentencia es equivalente, pero no copia los datos de la tabla origen:

```
mysql> create table nueva_tabla like otra_tabla;
```

3.4.2. Tablas temporales

MySQL permite la creación de tablas temporales, visibles exclusivamente en la sesión abierta, y guardar datos entre consultas. La creación de una tabla temporal sólo requiere la utilización de la palabra ***temporary*** en cualquier formato del comando ***create table***. La utilidad de las tablas temporales se limita a consultas complejas que deben generar resultados intermedios que debemos consultar (hacer 'join' con ellas) varias veces o en consultas separadas. Internamente, MySQL genera también tablas temporales para resolver determinadas consultas:

```
mysql> create temporary table nueva_tabla ...
```

4. Consultas

Como ya hemos explicado, las consultas sobre la base de datos se ejecutan mediante sentencias SELECT introducidas en el propio programa cliente y los resultados se presentan en forma de tabla.

4.1. La base de datos demo

En esta sección utilizaremos la base de datos *demo* que hemos creado con el comando source *demo.sql*. Así que, antes de estudiar las consultas en MySQL, revisaremos brevemente la estructura de esta base de datos, que consta de las siguientes tablas:

Podeís ver la creación de la base de datos *demo* en el apartado "Proceso por lotes" de esta misma unidad didáctica.

```
mysql> show tables;
+----------------+
| Tables_in_demo |
+----------------+
|  ganancia      |
|   precios      |
|  productos     |
|  proveedores   |
+----------------+
```

Las cuatro tablas representan, de manera ficticia, la base de datos de un distribuidor de equipos de procesamiento. Están diseñadas para servir de ejemplo a los casos presentados en este capítulo, por lo que no necesariamente serán útiles en la vida real.

En nuestro ejemplo imaginario representamos la siguiente situación.

- Nuestro vendedor tiene una relación de proveedores que venden sus productos a crédito, en efectivo o ambos. Las compras a crédito pagan intereses, pero son útiles porque no siempre es posible pagar en efectivo. Se utiliza una columna de tipo conjunto para *pago*, que puede tomar los valores '*crédito*', '*efectivo*' o ambos:

```
create table proveedores (
empresa varchar(20) not null,
pago set('crédito','efectivo'),
primary key (empresa)
);
```

Los productos que se distribuyen son partes de equipo de cómputo. Para la mayoría de los productos en el mercado, los fabricantes sugieren un precio de venta al público que, aunque no es obligatorio, los consumidores no están dispuestos a pagar más. Las claves de los productos son asignadas para control

interno con un número consecutivo. Con estas especificaciones, la tabla *productos* se define de la manera siguiente:

```
create table productos (
parte varchar(20),
tipo varchar(20) ,
especificación varchar (20) ,
psugerido float(6,2),
clave int(3) zerofill not null auto_increment,
primary key (clave)
);
```

- La empresa define una política para las ganancias mínimas que se deben obtener en ventas: el 5% al por mayor y el 12% al por menor. Estos valores se almacenan en la tabla *ganancias*, donde se decidió incluir una columna de nombre *factor*, con el número por el que se multiplica el precio de compra para obtener el precio de venta. Los tipos de venta '*Por mayor*' y '*Por menor*' se definen con un tipo de datos **enum**:

```
create table ganancia(
venta enum('Por mayor','Por menor'),
factor decimal (2,2)
);
```

- La lista de precios se define a partir de la empresa proveedor y el producto, asignándole un precio. Por ese motivo, las columnas *empresa* y *clave* se definen como **foreign key**.

```
create table precios (
empresa varchar(20) not null,
clave int(3) zerofill not null,
precio float(6,2),
foreign key (empresa) references proveedores,
foreign key (clave) references productos
);
```

4.2. Consultar información

MySQL ofrece un conjunto muy amplio de funciones auxiliares (tanto estándares como propias) que nos pueden ayudar mucho en determinados momentos, dejando parte del trabajo de manipular los resultados al propio gestor. Debido al rápido ritmo en el desarrollo de este SGBD, es muy conveniente consultar siempre la documentación de nuestra versión para conocer sus posibilidades concretas.

En el módulo 3 de este curso ya estudiamos en detalle el Lenguaje SQL, por lo que no vamos a extendernos aquí en su uso y posibilidades. Únicamente mostraremos los aspectos destacables, facilidades o limitaciones que ofrece MySQL respecto a él.

4.2.1. Funciones auxiliares

Las funciones auxiliares que podemos utilizar en nuestras consultas (tanto en la proyección de las columnas como en condiciones en su selección) se pueden clasificar según el tipo de datos con el que trabajan.

Ejemplo

```
mysql> select concat(parte,' ',tipo) as producto,
-> psugerido as 'precio sugerido',
-> psugerido + 10 as precio_con_envio
-> from productos;
+--------------------+-----------------+------------------+
|   producto         | precio sugerido | precio_con_envio |
+--------------------+-----------------+------------------+
| Procesador 2 GHz   |    NULL         |    NULL          |
| Procesador 2.4 GHz |   35.00         |    45.00         |
| Procesador 1.7 GHz |   205.00        |   215.00         |
| Procesador 3 GHz   |   560.00        |   570.00         |
|     RAM 128MB      |   10.00         |    20.00         |
|     RAM 256MB      |   35.00         |    45.00         |
| Disco Duro 80 GB   |   60.00         |    70.00         |
| Disco Duro 120 GB  |   78.00         |    88.00         |
| Disco Duro 200 GB  |   110.00        |   120.00         |
| Disco Duro 40 GB   |    NULL         |    NULL          |
| Monitor 1024x876   |   80.00         |    90.00         |
| Monitor 1024x876   |   67.00         |    77.00         |
+--------------------+-----------------+------------------+
12 rows in set (0.00 sec)
```

Algunos ejemplos de las funciones más usadas:

Operadores lógicos

Comparación. Aparte de los estándares =, !=, <, >, IS NULL, IS NOT NULL, BETWEN, IN, destacan COALESCE, INTERVAL, LEAST, GREATEST para trabajar con listas de valores.

Control del flujo de ejecución

- CASE .. WHEN .. THEN .. ELSE .. END: Similar a la estructura que crearíamos mediante cualquier lenguaje de programación:

```
mysql> SELECT CASE WHEN 1>0 THEN 'true' ELSE 'false' END;
+-----------------------------------------+
| CASE WHEN 1>0 THEN 'true' ELSE 'false' END |
+-----------------------------------------+
| true                                    |
+-----------------------------------------+
1 row in set (0.00 sec)
```

- IF(expr1,expr2,expr3): Típica estructura condicional, si la expr1 es cierta, devuelve la expr2, en caso contrario, la expr3:

```
mysql> SELECT IF(STRCMP('test','test1'),'no','yes');
+---------------------------------------+
| IF(STRCMP('test','test1'),'no','yes') |
+---------------------------------------+
| no                                    |
+---------------------------------------+
1 row in set (0.00 sec)
```

Funciones para trabajar con cadenas de caracteres (sólo algunos ejemplos)

- CONCAT, INSTR (encontrar en una cadena), SUBSTRING, LCASE/RCASE, LENGTH, REPLACE, TRIM, entre otras, son funciones similares a las que podemos encontrar en lenguajes de programación para manipular cadenas de caracteres.

- QUOTE: delimita una cadena de texto correctamente para evitar problemas al usarla en sentencias SQL. La cadena resultante estará delimitada por comillas simples. Las comillas, el valor ASCII NUL y otros potencialmente conflictivos serán devueltos precedidos del carácter '\'.

- ENCODE/DECODE, CRYPT, COMPRESS/UNCOMPRESS, MD5, etc. son funciones que nos pueden ayudar mucho en el almacenamiento de datos sensibles como contraseñas, etc.

Funciones numéricas

- Los operadores aritméticos clásicos para realizar todo tipo de operaciones, suma, resta, división, producto, división entera, etc.

- Funciones matemáticas de todo tipo, trigonométricas, logarítmicas, etc.

Funciones para trabajar con fechas y horas

- Obtención de fechas en cualquier formato: DATE_FORMAT, DATE, NOW, CURRDATE, etc.

- Manipulación y cálculos con fechas: ADDDATE, ADDTIME, CONVERT_TZ, DATE_DIFF, etc.

4.2.2. La sentencia EXPLAIN

MySQL nos ofrece también facilidades a la hora de evaluar las sentencias SQL, gracias a la sentencia EXPLAIN.

Presentamos primero la ejecución de una sentencia SQL más o menos compleja:

```
mysql> select productos.clave, concat(parte,' ',tipo,' ', especificación) as producto, proveedores.em-
presa , precio , pago from productos natural join precios natural join proveedores;
+-------+----------------------------+----------+--------+------------------+
| clave | producto                   | empresa  | precio | pago             |
+-------+----------------------------+----------+--------+------------------+
|  003  | Procesador 1.7 GHz 64 bits | Tecno-k  | 198.34 | crédito          |
|  005  | RAM 128MB 333 MHz          | Tecno-k  | 9.27   | crédito          |
|  006  | RAM 256MB 400 MHz          | Tecno-k  | 34.85  | crédito          |
|  007  | Disco Duro 80 GB 7200 rpm  | Tecno-k  | 59.95  | crédito          |
|  010  | Disco Duro 40 GB 4200 rpm  | Tecno-k  | 61.22  | crédito          |
|  012  | Monitor 1024x876 60 Hz     | Tecno-k  | 62.29  | crédito          |
|  001  | Procesador 2 GHz 32 bits   | Patito   | 30.40  | efectivo         |
|  002  | Procesador 2.4 GHz 32 bits | Patito   | 33.63  | efectivo         |
|  003  | Procesador 1.7 GHz 64 bits | Patito   | 195.59 | efectivo         |
|  005  | RAM 128MB 333 MHz          | Patito   | 9.78   | efectivo         |
|  006  | RAM 256MB 400 MHz          | Patito   | 32.44  | efectivo         |
|  007  | Disco Duro 80 GB 7200 rpm  | Patito   | 59.99  | efectivo         |
|  010  | Disco Duro 40 GB 4200 rpm  | Patito   | 62.02  | efectivo         |
|  001  | Procesador 2 GHz 32 bits   | Nacional | 30.82  | crédito,efectivo |
|  002  | Procesador 2.4 GHz 32 bits | Nacional | 32.73  | crédito,efectivo |
|  003  | Procesador 1.7 GHz 64 bits | Nacional | 202.25 | crédito,efectivo |
|  005  | RAM 128MB 333 MHz          | Nacional | 9.76   | crédito,efectivo |
|  006  | RAM 256MB 400 MHz          | Nacional | 31.52  | crédito,efectivo |
|  007  | Disco Duro 80 GB 7200 rpm  | Nacional | 58.41  | crédito,efectivo |
|  010  | Disco Duro 40 GB 4200 rpm  | Nacional | 64.38  | crédito,efectivo |
+-------+----------------------------+----------+--------+------------------+
20 rows in set (0.00 sec)
```

Ahora utilizamos la sentencia EXPLAIN para que MySQL nos explique cómo ha realizado esta consulta:

mysql> explain select productos.clave, concat(parte,' ',tipo,' ', especificación) as producto, proveedores.empresa , precio , pago from productos natural join precios natural join proveedores;

```
+-------------+--------+---------------+---------+---------+---------------+------+------------+
| table       | type   | possible_keys | key     | key_len | ref           | rows | Extra      |
+-------------+--------+---------------+---------+---------+---------------+------+------------+
| precios     | ALL    | NULL          | NULL    |  NULL   | NULL          | 20   |            |
| productos   | eq_ref | PRIMARY       | PRIMARY |    4    | precios.clave|  1    |            |
| proveedores | ALL    | PRIMARY       | NULL    |  NULL   | NULL          | 3    | where used |
+-------------+--------+---------------+---------+---------+---------------+------+------------+
3 rows in set (0.00 sec)
```

En cada fila del resultado, nos explica cómo ha utilizado los índices de cada tabla involucrada en la consulta. La columna 'type' nos indica el tipo de “join” que ha podido hacer. En nuestro caso, 'eq_ref', 'ref' o 'ref_or_null' indica que se ha consultado una fila de esta tabla para cada combinación de filas de las otras. Es una buena señal, se están utilizando los índices, tal como indican el resto de columnas (en concreto el atributo 'clave' que es su clave primaria).

Vemos que en las otras dos tablas, el tipo de 'join' es ALL, esto indica que el gestor ha tenido que leer toda la tabla para comprobar las condiciones que le hemos exigido en la consulta. En el caso de la tabla proveedores, habría podido utilizar la clave primaria ('possible_keys'), pero no lo ha hecho.

Vamos a intentar mejorar esta consulta. Vemos que en la tabla precios no se ha definido ningún índice, lo que facilitaría la labor al SGBD:

```
mysql> alter table precios add index empresa_idx (empresa);
Query OK, 20 rows affected (0.00 sec)
Records: 20 Duplicates: 0 Warnings: 0

mysql> alter table precios add index clave_idx (clave);
Query OK, 20 rows affected (0.00 sec)
Records: 20 Duplicates: 0 Warnings: 0

mysql> explain select productos.clave, concat(parte,' ',tipo,' ', especificación) as producto,
proveedores.empresa , precio , pago from productos natural join precios natural join proveedores;
+-------------+--------+----------------------+------------+---------+-------------+-----+-----------+
| table       | type   | possible_keys        | key        | key_len | ref         |rows | Extra     |
+-------------+--------+----------------------+------------+---------+-------------+-----+-----------+
| proveedores | ALL    | PRIMARY              | NULL       | NULL    | NULL        | 3   |           |
| precios     | ref    | empresa_idx,clave_idex| empresa_idx | 20     | productos.emp| 7   |           |
| productos   | eq_ref | PRIMARY              | PRIMARY    | 4       | precios.clave| 1   |           |
+-------------+--------+----------------------+------------+---------+-------------+-----+-----------+
3 rows in set (0.00 sec)
```

Las cosas han cambiado sustancialmente. El gestor ha pasado de leer 24 filas de datos, a leer 11. También ha cambiado el orden de lectura de las tablas, haciendo primero una lectura total de la tabla proveedores (que es inevitable ya que no hemos puesto ninguna condición en el SELECT) y, después, ha aprovechado los índices definidos en 'precios' y en 'productos'.

Veremos más sobre los índices en el subapartado 5.5 "Análisis y optimización" de esta unidad didáctica.

4.3. Manipulación de filas

Para la manipulación de filas disponemos de las sentencias SQL INSERT, UPDATE y DELETE, su uso y sintaxis ya se ha visto en el módulo 3 de este curso. En algunos casos, MySQL nos proporciona extensiones o modificadores que nos pueden ayudar mucho en determinadas situaciones.

- INSERT [DELAYED]. Cuando la sentencia INSERT puede tardar mucho en devolver el resultado (tablas muy grandes o con muchos índices que deben recalcularse al insertar una nueva fila) puede ser interesante añadir la palabra clave DELAYED para que MySQL nos devuelva el control y realice la inserción en segundo plano.

- INSERT [[LOW_PRIORITY] | [HIGH_PRIORITY]]. En tablas muy ocupadas, donde muchos clientes realizan consultas constantemente, una inserción lenta puede bloquear al resto de clientes durante un tiempo. Mediante estos modificadores podemos variar este comportamiento.

- INSERT [IGNORE]. Este modificador convierte los errores de inserción en avisos. Por ejemplo, si intentamos insertar una fila que duplica una clave primaria existente, el SGBD nos devolverá un aviso (y no insertará la nueva fila), pero nuestro programa cliente podrá continuar con su cometido si el resultado de la inserción no era importante para su correcta ejecución.

- UPDATE [LOW_PRIORITY] [IGNORE]. Se comportan de igual modo que en la sentencia INSERT.

- DELETE [QUICK]. Borra el/los registros sin actualizar los índices.

- TRUNCATE. Es una forma muy rápida de borrar todos los registros de una tabla, si no necesitamos saber el número de registros que ha borrado. DELETE FROM <tabla> realiza el mismo cometido, pero devuelve el número de registros borrados.

- LAST_INSERT_ID(). Devuelve el último identificador asignado a una columna de tipo AUTO_INCREMENT después de una sentencia INSERT.

5. Administración de MySQL

Las tareas administrativas como la instalación, gestión de usuarios, copias de seguridad, restauraciones, entre otras, son tareas ineludibles en cualquier organización. Las políticas, los recursos y preferencias de los administradores generan una gran variedad de estilos y mecanismos para llevar a cabo estas tareas, por lo que no es posible hablar de métodos completamente estandarizados en estas áreas.

En este apartado se contemplan las opciones de uso común para la administración de un servidor MySQL. Existen tantas alternativas que no es posible incluirlas todas en un curso. Por tal motivo, en este capítulo se tratan algunos temas de importancia para el administrador, desde una perspectiva general, que permiten obtener una visión global de las posibilidades prácticas de las herramientas administrativas.

En este sentido, el manual de MySQL es la referencia principal para encontrar posibilidades y resolver dudas. En especial se recomienda leer los siguientes capítulos:

- Capítulo 2. Instalación de MySQL.
- Capítulo 4. Administración bases de datos.
- Capítulo 5. Optimización.

La información contenida en ellos es muy amplia y clara, y representa una excelente guía para resolver dudas. Asimismo, se deben tener en cuenta las listas de correo incluidas en el sitio oficial www.mysql.com.

Este capítulo se inicia con una breve reseña del proceso de instalación de MySQL. En la actualidad es posible realizar la instalación a partir de binarios empaquetados que facilitan enormemente el proceso. La administración de usuarios se trata con algo más de detalle, incluyendo una breve descripción de las tablas del directorio de datos. Para los temas de copias de seguridad y restauración se muestran los comandos y utilidades de mayor uso en la práctica omitiendo algunos detalles técnicos poco usuales. La optimización se trata de manera muy general, exponiendo los temas básicos que en la práctica son pasados por alto.

Finalmente, se describe brevemente cómo realizar la replicación de datos en un servidor esclavo.

5.1. Instalación de MySQL

La instalación de MySQL no representa mayores problemas, ya que muchas distribuciones incluyen paquetes con los que realizar la instalación y configuración básica. Sin embargo, aquí veremos la instalación de MySQL utilizando el código fuente que se puede obtener en www.mysql.com. Cabe destacar que el uso de una versión de MySQL compilada tiene la ventaja de que, probablemente,

se adaptará mucho mejor al entorno del servidor donde se ejecutará, proporcionando así un mejor rendimiento. Por contra, implicará más trabajo en caso de que surjan errores en la versión y tengamos que actualizarla. Las instrucciones que se describen en este apartado se basan en la documentación incluida en la distribución.

En primer lugar, debemos asegurarnos de que contamos con las librerías y utilidades necesarias para compilar los ficheros fuente. Principalmente la lista de verificación debe incluir los ficheros siguientes:

- Compilador gcc
- Librerías libgc

El proceso de instalación incluye los siguientes pasos:

- Descomprimir los archivos fuente

```
cd /usr/local/src
tar xzvf mysql-VERSION.tar.gz
cd mysql-VERSION
```

- Configurar la versión de MySQL que vamos a obtener. El script 'configure' admite muchos parámetros que deberemos examinar mediante la opción '--help'. Según los esquemas de tabla que necesitemos o extensiones muy concretas que debamos utilizar, deberemos examinar con cuidado sus opciones. En su versión más simple lo ejecutaríamos de la siguiente manera:

```
./configure –prefix=/usr/local/mysql
```

- Compilar. Procederemos a compilar si no ha habido problemas con la configuración. El parámetro -prefix especifica la ruta del sistema de ficheros donde será instalado.

```
make
```

- Instalar el sistema el servidor ya compilado, mediante la siguiente instrucción:

```
make install
```

- Crear la base de datos inicial del servidor, la que almacenará los usuarios y privilegios. Esta base de datos es imprescindible para que los usuarios se puedan conectar al servidor.

```
scripts/mysql_istall_db
```

- Crear un nuevo usuario y su grupo, para que el servicio se ejecute en un entorno de privilegios restringido en el sistema operativo. En ningún caso se recomienda que el usuario que ejecute el servicio mysqld sea root.

```
groupadd mysql
useradd -g mysql mysql
```

- Todos los archivos deben ser propiedad de root (mysql no debe poder modificarse a sí mismo) y del grupo mysql. El directorio de datos será del usuario mysql para que pueda trabajar con las bases de datos, ficheros de registro, etc.

```
chown -R root /usr/local/mysql
chgrp -R mysql /usr/local/mysql
chown -R mysql /usr/local/mysql/var
```

- Crear el archivo de configuración. La distribución incluye varios archivos de configuración que sirven como plantilla para adaptarlo a nuestras necesidades. En este caso, utilizamos la configuración media como plantilla. Opcionalmente podemos editar el archivo /etc/my.cnf

```
cp support-files/my-medium.cnf /etc/my.cnf
```

- Lanzar el servidor

```
/usr/local/mysql/bin/mysql_safe &
```

- En este estado, el servidor no puede servir aún de SGBD. Por defecto, tendremos creado un usuario '*root*' sin contraseña que podrá acceder tanto desde el equipo local como remotamente. El siguiente paso será asignar una contraseña a este usuario y repasar los usuarios y privilegios definidos. Para asignar la contraseña, deberemos hacer lo siguiente:

```
mysqladmin -u root password "nuevapasswd"
mysqladmin -u root -h host_name password "nuevapasswd"
```

Podemos probar el funcionamiento del SGBD conectando con el cliente 'mysql':

```
mysql -u root -p
```

Veamos ahora algunas características del servidor que acabamos de instalar:

- **mysqld**. El primer método es lanzarlo directamente, se le pueden especificar las opciones que el administrador desee.
- **mysqld_safe**. Es un *script* que ejecuta *mysqld* garantizando una configuración segura. Es mucho más recomendable que ejecutar *mysqld* directamente.
- **mysql_server**. Es un guión que realiza dos tareas: iniciar y detener el servidor *mysqld* con los parámetros start y stop respectivamente. Utiliza *mysqld_safe* para lanzar el servidor *mysqld*. No es común encontrarlo con ese nombre, ya que generalmente se copia como el archivo /etc/init.d/mysql
- **mysql_multi**. Permite la ejecución de múltiples servidores de forma simultanea.

Para detener el servidor básicamente tenemos dos métodos:

- **/etc/init.d/mysql stop**. Es el mecanismo estándar en los sistemas tipo UNIX. Aunque los directorios pueden cambiar.
- **$ mysqladmin -u root -p shutdown**. Es la utilidad para realizar tareas administrativas en un servidor MySQL, en este caso le pasamos el parámetro 'shutdown' para detener el servicio.

Para que los mensajes del servidor aparezcan en español, se debe ejecutar con el parámetro *-language:*

```
$ mysqld --language=spanish
```

Otra opción es agregar en el archivo /etc/my.cnf una línea en la sección [mysqld]

```
[mysqld]
language = /usr/share/mysql/spanish
```

5.2. Usuarios y privilegios

El acceso al servidor MySQL está controlado por usuarios y privilegios. Los usuarios del servidor MySQL no tienen ninguna correspondencia con los

usuarios del sistema operativo. Aunque en la práctica es común que algún administrador de MySQL asigne los mismos nombres que los usuarios tienen en el sistema, son mecanismos totalmente independientes y suele ser aconsejable en general.

El usuario administrador del sistema MySQL se llama *root*. Igual que el superusuario de los sistemas tipo UNIX.

Además del usuario *root*, las instalaciones nuevas de MySQL incluyen el usuario anónimo, que tiene permisos sobre la base de datos test. Si queremos, también podemos restringirlo asignándole una contraseña. El usuario anónimo de MySQL se representa por una cadena vacía. Vemos otra forma de asignar contraseñas a un usuario, desde el cliente de mysql y como usuario *root*:

```
mysql> set password for ''@'localhost' = password('nuevapasswd');
```

La administración de privilegios y usuarios en MySQL se realiza a través de las sentencias:

- **GRANT**. Otorga privilegios a un usuario, en caso de no existir, se creará el usuario.
- **REVOKE**. Elimina los privilegios de un usuario existente.
- **SET PASSWORD**. Asigna una contraseña.
- **DROP USER**. Elimina un usuario.

5.2.1. La sentencia GRANT

La sintaxis simplificada de **grant** consta de tres secciones. No puede omitirse ninguna, y es importante el orden de las mismas:

- **grant** *lista de privilegios*
- **on** *base de datos.tabla*
- **to** *usuario*

Ejemplo

Creación de un nuevo usuario al que se otorga algunos privilegios

```
mysql> grant update, insert, select
-> on demo.precios
-> to visitante@localhost ;
```

En la primera línea se especifican los privilegios que serán otorgados, en este caso se permite actualizar (**update**), insertar (**in**sert) y consultar (**select**). La segunda línea especifica que los privilegios se aplican a la tabla precios de la base de datos *demo*. En la última línea se encuentra el nombre del usuario y el equipo desde el que se va a permitir la conexión.

El comando **grant** crea la cuenta si no existe y, si existe, agrega los privilegios especificados. Es posible asignar una contraseña a la cuenta al mismo tiempo que se crea y se le otorgan privilegios:

```
mysql> grant update, insert, select
-> on demo.precios
-> to visitante@localhost identified by 'nuevapasswd';
```

En la misma sentencia es posible también otorgar permisos a más de un usuario y asignarles, o no, contraseña:

```
mysql> grant update, insert, select
-> on demo.precios
-> to visitante@localhost,
-> yo@localhost identified by 'nuevapasswd',
-> tu@equipo.remoto.com;
```

5.2.2. Especificación de lugares origen de la conexión

MySQL proporciona mecanismos para permitir que el usuario realice su conexión desde diferentes equipos dentro de una red específica, sólo desde un equipo, o únicamente desde el propio servidor.

```
mysql> grant update, insert, select
-> on demo.precios
-> to visitante@'%.empresa.com';
```

El carácter % se utiliza de la misma forma que en el comando **like**: sustituye a cualquier cadena de caracteres. En este caso, se permitiría el acceso del usuario 'visitante' (con contraseña, si la tuviese definida) desde cualquier equipo del dominio 'empresa.com'. Obsérvese que es necesario entrecomillar el nombre del equipo origen con el fin de que sea aceptado por MySQL. Al igual que en **like**, puede utilizarse el carácter '_'.

Entonces, para permitir la entrada desde cualquier equipo en Internet, escribiríamos:

```
-> to visitante@'%'
```

Obtendríamos el mismo resultado omitiendo el nombre del equipo origen y escribiendo simplemente el nombre del usuario:

```
-> to visitante
```

Los anfitriones válidos también se pueden especificar con sus direcciones IP.

```
to visitante@192.168.128.10
to visitante@'192.168.128.%'
```

Los caracteres ' %' y '_' no se permiten en los nombres de los usuarios.

5.2.3. Especificación de bases de datos y tablas

Después de analizar las opciones referentes a los lugares de conexión permitidos, veamos ahora cómo podemos limitar los privilegios a bases de datos, tablas y columnas.

En el siguiente ejemplo otorgamos privilegios sobre todas las tablas de la base de datos *demo*.

```
mysql> grant all
-> on demo.*
-> to 'visitante'@'localhost';
```

Podemos obtener el mismo resultado de esta forma:

```
mysql> use demo;
mysql> grant all
-> on *
-> to 'visitante'@'localhost';
```

De igual modo, al especificar sólo el nombre de una tabla se interpretará que pertenece a la base de datos en uso:

```
mysql> use demo;
mysql> grant all
-> on precios
-> to 'visitante'@'localhost';
```

Opciones para la clausula on del comando grant

Opción	Significado
.	Todas las bases de datos y todas las tablas
base.*	Todas las tablas de la base de datos especificada
tabla	Tabla especificada de la base de datos en uso
*	Todas las tablas de la base de datos en uso

5.2.4. Especificación de columnas

A continuación presentamos un ejemplo donde se especifican las columnas sobre las que se otorgan privilegios con el comando **grant**:

```
mysql> grant update(precio,empresa)
-> on demo.precios
-> to visitante@localhost;
```

Podemos especificar privilegios diferentes para cada columna o grupos de columnas:

```
mysql> grant update(precio), select (precio, empresa)
-> on demo.precios
-> to visitante@localhost;
```

5.2.5. Tipos de privilegios

MySQL proporciona una gran variedad de tipos de privilegios.

- Privilegios relacionados con tablas: **alter**, **create**, **delete**, **drop**, **index**, **insert**, **select**, **update**
- Algunos privilegios administrativos: file, proccess, super reload, replication client, grant option, shutdown
- Algunos privilegios para fines diversos: **lock tables**, **show databases**, **create temporary tables**.

El privilegio **all** otorga todos los privilegios exceptuando el privilegio **grant option**. Y el privilegio **usage** no otorga ninguno, lo cual es útil cuando se desea, por ejemplo, simplemente cambiar la contraseña:

```
grant usage
on *.*
to visitante@localhost identified by 'secreto';
```

Tipos de privilegios en MySQL

Tipo de privilegio	Operación que permite
all [privileges]	Otorga todos los privilegios excepto **grant option**
usage	No otorga ningún privilegio
alter	Privilegio para alterar la estructura de una tabla
create	Permite el uso de **create table**
delete	Permite el uso de **delete**
drop	Permite el uso de **drop table**
index	Permite el uso de **index** y **drop index**
insert	Permite el uso de **insert**
select	Permite el uso de **select**
update	Permite el uso de **update**
file	Permite le uso de **select . . . into outfile** y **load data infile**
process	Permite el uso de **show full procces list**
super	Permite la ejecución de comandos de supervisión
reload	Permite el uso de **flush**
replication client	Permite preguntar la localización de maestro y esclavo
replication slave	Permite leer los *binlog* del maestro
grant option	Permite el uso de **grant** y **revoke**
shutdown	Permite dar de baja al servidor
lock tables	Permite el uso de **lock tables**
show tables	Permite el uso de **show tables**
create temporary tables	Permite el uso de **create temporary table**

En entornos grandes, es frecuente encontrarse en la necesidad de delegar el trabajo de administrar un servidor de bases de datos para que otros usuarios, además del administrador, puedan responsabilizarse de otorgar privilegios sobre una base de datos particular. Esto se puede hacer en MySQL con el privilegio **grant option**:

```
mysql> grant all, grant option
-> on demo.*
-> to operador@localhost;
```

El mismo resultado se puede obtener con la siguiente sintaxis alternativa:

```
mysql> grant all
-> on demo.*
-> to operador@localhost
-> with grant option;
```

De este modo el usuario *operador* podrá disponer de todos los privilegios sobre la base de datos *demo*, incluido el de controlar el acceso a otros usuarios.

5.2.6. Opciones de encriptación

MySQL puede establecer conexiones seguras encriptándolas mediante el protocolo SSL*; de esta manera, los datos que se transmiten (tanto la consulta, en un sentido, como el resultado, en el otro) entre el cliente y el servidor estarán protegidos contra intrusos. Para especificar que un usuario debe conectarse obligatoriamente con este protocolo, se utiliza la cláusula **require**:

** Secure Sockets Layer*

```
mysql> grant all
-> on *.*
-> to visitante@localhost
-> require ssl;
```

Las conexiones encriptadas ofrecen protección contra el robo de información, pero suponen una carga adicional para el servicio, que debe desencriptar la petición del cliente y encriptar la respuesta (además de un proceso más largo de negociación al conectar), por ello, merman el rendimiento del SGBD.

5.2.7. Limites de uso

Los recursos físicos del servidor siempre son limitados: si se conectan muchos usuarios al mismo tiempo al servidor y realizan consultas o manipulaciones de datos complejas, es probable que pueda decaer el rendimiento notablemente. Una posible solución a este problema es limitar a los usuarios el trabajo que pueden pedir al servidor con tres parámetros:

- Máximo número de conexiones por hora.
- Máximo número de consultas por hora.
- Máximo número de actualizaciones por hora.

La sintaxis de estas limitaciones es como se muestra a continuación:

```
mysql> grant all
-> on *.*
-> to
-> with MAX_CONECTIONS_PER_HOUR 3
-> MAX_QUERIES_PER_HOUR 300
-> MAX_UPDATES_PER_HOUR 30;
```

5.2.8. Eliminar privilegios

El comando **revoke** permite eliminar privilegios otorgados con **grant** a los usuarios. Veamos un ejemplo representativo:

```
revoke all
on *.*
from visitante@localhost;
```

Al ejecutar este comando se le retiran al usuario *visitante* todos sus privilegios sobre todas las bases de datos, cuando se conecta desde *localhost*.

El comando anterior no retira todos los privilegios del usuario *visitante*, sólo se los retira cuando se conecta desde *localhost*. Si el usuario se conecta desde otra localidad (y tenía permiso para hacerlo) sus privilegios permanecen intactos.

5.2.9. Eliminar usuarios

Antes de proceder a la eliminación de un usuario, es necesario asegurarse de que se le han quitado primero todos sus privilegios. Una vez asegurado este detalle, se procede a eliminarlo mediante el comando **drop user**:

```
mysql> drop user visitante;
```

5.2.10. La base de datos de privilegios: mysql

MySQL almacena la información sobre los usuarios y sus privilegios en una base de datos como cualquier otra, cuyo nombre es *mysql*. Si exploramos su estructura, entenderemos la manera como MySQL almacena la información de sus usuarios y privilegios:

```
mysql -u root -p
mysql> use mysql;
mysql> show tables;
+-----------------+
| Tables_in_mysql |
+-----------------+
| columns_priv    |
| db              |
| func            |
| host            |
| tables_priv     |
| user            |
+-----------------+
```

```
mysql> show columns from user;
+-----------------+------------------+------+-----+---------+-------+
| Field           | Type             | Null | Key | Default | Extra |
+-----------------+------------------+------+-----+---------+-------+
| Host            | char(60) binary  | PRI  |     |         |       |
| User            | char(16) binary  | PRI  |     |         |       |
| Password        | char(16) binary  |      |     |         |       |
| Select_priv     | enum('N','Y')    | N    |     |         |       |
| Insert_priv     | enum('N','Y')    |      | N   |         |       |
| Update_priv     | enum('N','Y')    | N    |     |         |       |
| Delete_priv     | enum('N','Y')    |      | N   |         |       |
| Create_priv     | enum('N','Y')    | N    |     |         |       |
| Drop_priv       | enum('N','Y')    | N    |     |         |       |
| Reload_priv     | enum('N','Y')    |      | N   |         |       |
| Shutdown_priv   | enum('N','Y')    | N    |     |         |       |
| Process_priv    | enum('N','Y')    | N    |     |         |       |
| File_priv       | enum('N','Y')    |      | N   |         |       |
| Grant_priv      | enum('N','Y')    |      | N   |         |       |
| References_priv | enum('N','Y')    |      | N   |         |       |
| Index_priv      | enum('N','Y')    |      | N   |         |       |
| Alter_priv      | enum('N','Y')    |      |N    |         |       |
+-----------------+------------------+------+-----+---------+-------+
17 rows in set (0.04 sec)

mysql> show columns from db;
+-----------------+------------------+------+-----+---------+-------+
| Field           | Type             | Null | Key | Default | Extra |
+-----------------+------------------+------+-----+---------+-------+
| Host            | char(60) binary  | PRI  |     |         |       |
| Db              | char(64) binary  | PRI  |     |         |       |
| User            | char(16) binary  | PRI  |     |         |       |
| Select_priv     | enum('N','Y')    |      | N   |         |       |
| Insert_priv     | enum('N','Y')    |      | N   |         |       |
| Update_priv     | enum('N','Y')    |      | N   |         |       |
| Delete_priv     | enum('N','Y')    |      | N   |         |       |
| Create_priv     | enum('N','Y')    |      | N   |         |       |
| Drop_priv       | enum('N','Y')    |      | N   |         |       |
| Grant_priv      | enum('N','Y')    |      | N   |         |       |
| References_priv | enum('N','Y')    |      | N   | N       |       |
| Index_priv      | enum('N','Y')    |      | N   |         |       |
| Alter_priv      | enum('N','Y')    |      | N   |         |       |
+-----------------+------------------+------+-----+---------+-------+
13 rows in set (0.00 sec)

mysql> show columns from tables_priv;
+-------------+-----------------------------------------------------------------------------------------------------+----+---+-------+-----+
| Field       | Type                                                                                                |Null|Key|Default|Extra|
+-------------+-----------------------------------------------------------------------------------------------------+----+---+-------+-----+
| Host        | char(60) binary                                                                                     |PRI |   |       |     |
| Db          | char(64) binary                                                                                     |PRI |   |       |     |
| User        | char(16) binary                                                                                     |PRI |   |       |     |
| Table_name  | char(60) binary                                                                                     |    | N |       |     |
| Grantor     | char(77)                                                                                            |    | N |       |     |
| Timestamp   | timestamp(14)                                                                                       |    | N |       |     |
| Table_priv  | set('Select','Insert','Update','Delete','Create','Drop','Grant','References','Index','Alter')      |    | N |       |     |
| Column_priv | set('Select','Insert','Update','References')                                                        |N   |   |       |     |
+-------------+-----------------------------------------------------------------------------------------------------+----+---+-------+-----+
8 rows in set (0.00 sec)
```

Es posible realizar modificaciones directamente sobre estas tablas y obtener los mismos resultados que si utilizáramos los comandos **grant**, **revoke**, **set password** o **drop user**:

```
mysql> update user
-> set Password = password('nuevapasswd')
-> where User ='visitante' and Host = 'localhost';
mysql> flush privileges;
```

El comando **flush privileges** solicita a MySQL que vuelva a leer las tablas de privilegios. En el momento de ejecutarse, el servidor lee la información de estas tablas sobre privilegios. Pero si se han alterado las tablas manualmente, no se enterará de los cambios hasta que utilicemos el comando **flush privileges**.

Tablas de la base de datos *mysql*

Tabla	Contenido
user	Cuentas de usuario y sus privilegios globales
db	Privilegios sobres bases de datos
tables_priv	Privilegios sobre tablas
columns_priv	Privilegios sobre columnas
host	Privilegios de otros equipos anfitriones sobre bases de datos

El acceso directo a las tablas de privilegios es útil en varios casos; por ejemplo, para borrar un usuario del sistema en las versiones de MySQL anteriores a la 4.1.1:

```
mysql> delete from user
-> where User = 'visitante' and Host = 'localhost';
mysql> flush privileges;
```

No es posible eliminar mediante un solo comando **revoke** todos los privilegios de un usuario.

Ejemplo

Se otorgan derechos a un usuario con dos comandos **grant**.

Observando el contenido de la base de datos de privilegios, podemos entender el comportamiento de los comandos **grant** y **revoke**. Primero asignamos privilegios para usar el comando **select** al usuario visitante con dos comandos **grant**: el primero de ellos le permite el ingreso desde el servidor *nuestra-ong.org* y el segundo le otorga el mismo tipo de privilegio, pero desde cualquier equipo en Internet.

```
mysql> grant select
-> on *.*
-> to visitante@nuestra-ong.org;
Query OK, 0 rows affected (0.01 sec)
mysql> grant select
-> on *.*
-> to visitante@'%';
Query OK, 0 rows affected (0.00 sec)
```

Consultando la tabla *user* de la base de datos de privilegios, podemos observar los valores 'Y' en la columna del *privilegio select*.

```
mysql> select user,host,select_priv from user
-> where user = 'visitante';
```

```
+-----------+-----------------+-------------+
|   user    |      host       | select_priv |
+-----------+-----------------+-------------+
| visitante | nuestra-ong.org |      Y      |
| visitante |        %        |      Y      |
+-----------+-----------------+-------------+
2 rows in set (0.00 sec)
```

Ahora solicitamos eliminar el *privilegio select* de todas las bases de datos y de todos los equipos en Internet.

```
mysql> revoke all
-> on *.*
-> from visitante@'%';
Query OK, 0 rows affected (0.00 sec)
mysql> select user,host,select_priv from user
-> where user = 'visitante';
```

```
+-----------+-----------------+-------------+
|   user    |      host       | select_priv |
+-----------+-----------------+-------------+
| visitante | nuestra-ong.org |      Y      |
| visitante |        %        |      N      |
+-----------+-----------------+-------------+
2 rows in set (0.01 sec)
```

En la tabla *user* observamos que, efectivamente, se ha eliminado el privilegio para visitante@'%' pero no para 'visitante@nuestra-ong.org'. MySQL considera que son direcciones diferentes y respeta los privilegios otorgados a uno cuando se modifica otro.

5.3. Copias de seguridad

Ningún sistema es perfecto ni está a salvo de errores humanos, cortes en el suministro de la corriente eléctrica, desperfectos en el *hardware* o errores de *software*; así que una labor más que recomendable del administrador del servidor de bases de datos es realizar copias de seguridad y diseñar un plan de contingencia. Se deben hacer ensayos del plan para asegurar su buen funcionamiento y, si se descubren anomalías, realizar los ajustes necesarios.

No existe una receta universal que nos indique cómo llevar nuestras copias de seguridad de datos. Cada administrador debe diseñar el de su sistema de acuerdo a sus necesidades, recursos, riesgos y el valor de la información.

MySQL ofrece varias alternativas de copia de seguridad de la información. La primera que podemos mencionar consiste simplemente en copiar los archivos de datos. Efectivamente, es una opción válida y sencilla.

En primera instancia son necesarios dos requisitos para llevarla a cabo:

- Conocer la ubicación y estructura del directorio de datos.
- Parar el servicio MySQL mientras se realiza la copia.

En cuanto a la ubicación y estructura del directorio, recordemos que la distribución de MySQL ubica el directorio de datos en /usr/local/mysql/var, las distribuciones GNU/Linux basadas en paquetes como DEB o RPM ubican, por lo general, los datos en /var/lib/mysql.

Si por algún motivo no encontramos el directorio de datos, podemos consultarlo a MySQL. El comando **show variables** nos muestra todas las variables disponibles, basta realizar un filtro con la clausula **like**:

```
mysql> show variables like 'datadir';
+---------------+-----------------+
| Variable_name |      Value      |
+---------------+-----------------+
|    datadir    | /var/lib/mysql/ |
+---------------+-----------------+
1 row in set (0.00 sec)
```

Una vez ubicados los archivos, detenemos la ejecución del servidor: un modo sencillo de asegurarnos de que la base de datos no será modificada mientras terminamos la copia:

```
$ mysqladmin -u root -p shutdown
```

Finalmente, copiamos el directorio completo con todas las bases de datos:

```
$ cp -r /var/lib/mysql/ /algun_dir/
```

Por supuesto podemos elegir otras formas de copiarlo o comprimirlo, de acuerdo a nuestras preferencias y necesidades.

```
$ tar czf mysql-backup.tar.gz /var/lib/mysql
```

Si queremos copiar sólo una base de datos, copiamos el directorio con el mismo nombre de la base de datos:

```
$ cp -r /var/lib/mysql/demo/ /algun_dir/respaldo_demo/
```

También es posible hacer copia de seguridad de una sola tabla.

```
$ cp -r /var/lib/mysql/demo/productos.* / algun_dir/backup_demo/
```

Como podemos observar, la organización de la base de datos en MySQL es muy simple:

- Todas las bases de datos se almacenan en un directorio, llamado el directorio de datos(*datadir*).

- Cada base de datos se almacena como un subdirectorio del directorio de datos.
- Cada tabla se almacena en un archivo, acompañada de otros archivos auxiliares con el mismo nombre y diferente extensión.

El problema de este mecanismo es que debemos detener el servicio de bases de datos mientras realizamos el respaldo. 

5.3.1. mysqlhotcopy

Un mecanismo que permite realizar la copia de los archivos del servidor sin necesidad de detener el servicio es el *script* '*mysqlhotcopy*'. El *script* está escrito en Perl y bloquea las tablas mientras realiza el respaldo para evitar su modificación. Se usa de la siguiente manera:

```
$ mysqlhotcopy demo /algun_directorio
```

En este caso, creará un directorio /algun_directorio/demo con todos los archivos de la base de datos.

El comando *mysqlhotcopy* puede recibir sólo el nombre de una base de datos como parámetro:

```
$ mysqlhotcopy demo
```

En este caso, creará un directorio /var/lib/mysql/demo_copy.

Este método no funciona para tablas con el mecanismo de almacenamiento tipo InnoDB. 

5.3.2. mysqldump

Las dos opciones anteriores representan copias binarias de la base de datos. El comando *mysqldump*, en cambio, realiza un volcado de las bases de datos pero traduciéndolas a SQL; es decir, entrega un archivo de texto con todos los comandos necesarios para volver a reconstruir las bases de datos, sus tablas y sus datos. Es el método más útil para copiar o distribuir una base de datos que deberá almacenarse en otros servidores. 

```
$ mysqldump demo > demo.sql
```

El comando *mysqldump* ofrece multitud de parámetros para modificar su comportamiento o el tipo de volcado generado: por defecto, genera sentencias SQL, pero puede generar ficheros de datos tipo CSV u otros formatos. También podemos especificarle que haga el volcado de todas las bases de datos o que sólo vuelque los datos y no la creación de las tablas, etc.

Las primeras líneas del archivo *demo.sql* según el ejemplo anterior tendrían el siguiente aspecto:

```
~$ mysqldump demo | head -25
-- MySQL dump 8.21
--
-- Host: localhost Database: demo
---------------------------------------------------------
-- Server version 3.23.49-log
--
-- Table structure for table 'ganancia'
--
DROP TABLE IF EXISTS ganancia;
CREATE TABLE ganancia (
venta enum('Por mayor','Por menor') default NULL,
factor decimal(4,2) default NULL
) TYPE=MyISAM;
--
Dumping data for table 'ganancia'
--
INSERT INTO ganancia VALUES ('Por mayor',1.05);
INSERT INTO ganancia VALUES ('Por menor',1.12);--
```

La ventaja de utilizar *mysqldump* es que permite que los archivos puedan ser leídos (y modificados) en un simple editor de textos, y pueden ser utilizados para migrar la información a otro SGBD que soporte SQL. Además soporta todos los tipos de tablas. La desventaja es que su procesamiento es lento y los archivos que se obtienen son muy grandes. 

5.3.3. Restaurar a partir de respaldos

En algún momento, sea por el motivo que sea, necesitaremos realizar la restauración de nuestras bases de datos.

Si tenemos una copia binaria del directorio de datos, bastará con copiarla al directorio original y reiniciar el servidor:

```
# mysqladmin -u root -p shutdown
# cp /algun_dir/respaldo-mysql/* /var/lib/mysql
# chown -R mysql.mysql /var/lib/mysql
# mysql_safe
```

Es importante restaurar también el dueño y el grupo de los archivos de datos, para tener los accesos correctamente establecidos. En este ejemplo se adopta el supuesto que el usuario mysql es el que ejecuta el servidor *mysqld*.

La restauración de un archivo SQL obtenido con *mysqldump*, se realiza desde el cliente *mysql*, la base de datos debe existir, ya que el archivo *demo.sql* no la crea por defecto.

```
$ mysql demo -u root -p < demo.sql
```

5.4. Reparación de tablas

En determinadas circunstancias de uso muy frecuente, como la inserción y borrado masivos de datos, coincidiendo con bloqueos del sistema o llenado del espacio en disco u otras circunstancias, es posible que una tabla o algunos de sus índices se corrompan.

Podemos consultar el estado de integridad de una tabla con el comando **check table**, que realiza algunas verificaciones sobre la tabla en busca de errores y nos entrega un informe con las siguientes columnas de información:

```
mysql> check table precios;
+--------------+-------+----------+----------+
|    Table     |  Op   | Msg_type | Msg_text |
+--------------+-------+----------+----------+
| demo.precios | check |  status  |    OK    |
+--------------+-------+----------+----------+
1 row in set (0.03 sec)
```

La columna *Op* describe la operación que se realiza sobre la tabla. Para el comando **check table** esta columna siempre tiene el valor *check* porque ésa es la operación que se realiza. La columna *Msg_type* pude contener uno de los valores *status*, *error*, info, o *warning*. Y la columna *Msg_text* es el texto que reporta de alguna situación encontrada en la tabla.

Es posible que la información entregada incluya varias filas con diversos mensajes, pero el último mensaje siempre debe ser el mensaje *OK* de tipo *status*.

En otras ocasiones **check table** no realizará la verificación de tabla, en su lugar entregará como resultado el mensaje *Table is already up to date*, que significa que el gestor de la tabla indica que no hay necesidad de revisarla.

MySQL no permite realizar consultas sobre una tabla dañada y enviará un mensaje de error sin desplegar resultados parciales:

```
mysql> select * from precios;
ERROR 1016: No puedo abrir archivo: 'precios.MYD'. (Error: 145)
```

Para obtener información del significado del error 145, usaremos la utilidad en línea de comandos *perror*:

```
$ perror 145
145 = Table was marked as crashed and should be repaired
```

Después de un mensaje como el anterior, es el momento de realizar una verificación de la integridad de la tabla para obtener el reporte.

```
mysql> check table precios extended;
+-------------+------+----------+----------------------------------------+
|    Table    |  Op  | Msg_type |              Msg_text                  |
+-------------+------+----------+----------------------------------------+
|demo.precios |check |  error   |Size of datafile is:450 Should be:452 |
|demo.precios |check |  error   |               Corrupt                  |
+-------------+------+----------+----------------------------------------+
2 rows in set (0.00 sec)
```

En este caso localizamos dos errores en la tabla. La opción **extended** es uno de los cinco niveles de comprobación que se pueden solicitar para verificar una tabla.

Tipos de verificación

Tipo	Significado
quick	No revisa las filas en busca de referencias incorrectas.
fast	Solamente verifica las tablas que no fueron cerradas adecuadamente.
changed	Verifica sólo las tablas modificadas desde la última verificación o que no se han cerrado apropiadamente.
medium	Revisa las filas para verificar que los ligados borrados son correctos, verifica las sumas de comprobación de las filas.
extended	Realiza una búsqueda completa en todas las claves de cada columna. Garantiza el 100% de la integridad de la tabla.

La sentencia **repair table** realiza la reparación de tablas tipo MyISAM corruptas:

```
mysql> repair table precios;
+-------------+------+---------+----------------------------------------+
|    Table    |  Op  | Msg_type|               Msg_text                 |
+-------------+------+---------+----------------------------------------+
|demo.precios |repair|   info  | Wrong bytesec: 0-17-1 at 168;Skipped |
|demo.precios |repair| warning | Number of rows changed from 20 to 7  |
|demo.precios |repair|  status |                  OK                    |
+-------------+------+---------+----------------------------------------+
3 rows in set (0.04 sec)
```

El segundo mensaje informa de la pérdida de 13 filas durante el proceso de reparación. Esto significa, como es natural, que el comando **repair table** es útil sólo en casos de extrema necesidad, ya que no garantiza la recuperación total de la información. En la práctica, siempre es mejor realizar la restauración de la información utilizando las copias de seguridad. En caso de desastre, se debe conocer el motivo que origina la corrupción de las tablas y tomar las medidas adecuadas para evitarlo. En lo que respecta a la estabilidad de MySQL, se puede confiar en que muy probablemente nunca será necesario utilizar el comando **repair table**.

El comando **optimize table** puede también realizar algunas correcciones sobre una tabla.

5.4.1. myisamchk

El programa **myisamchk** es una utilidad en línea de comandos que se incluye con la distribución de MySQL y sirve para reparar tablas tipo MyISAM. Para utilizarlo con seguridad el servidor no debe estar ejecutándose y se recomienda realizar un respaldo del directorio de datos antes de su utilización.

Recibe como parámetro principal los archivos .MYI correspondientes a las tablas que hay que revisar; es decir, **myisamchk** no conoce la ubicación del directorio de datos. Por ejemplo, si el directorio de datos está ubicado en /val/lib/mysql, las siguientes serían dos maneras de realizar una comprobación de los archivos de la base de datos *demo*:

```
# myisamchk /var/lib/mysql/demo/*.MYI
# cd /var/lib/lib/mysql/demo
# myisamchk *.MYI
```

Se pueden revisar todas las bases de datos utilizando '*' para denominar el directorio de la base de datos:

```
# myisamchk /var/lib/mysql/*/*.MYI
```

Para realizar una comprobación rápida, el manual sugiere utilizar el siguiente comando:

```
# myisamchk --silent --fast *.MYI
```

Y para realizar la corrección de las tablas corruptas, el manual sugiere la sintaxis siguiente:

```
# myisamchk --silent --force --update-state -O key_buffer=64M \
-O sort_buffer=64M -O read_buffer=1M -O write_buffer=1M *.MYI
```

Las opciones dadas por -O se refieren al uso de memoria, que permiten acelerar de forma notoria el proceso de reparación.

`--force` reinicia myisamchk con el parámetro `--recover` cuando encuentra algún error.

`--updatestate` almacena información sobre el resultado del análisis en la tabla MYI.

Nota

En la práctica con estas opciones se logran corregir los errores más comunes. Para conocer otras opciones de recuperación con **myisamchk**, podéis consultar el manual que acompaña a la distribución de MySQL.

5.5. Análisis y optimización

El diseño de MySQL le permite funcionar con un rendimiento notable, sin embargo, se pueden cometer fácilmente errores que disminuyan la capacidad de respuesta del servidor. También se pueden realizar algunos ajustes a la configuración de MySQL que incrementan su rendimiento.

5.5.1. Indexación

La indexación es la principal herramienta para optimizar el rendimiento general de cualquier base de datos. Es también la más conocida por los usuarios de servidores MySQL y, paradójicamente, su no utilización es una de las principales causas de bajo rendimiento en servidores de bases de datos.

> Muchos administradores y diseñadores simplemente parecen olvidar usar índices para optimizar los accesos a las bases de datos. Por otro lado, algunas personas tienden a indexar todo, esperando que de esta manera el servidor acelere cualquier tipo de consulta que se le solicite. En realidad, esta práctica puede causar una disminución en el rendimiento, sobre todo en lo que respecta a inserciones y modificaciones.

Para ver las ventajas de utilizar índices, analizaremos en primer término una simple búsqueda en una tabla sin índice alguno:

- El constante acceso de escritura de una tabla la mantiene desordenada.
- La ordenación de una tabla es una operación costosa: el servidor tendría que detenerse un tiempo considerable para ordenar sus tablas.
- Muchas tablas tienen más de un criterio de ordenación: ordenar según una columna implica desordenar otra.
- La inserción y eliminación de datos sin alterar el orden en una tabla es costosa: la inserción de un registro en una tabla grande implicaría una larga espera en la actualización de la misma.
- Si se opta por mantener la tabla desordenada (que es la opción más viable), una búsqueda implicaría forzosamente un recorrido secuencial (también denominado *full scan*), registro por registro.

El uso de índices en la ordenación de las bases de datos ofrece las ventajas siguientes:

- Permite ordenar las tablas por varios criterios simultáneamente.
- Es menos costoso ordenar un archivo índice, porque incluye sólo referencias a la información y no la información en sí.
- El coste de inserción y eliminación es menor.
- Con los registros siempre ordenados se utilizaran algoritmos mucho más eficientes que el simple recorrido secuencial en las consultas.

El uso de índices también comporta alguna desventaja:

- Los índices ocupan espacio en disco.

- Aún teniendo registros pequeños, el mantener en orden un índice disminuye la velocidad de las operaciones de escritura sobre la tabla.

A pesar de estos inconvenientes, la utilización de índices ofrece mayores ventajas que desventajas, sobre todo en la consulta de múltiples tablas, y el aumento de rendimiento es mayor cuanto mayor es la tabla.

Consideremos por ejemplo una consulta sobre las tablas A, B, y C, independientemente del contenido de la cláusula **where**, las tres tablas se deben de combinar para hacer posible posteriormente el filtrado según las condiciones dadas:

```
select *
from A,B,C
where A.a = B.b
and B.b = C.c;
```

Consideremos que no son tablas grandes, que no sobrepasan los 1.000 registros. Si A tiene 500 registros, B tiene 600 y C 700, la tabla resultante de la consulta anterior tendrá 210 millones de registros. MySQL haría el producto cartesiano de las tres tablas y, posteriormente, se recorrería la relación resultante para buscar los registros que satisfacen las condiciones dadas, aunque al final el resultado incluya solamente 1.000 registros.

Si utilizamos índices MySQL los utilizaría de una forma parecida a la siguiente:

- Tomaría cada uno de los registros de A.

- Por cada registro de A, buscaría los registros en B que cumpliesen con la condición A.a = B.b. Como B está indexado por el atributo 'b', no necesitaría hacer el recorrido de todos los registros, simplemente accedería directamente al registro que cumpliese la condición.

- Por cada registro de A y B encontrado en el paso anterior, buscaría los registros de C que cumpliesen la condición B.b = C.c. Es el mismo caso que en el paso anterior.

Comparando las dos alternativas de búsqueda, la segunda ocuparía cerca del 0,000005% del tiempo original. Por supuesto que sólo se trata de una aproximación teórica, pero adecuada para comprender el efecto de los índices en las consultas sobre bases de datos.

5.5.2. Equilibrio

El índice ideal debería tener las siguientes características:

- Los registros deberían ser lo más pequeños posible.
- Sólo se debe indexar valores únicos.

Analicemos cada recomendación:

- Cuanto más pequeños sean los registros, más rápidamente se podrán cambiar de lugar (al insertar, modificar o borrar filas), además, en un momento dado, el índice puede permanecer en memoria. Consideremos las dos definiciones posibles:

```
create table Empresa(
nombre char(30),
teléfono char(20),
index (nombre)
);
```

En esta tabla el índice se realiza sobre *nombre*, que es un campo de 30 caracteres, y se utiliza como clave para hacer los '*joins*' con otras tablas.

Ahora considérese la siguiente alternativa:

```
create table Empresa(
id int ,
nombre char(30),
teléfono char(20),
index (id)
);
```

Se agrega una columna que servirá como identificador de la empresa. Desde el punto de vista de rendimiento implica una mejora, ya que el índice se realiza sobre números enteros, por lo tanto, ocupará menos espacio y funcionará más rápido.

Cuanto más pequeña sea la columna indexada mayor velocidad se tendrá en el acceso a la tabla.

- Consideremos el índice siguiente, creado para disminuir la necesidad de efectuar accesos a la tabla:

```
create table Empresa(
nombre char(30),
crédito enum{'SI','NO'},
index(crédito)
);
```

Si consideramos que un índice se crea para evitar la necesidad de recorrer la tabla, veremos que el índice creado es prácticamente inútil, ya que alguno de los valores ocurre el 50% o más de las veces: para encontrar todos los resultados hay que recorrer gran parte de la tabla. MySQL no utiliza los índices que implican un 30% de ocurrencias en una tabla.

Aun así, y exceptuando casos exagerados como este último, puede ser interesante indexar una tabla por algún atributo que no sea único, si ese atributo se utiliza para ordenar los resultados. También puede ser conveniente crear un índice por varios atributos simultáneamente si se usan todos en alguna consulta en la cláusula ORDER BY.

Cuanto menor sea la repetición de valores en una columna indexada, menor será la necesidad de acceder a la tabla y más eficiente será el índice.

5.5.3. La *cache* de consultas de MySQL

El servidor MySQL incluye la posibilidad de utilizar una *cache** con los resultados de las últimas consultas para acelerar la velocidad de respuesta. Esta solución es útil cuando las tablas tienen relativamente pocos cambios y se realizan los mismos tipos de consultas. El funcionamiento de la *cache* se basa en las premisas siguientes:

*** Memoria intermedia de acceso rápido.**

- La primera vez que se recibe una consulta se almacena en la *cache*.
- Las siguientes veces la consulta se realiza primero en la *cache*; si tiene éxito, el resultado se envía inmediatamente.

La *cache* tiene las siguientes características:

- El servidor compara el texto de la consulta; aunque técnicamente sea igual si difiere en uso de mayúsculas-minúsculas o cualquier otro cambio, no se considera la solicitud idéntica y no será tratada por la *cache*.
- Si alguna tabla incluida en alguna consulta cambia, el contenido de la consulta es eliminado de la *cache*.

La configuración de la *cache* se realiza a través de variables globales:

- **query_cache_limit**. No almacena resultados que sobrepasen dicho tamaño. Por omisión es de 1M.
- **query_cache_size**. Tamaño de la memoria cache expresada en bytes. Por omisión es 0; es decir, no hay cache.
- **query_cache_type**. Puede tener tres valores: ON , OFF o DEMAND.

Tipos de *cache*

Valor	Tipo	Significado
0	OFF	*Cache* desactivado
1	ON	*Cache* activado
2	DEMAND	Sólo bajo solicitud explicita

Cuando la *cache* del servidor esta en modo DEMAND, se debe solicitar explícitamente que la consulta utilice o no la *cache*:

```
select sql_cache
select sql_no_cache
```

5.6. Replicación

La replicación es la copia sincronizada entre dos servidores de bases de datos de forma que cualquiera de los dos puede entregar los mismos resultados a sus clientes.

MySQL incluye la posibilidad de replicación con las siguientes características:

- Funciona con el esquema *maestro-esclavo*: existe un servidor maestro que lleva el control central y uno o varios servidores esclavos que se mantienen sincronizados con el servidor maestro.

- La réplica se realiza mediante un registro de los cambios realizados en la base de datos: no se realizan las copias de las bases de datos para mantenerlas sincronizadas, en su lugar se informa de las operaciones realizadas en el servidor maestro (insert, delete , update ...) para que las realicen a su vez los servidores esclavos.

- No es posible realizar cambios en los servidores esclavos, son exclusivamente para consultas.

Este sencillo esquema permite la creación de replicas sin mayores complicaciones obteniendo los siguientes beneficios:

- Se distribuye la carga de trabajo.

- El sistema es redundante, por lo que en caso de desastre hay menos probabilidades de perder los datos.

- Es posible realizar los respaldos de un esclavo sin interrumpir el trabajo del servidor maestro.

5.6.1. Preparación previa

El equipo maestro debe tener acceso por red. Antes de realizar la configuración de los servidores maestro y esclavo es necesario realizar las siguientes tareas:

- Asegurarse de que en ambos está instalada la misma versión de MySQL.
- Asegurarse de que ninguno de los servidores atenderá peticiones durante el proceso de configuración.
- Asegurarse de que las bases de datos del servidor maestro han sido copiadas manualmente en el servidor esclavo, de manera que en ambos se encuentre exactamente la misma información.
- Asegurarse de que ambos atienden conexiones vía TCP/IP. Por seguridad, esta opción está desactivada por omisión. Para activarla se debe comentar la línea *skip_networking* en el archivo de configuración /etc/my.cnf

5.6.2. Configuración del servidor maestro

En el servidor maestro creamos una cuenta de usuario con permisos de replicación para autorizar, en el servidor maestro, al nuevo usuario para realizar réplicas:

```
mysql> grant replication slave
-> on *.*
-> to replicador@esclavo.empresa.com identified by 'secreto';
```

Replicador es el nombre del nuevo usuario.

Esclavo.empresa.com es la dirección del servidor esclavo.

'Secreto' es la contraseña.

El servidor maestro llevará un archivo de registro '*binlog*' donde se registrarán todas las solicitudes de actualización que se realicen en las bases de datos. Para activar la creación de este archivo debemos editar el archivo /etc/my.cnf y agregar las siguientes líneas en la sección [mysqld]:

```
[mysqld]
log-bin
server-id = 1
```

El servidor maestro debe identificarse con un id, en este caso será el número 1. a continuación, reiniciamos el servidor:

```
/etc/init.d/mysql restart
```

Finalmente, consultamos el nombre del archivo '*binlog*' y la posición de compensación (estos datos son necesarios para configurar el esclavo):

```
mysql> show master status;
+-----------------+----------+--------------+------------------+
|      File       | Position | Binlog_do_db | Binlog_ignore_db |
+-----------------+----------+--------------+------------------+
| maestro-bin.001 |    76    |              |                  |
+-----------------+----------+--------------+------------------+
```

5.6.3. Configuración del servidor esclavo

En el servidor esclavo, editamos el archivo /etc/my.cnf y agregamos, al igual que en el maestro, la activación del archivo '*binlog*' y un identificador del servidor (que debe ser distinto del identificador del servidor maestro):

```
[mysqld]
log-bin
server-id = 2
```

Reiniciamos el servidor esclavo:

```
# /etc/init.d/mysql restart
```

Configuramos los datos del maestro en el servidor esclavo:.

```
mysql> change master to
-> master_host = 'maestro.empresa.com',
-> master_user = 'replicador',
-> master_password = 'secreto',
-> master_log_file = 'maestro-log.001',
-> master_log_pos = 76;
```

El último paso es iniciar el servidor esclavo:

```
mysql> start slave;
```

Y ya tendremos el servidor esclavo funcionando.

5.7. Importación y exportación de datos

En muchas ocasiones es necesario mover datos de una aplicación a otra, para ello son necesarios formatos estándares que puedan ser escritos por la aplicación origen y leídos por la aplicación destino. El más simple de esos formatos es el texto plano, donde cada archivo es una tabla, cada fila es un registro y los valores de los campos se separan por tabuladores.

MySQL puede leer este tipo de archivos, incluyendo valores nulos representados por '\N'(N mayúscula).s

Utilizando el cliente *mysql*, podemos introducir los datos del archivo local *proveedores.txt* en la tabla *proveedores*:

```
mysql> load data local infile 'proveedores.txt'
-> into table proveedores;
```

Si se omite la palabra **local**, MySQL buscará el archivo en el servidor y no en el cliente. 

En un archivo se pueden entrecomillar los campos, utilizar comas para separarlos y terminar las líneas con los caracteres '\r\n' (como en los archivos Windows). El comando **load data** tiene dos clausulas opcionales, **fields**, en el que se especifican estos parámetros.

```
mysql> load data local infile 'prooveedores.txt'
-> fields terminated by ','
-> enclosed by '"'
-> lines terminated by '\r\n';
```

La opción **enclosed by** puede tener la forma **optionaly enclosed by**, en caso de que los campos numéricos no sean delimitados.

Además pueden omitirse las primeras lineas del archivo si contienen información de encabezados:

```
mysql> load data local infile 'proveedores.txt'
-> ignore 1 lines;
```

5.7.1. mysqlimport

La utilidad **mysqlimport** que se incluye en la distribución puede realizar el mismo trabajo que **load data**. Estos son algunos de sus parámetros:

```
mysqlimport basededatos archivo.txt
```

Estos son algunos de los argumentos de **mysqlimport** para realizar las tareas equivalentes a la sentencia **load data**:

```
--fields-terminated-by=
--fields-enclosed-by=
--fields-optionally-enclosed-by=
--fields-escaped-by=
--lines-terminated-by=
```

La forma más simple para exportar datos es redireccionando la salida del cliente *mysql*. El parámetro -e permite ejecutar un comando en modo de procesamiento por lotes. MySQL detecta si la salida es en pantalla o está redireccionada a un archivo y elige la presentación adecuada: con encabezados y líneas de separación para la salida en pantalla, y sin encabezados y con tabuladores para un archivo:

```
$ mysql demo -e "select * from proveedores" > proveedores.txt
```

La sentencia **select** también cuenta con una opción para realizar la tarea inversa de la sentencia **load data**:

```
mysql> select *
-> into outfile "/tmp/proveedores.txt"
-> fields termitated by ','
-> optionaly enclosed by '"'
-> lines termitates by '\n'
-> from proveedores;
```

5.7.2. mysqldump

La utilidad **mysqldump** realiza el volcado de bases de datos y puede utilizarse para transportar datos de una base a otra que también entienda SQL. Sin embargo, el archivo debe ser editado antes de utilizarse, ya que algunas opciones

son exclusivas de MySQL. Por lo general, basta con eliminar el tipo de tabla que se especifica al final de un comando **create table**.

El siguiente comando realiza el vaciado completo de la base de datos *demo*:

```
$ mysqldump demo > demo.sql
```

En algunos casos, los comandos **insert** son suficientes y no necesitamos las definiciones de las tablas.

El siguiente comando realiza un vaciado de la tabla *proveedores* de la base de datos *demo* filtrando la salida con el comando *grep* de UNIX que selecciona sólo las líneas que contienen la palabra INSERT. De este modo, el archivo *proveedores-insert.txt* contiene exclusivamente comandos **insert**:

```
$ mysqldump demo proveedores | grep INSERT
```

6. Clientes gráficos

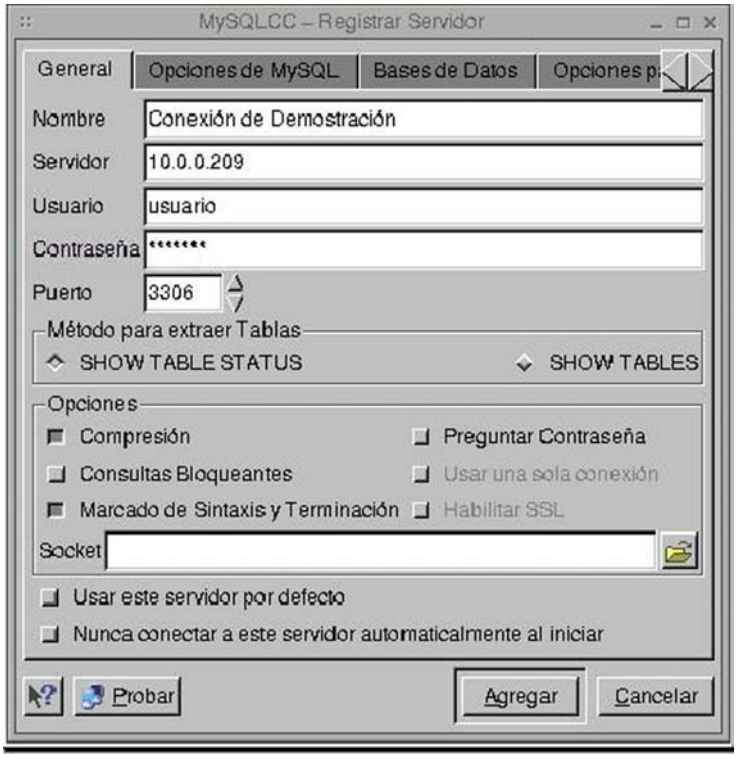
Existen múltiples clientes de entorno gráfico que permiten la interacción con un servidor MySQL. Analizaremos brevemente los que distribuye la empresa MySQL AB (*mysqlcc, mysql-query-browser y mysql-administrator*) y que se pueden descargar del sitio ofical www.mysql.com.

Nota

Actualmente, *mysqlcc* se ha dado por obsoleto en favor de los otros dos.

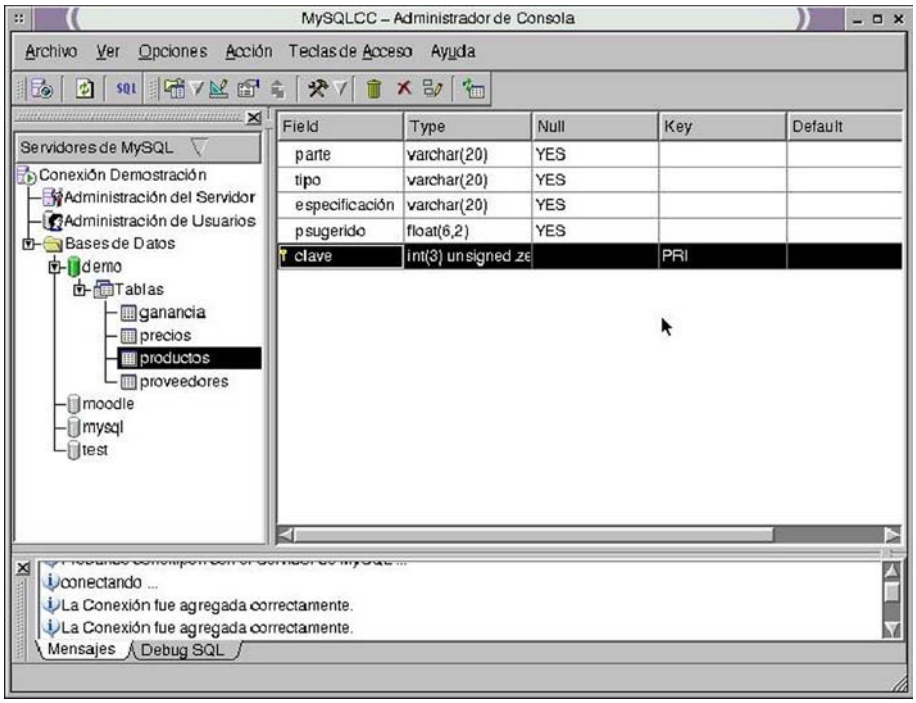
6.1. mysqlcc

Al ejecutarse por primera vez abrirá el diálogo que permite realizar el registro de un nuevo servidor MySQL:



En la ventana principal se pueden apreciar los servidores registrados, que en este caso es solamente uno.

Con el botón derecho del ratón sobre “Conexión de Demostración”, se puede activar la conexión. Después de eso, *mysqlcc* muestra las propiedades de los elementos de la base de datos.

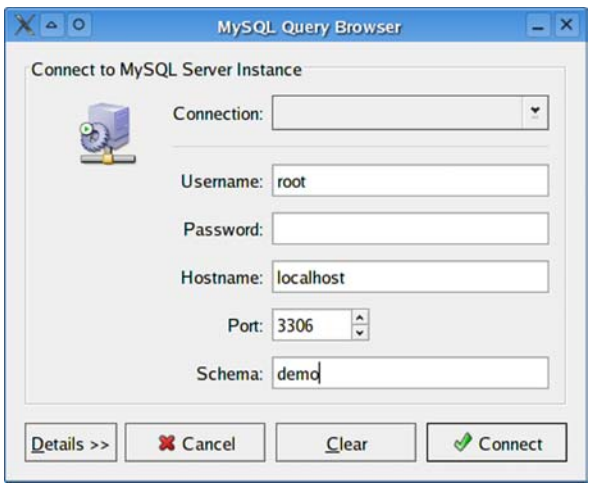


Ahora ya estamos en disposición de realizar consultas SQL con Ctrl-Q (o haciendo click sobre el icono 'SQL'). Se abrirá una nueva ventana en la que podremos escribir la consulta que, una vez escrita, se ejecutará al teclear Ctrl-E. Los resultados se mostrarán en forma de tabla como en la captura de pantalla anterior.

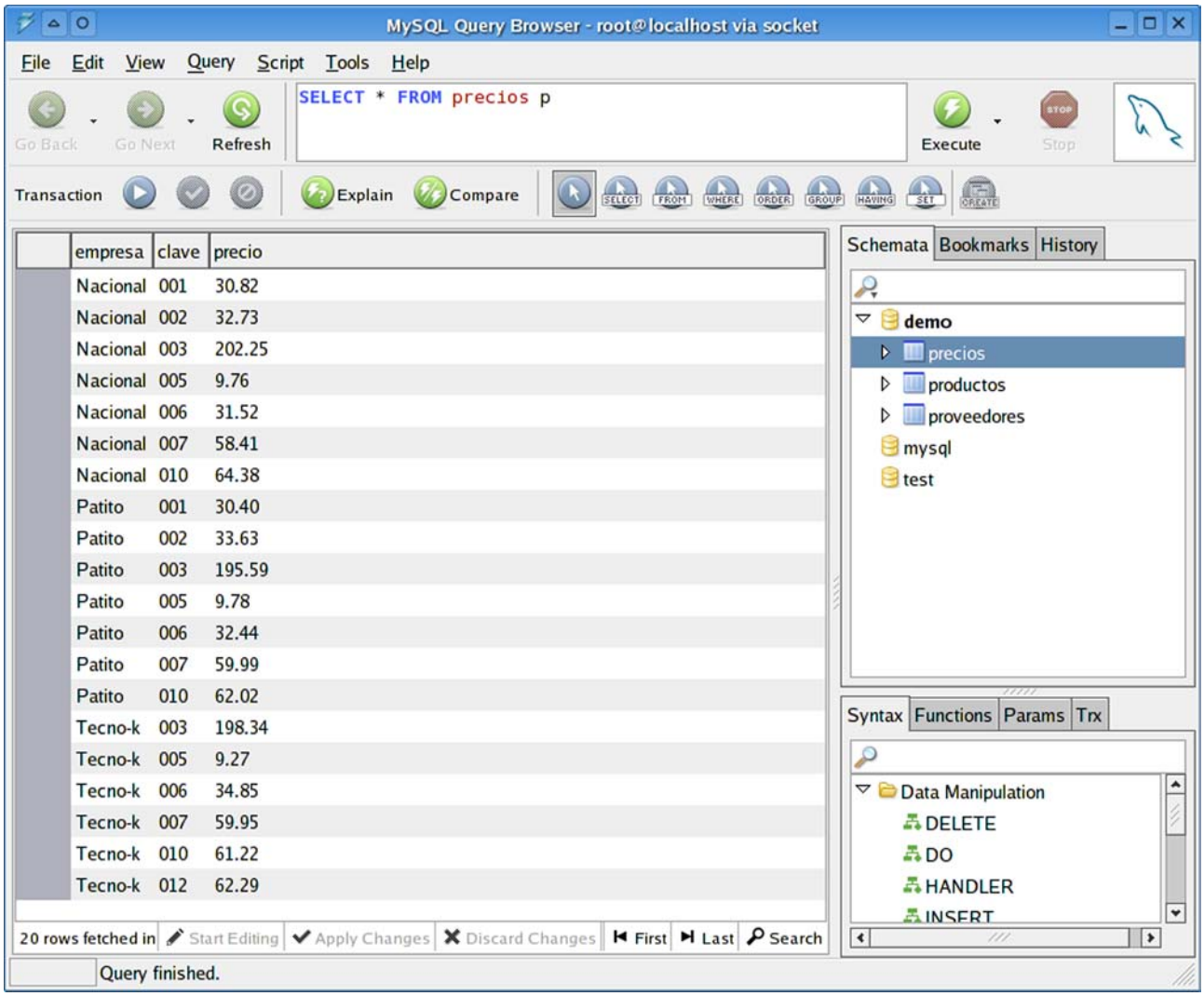
mysqlcc ofrece múltiples opciones para realizar inserciones, eliminaciones, configurar teclas de acceso rápido y una serie de características de uso muy intuitivo. También ofrece prestaciones para exportar el resultado de una consulta a un fichero de texto.

6.2. mysql-query-browser

Tanto *mysql-query-browser* como *mysql-administrator* comparten la información relativa a las conexiones almacenadas. La pantalla inicial nos permitirá seleccionar una existente o configurar una nueva:



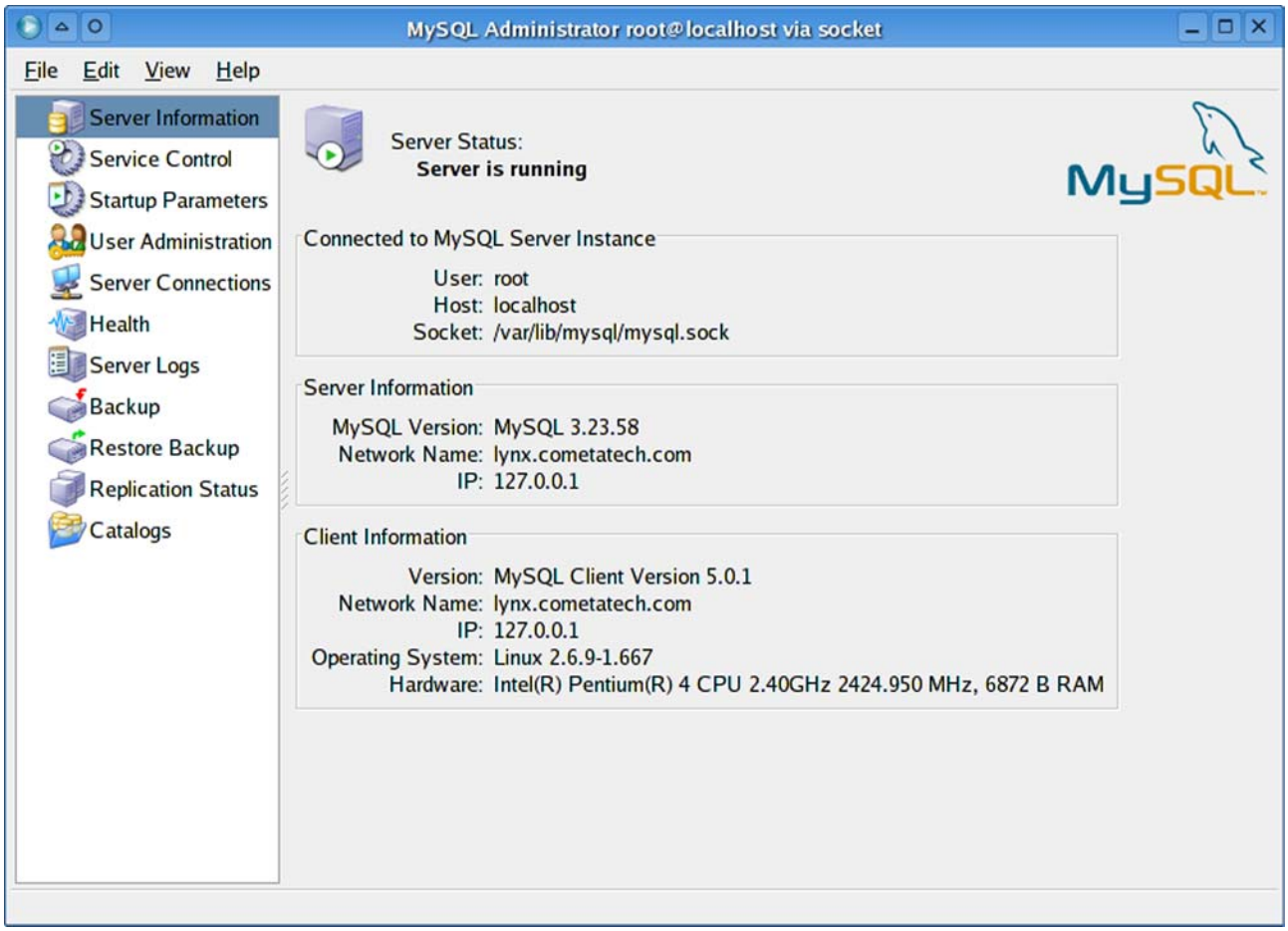
El aspecto de este programa es mejor que el de *mysqlcc* y ofrece prestaciones de ayuda en la generación de consultas, favoritos, marcadores, accesos rápidos a *EXPLAIN*, etc.



Asimismo, ofrece varios formatos de exportación de los resultados de una consulta y más facilidades a la hora de navegar por los resultados.

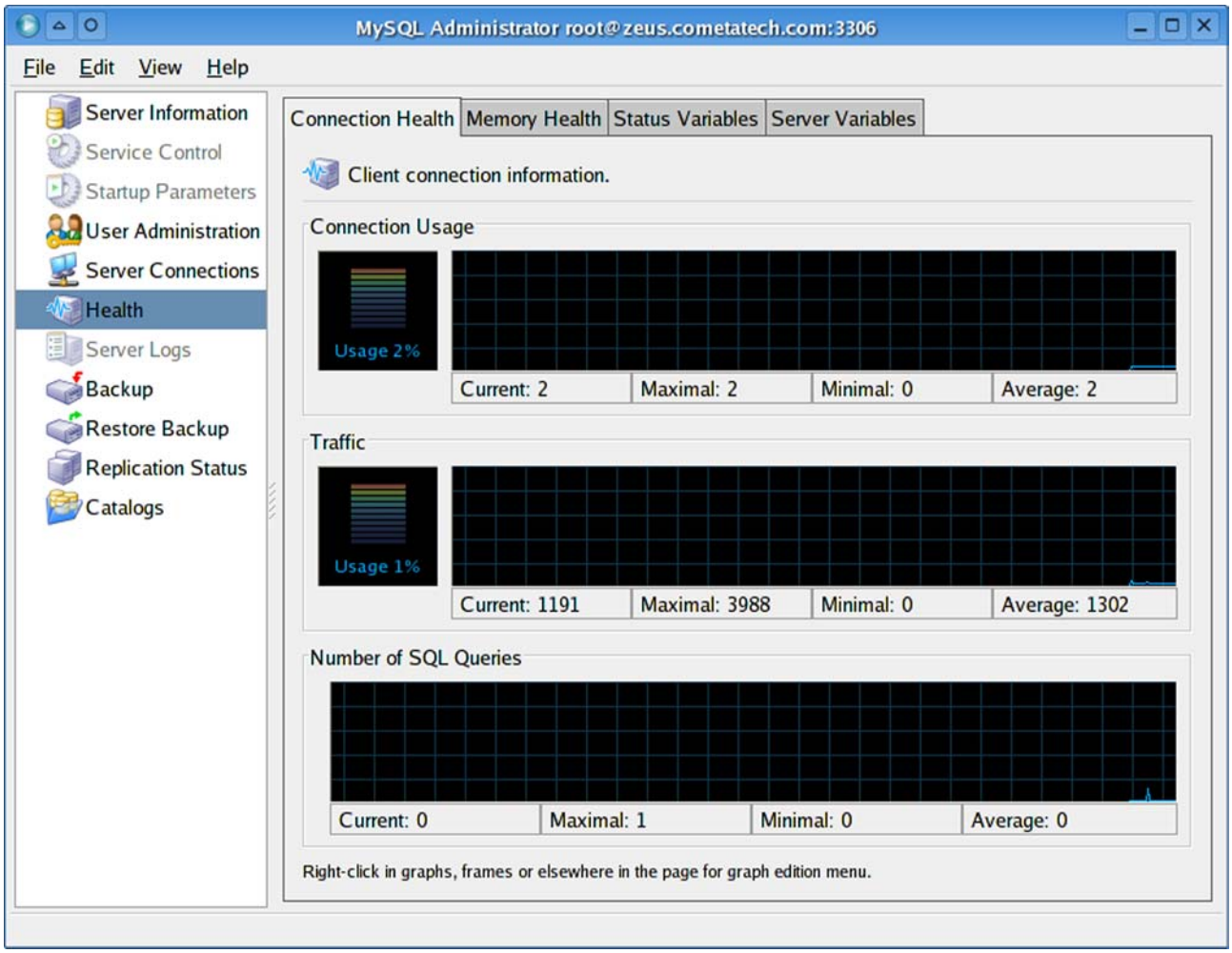
6.3. mysql-administrator

Esta novedosa herramienta es extremadamente potente y completa en cuanto a tareas de administración se refiere.



mysql-administrator permite, entre otras cosas:

- Encender y parar el SGBD.
- Gestionar el fichero de configuración */etc/my.cnf* de forma gráfica.
- Gestionar usuarios y privilegios.
- Monitorizar el uso del gestor que se está haciendo del mismo, el número de conexiones, consultas simultáneas y todo tipo de información estadística.
- Consultar los ficheros de registro (*log*) del servidor.
- Gestionar copias de seguridad.
- Gestionar la replicación de bases de datos.
- Crear y borrar bases de datos (SCHEMA).


MySQL Administrator root@zeus.cometatech.com:3306
File Edit View Help
Server Information
Service Control
Startup Parameters
User Administration
Server Connections
Health
Server Logs
Backup
Restore Backup
Replication Status
Catalogs
Connection Health
Memory Health
Status Variables
Server Variables
Client connection information.
Connection Usage
Usage 2%
Current: 2
Maximal: 2
Minimal: 0
Average: 2
Traffic
Usage 1%
Current: 1191
Maximal: 3988
Minimal: 0
Average: 1302
Number of SQL Queries
Current: 0
Maximal: 1
Minimal: 0
Average: 0
Right-click in graphs, frames or elsewhere in the page for graph edition menu.

Resumen

MySQL es un SGBD relacional de fácil uso y alto rendimiento, dos características muy valiosas para un desarrollador de sistemas: su facilidad de uso permite la creación de bases de datos con rapidez y sin muchas complicaciones, y su alto rendimiento lo hace sumamente atractivo para aplicaciones comerciales importantes o portales web de mucho tráfico. Si a ello le añadimos la disponibilidad de código y su licencia dual, se comprende que MySQL sea atractivo y accesible para todo el mundo.

Estos atributos tienen sus costes: mantenerlo con un alto rendimiento hace algo más lento su desarrollo, por lo que no es el más avanzado en cuanto a prestaciones y compatibilidad con estándares. MySQL carece de características que muchos otros SGBD poseen. Pero no se debe olvidar que está en continuo desarrollo, por lo que futuras versiones incluirán nuevas características. Por supuesto, para MySQL es más importante la eficiencia que incluir prestaciones sólo por competir o satisfacer a algunos usuarios.

Bibliografía

DuBois, P. (2003). *MySQL Second Edition: The definitive guide to using, programming, and administering MySQL 4 databases* Indianapolis: Developer's Library.

Manual de MySQL de la distribución (accesible en: http://dev.mysql.com/doc/).

Silberschatz, A.; Korth, H.; Sudarshan, S. (2002) *Fundamentos de Bases de Datos* (4.ª ed.). Madrid: McGraw Hill.

Bases de datos en PostgreSQL

Marc Gibert Ginestà
Oscar Pérez Mora

UOC
Formación de Posgrado

Índice

Introducción

PostgreSQL es un gestor de bases de datos orientadas a objetos (SGBDOO o ORDBMS en sus siglas en inglés) muy conocido y usado en entornos de software libre porque cumple los estándares SQL92 y SQL99, y también por el conjunto de funcionalidades avanzadas que soporta, lo que lo sitúa al mismo o a un mejor nivel que muchos SGBD comerciales.

El origen de PostgreSQL se sitúa en el gestor de bases de datos POSTGRES desarrollado en la Universidad de Berkeley y que se abandonó en favor de PostgreSQL a partir de 1994. Ya entonces, contaba con prestaciones que lo hacían único en el mercado y que otros gestores de bases de datos comerciales han ido añadiendo durante este tiempo.

PostgreSQL se distribuye bajo licencia BSD, lo que permite su uso, redistribución, modificación con la única restricción de mantener el *copyright* del software a sus autores, en concreto el PostgreSQL Global Development Group y la Universidad de California.

PostgreSQL puede funcionar en múltiples plataformas (en general, en todas las modernas basadas en Unix) y, a partir de la próxima versión 8.0 (actualmente en su segunda beta), también en Windows de forma nativa. Para las versiones anteriores existen versiones binarias para este sistema operativo, pero no tienen respaldo oficial.

Para el seguimiento de los ejemplos y la realización de las actividades, es imprescindible disponer de los datos de acceso del usuario administrador del gestor de bases de datos. Aunque en algunos de ellos los privilegios necesarios serán menores, para los capítulos que tratan la administración del SGBDOO será imprescindible disponer de las credenciales de administrador.

Las sentencias o comandos escritos por el usuario estarán en fuente monoespaciada, y las palabras que tienen un significado especial en PostgreSQL estarán en **negrita**. Es importante hacer notar que estas últimas no siempre son palabras reservadas, sino comandos o sentencias de psql (el cliente interactivo de PostgreSQL).

La versión de PostgreSQL que se ha utilizado durante la redacción de este material, y en los ejemplos, es la 7.4, la última versión estable en ese momento, aunque no habrá ningún problema en ejecutarlos en versiones anteriores, hasta la 7.0.

Objetivos

El objetivo principal de esta unidad es conocer el gestor de bases de datos relacionales con soporte para objetos PostgreSQL, y comentar tanto sus características comunes a otros gestores de bases de datos, como las que lo distinguen de sus competidores de código abierto.

Además, se ofrece la oportunidad de aplicar los conocimientos aprendidos en el módulo referido a SQL sobre un sistema gestor de base de datos real y examinar las diferencias entre el estándar y la implementación que hace de él el SGBD.

Por último, se presentan las tareas de administración del SGBD más habituales que un usuario debe llevar a cabo como administrador de Post-greSQL.

1. Características de PostgreSQL

En este apartado comentaremos las características más relevantes de este SGBD con soporte para objetos, tanto sus prestaciones más interesantes o destacadas, como las limitaciones en su diseño o en implementación de los estándares SQL.

También es interesante conocer un poco su historia, ya que tanto por las personas que han participado en su desarrollo como por su relación con otros gestores de bases de datos, nos ayudará a tener una mejor visión de la evolución del mismo.

1.1. Breve historia

La historia de PostgreSQL se inicia en 1986 con un proyecto del profesor Michael Stonebraker y un equipo de desarrolladores de la Universidad Berkeley (California), cuyo nombre original era POSTGRES. En su diseño se incluyeron algunos conceptos avanzados en bases de datos y soporte parcial a la orientación a objetos.

POSTGRES fue comercializado por Illustra, una empresa que posteriormente formó parte de Informix (que comercializaba el conocido SGBD del mismo nombre, recientemente absorbida por IBM y su DB/2). Llegó un momento en que mantener el proyecto absorbía demasiado tiempo a los investigadores y académicos, por lo que en 1993 se liberó la versión 4.5 y oficialmente se dio por terminado el proyecto.

En 1994, Andrew Yu y Jolly Chen incluyeron SQL en Postgres para posteriormente liberar su código en la web con el nombre de Postgres95. El proyecto incluía múltiples cambios al código original que mejoraban su rendimiento y legibilidad.

En 1996 el nombre cambió a PostgreSQL retomando la secuencia original de versiones, por lo que se liberó la versión 6.0. En el año 2004 la última versión estable oficial es la 7.4.6, mientras que la versión 8.0 está ya en fase final de estabilización.

1.2. Prestaciones

PostgreSQL destaca por su amplísima lista de prestaciones que lo hacen capaz de competir con cualquier SGBD comercial:

Los desarrolladores de proyectos basados en software libre tienen muy en cuenta PostgreSQL cuando los requerimientos de un proyecto exigen prestaciones de alto nivel.

- Está desarrollado en C, con herramientas como Yacc y Lex.

- La API de acceso al SGBD se encuentra disponible en C, C++, Java, Perl, PHP, Python y TCL, entre otros.

- Cuenta con un rico conjunto de tipos de datos, permitiendo además su extensión mediante tipos y operadores definidos y programados por el usuario.

- Su administración se basa en usuarios y privilegios.

- Sus opciones de conectividad abarcan TCP/IP, *sockets* Unix y *sockets* NT, además de soportar completamente ODBC.

- Los mensajes de error pueden estar en español y hacer ordenaciones correctas con palabras acentuadas o con la letra 'ñ'.

- Es altamente confiable en cuanto a estabilidad se refiere.

- Puede extenderse con librerías externas para soportar encriptación, búsquedas por similitud fonética (soundex), etc.

- Control de concurrencia multi-versión, lo que mejora sensiblemente las operaciones de bloqueo y transacciones en sistemas multi-usuario.

- Soporte para vistas, claves foráneas, integridad referencial, disparadores, procedimientos almacenados, subconsultas y casi todos los tipos y operadores soportados en SQL92 y SQL99.

- Implementación de algunas extensiones de orientación a objetos. En PostgreSQL es posible definir un nuevo tipo de tabla a partir de otra previamente definida.

1.3. Limitaciones

Las limitaciones de este tipo de gestores de bases de datos suelen identificarse muy fácilmente analizando las prestaciones que tienen previstas para las próximas versiones. Encontramos lo siguiente:

- Puntos de recuperación dentro de transacciones. Actualmente, las transacciones abortan completamente si se encuentra un fallo durante su ejecución. La definición de puntos de recuperación permitirá recuperar mejor transacciones complejas.

- No soporta *tablespaces* para definir dónde almacenar la base de datos, el esquema, los índices, etc.

- El soporte a orientación a objetos es una simple extensión que ofrece prestaciones como la herencia, no un soporte completo.

2. Introducción a la orientación a objetos

Dado que PostgreSQL incluye extensiones de orientación a objetos (aunque no es, como ya hemos comentado, un SGBDOO completo), es interesante repasar algunos de los conceptos relacionados con este paradigma de programación y estructuración de datos.

2.1. El modelo orientado a objetos

En 1967, el lenguaje de programación Simula aplicaba algunas ideas para modelar aspectos de la realidad de forma mucho más directa que los métodos tradicionales. Desde entonces, la orientación a objetos (OO) ha adquirido cada vez mayor popularidad al demostrar sus ventajas, entre las cuales:

- Permite un modelado más "natural" de la realidad.
- Facilita la reutilización de componentes de software.
- Ofrece mecanismos de abstracción para mantener controlable la construcción de sistemas complejos.

En el mercado aparecen constantemente herramientas y lenguajes de programación autodenominados orientados a objetos y los ya existentes evolucionan rápidamente incluyendo nuevas características de OO. De la misma manera, se han desarrollado múltiples métodos y metodologías bajo este enfoque, cada una con aportaciones propias que llegan, en ocasiones, a resultar contradictorias entre sí. Se ha logrado la creación de un lenguaje unificado para el modelado, llamado precisamente UML (*unified modeling language*). La intención de UML es ser independiente de cualquier metodología y es precisamente esta independencia la que lo hace importante para la comunicación entre desarrolladores, ya que las metodologías son muchas y están en constante evolución.

Lamentablemente, a pesar de los muchos esfuerzos y de importantes avances, las ciencias de la computación no han creado aún una definición de **modelo de objetos** como tal. En un panorama como éste, es indispensable, al menos, la existencia de un *modelo informal de objetos* que oriente la evolución de la tecnología y que tenga la aprobación de los expertos en la materia. Un modelo así permitiría su estudio consistente por parte de los profesionales de las tecnologías de la información, facilitaría la creación de mejores lenguajes y herramientas y, lo que es más importante, definiría los estándares para una metodología de desarrollo consistente y aplicable.

Sin embargo, este modelo no existe, lo que provoca inconsistencias incluso en el tratamiento de los principios y conceptos básicos de la OO. Por eso, es frecuente encontrar errores graves en el desarrollo de sistemas OO y, lo que es aún peor, se implementan soluciones de dudosa validez en herramientas de desarrollo que se dicen orientadas a objetos.

Aun sin haber alcanzado la madurez, la orientación a objetos es el paradigma que mejor permite solucionar los muchos y variados problemas que existen en el desarrollo de software. En los próximos apartados analizaremos los conceptos básicos de este modelo para identificar algunos de los problemas que aún debe resolver, lo que facilitará la comprensión y evaluación de métodos y herramientas OO.

2.2. Objetos: clase frente a instancia

Los **objetos** son abstracciones que realizamos del mundo que nos rodea y que identificamos por sus propiedades. Para la OO todo es un objeto.

Cada objeto tiene una existencia un tanto independiente de los demás objetos; es decir, tiene **identidad** propia. Aunque dos objetos tengan exactamente los mismos valores, no por eso serán el mismo objeto, seguirán siendo entidades diferentes. En los modelos OO, la identidad se representa con el identificador de objeto, IDO (OID en inglés, de *object identifier*). Teóricamente, el IDO de un objeto es único e irrepetible en el tiempo y el espacio.

Los IDO son el mecanismo que permite hacer referencia a un objeto desde otro. De esta manera las referencias tejen las relaciones entre objetos.

Función del IDO

El IDO permite que dos objetos idénticos puedan diferenciarse, no es importante que el usuario conozca los IDO, lo importante es que los diferencie el sistema.

Todos los objetos que comparten las mismas propiedades se dice que pertenecen a la misma **clase**. En los modelos OO, las clases le roban el papel central a los objetos, ya que es a través de ellas como se definen las propiedades de éstos y además se utilizan como plantillas para crear objetos.

Elementos fundamentales en OO

objeto	clase

Al crear un objeto utilizando la definición dada por una clase, obtenemos un valor para él, es lo que se llama una **instancia** del objeto. Durante la ejecución de los programas se trabaja con instancias. Como concepto, la instancia es equivalente a una tupla (fila) concreta en una tabla de una base de datos.

2.3. Propiedades: atributo frente a operación

Las propiedades de los objetos pueden ser de dos tipos, dinámicas y estáticas. Un **atributo** representa una propiedad estática de un objeto (color, coste, edad, etc.). Una **operación** representa una propiedad dinámica; es decir, una transformación sobre un atributo o una acción que puede realizar.

El conjunto de valores de los atributos en un momento dado se conoce como **estado** del objeto. Los operadores actúan sobre el objeto cambiando su estado. La secuencia de estados por la que pasa un objeto al ejecutar operaciones definen su **comportamiento**.

La posibilidad de definir comportamientos complejos es lo que hace diferente la OO.

Propiedades de los objetos

objeto (instancia)	clase
atributos (estado)	atributos
operaciones (comportamiento)	operaciones

2.4. Encapsulamiento: implementación frente a interfaz

La estructura interna de los objetos debe estar oculta al usuario de un objeto, no necesita conocerla para interactuar con él. Los objetos se conciben como una cápsula cuyo interior está oculto y no puede ser alterado directamente desde el exterior.

A la estructura interna de un objeto se la denomina **implementación** y a la parte visible, la que se presenta al exterior, **interfaz**. La interfaz se define por sus atributos y operaciones.

La implementación de una operación se conoce como **método**. La implementación de un atributo se realiza generalmente con **variables de instancia**.

Encapsulamiento

Clase	
implementación	**interfaz**
variables	atributos
métodos	operaciones

Los tipos de datos abstractos

Los tipos de datos abstractos (TDA) obedecen al mismo principio de independencia de la implementación. La diferencia respecto a los objetos es que éstos incluyen los datos y las operaciones en la misma cápsula.

El encapsulamiento comporta las siguientes ventajas:

- La modificación interna (de la implementación) de un objeto para corregirlo o mejorarlo no afecta a sus usuarios.

- La dificultad inherente a la modificación de la implementación de un objeto sea independiente del tamaño total del sistema. Esto permite que los sistemas evolucionen con mayor facilidad.

- La simplificación en el uso del objeto al ocultar los detalles de su funcionamiento y presentarlo en términos de sus propiedades. Al elevar el nivel de abstracción se disminuye el nivel de complejidad de un sistema. Es posible modelar sistemas de mayor tamaño con menor esfuerzo.

- Constituye un mecanismo de integridad. La dispersión de un fallo a través de todo el sistema es menor, puesto que al presentar una división entre interfaz e implementación, los fallos internos de un objeto encuentran una barrera en el encapsulamiento antes de propagarse al resto del sistema.

- Permite la sustitución de objetos con la misma interfaz y diferente implementación. Esto permite modelar sistemas de mayor tamaño con menor esfuerzo.

Estas características del encapsulamiento han contribuido en gran medida a la buena reputación de la OO.

Paradójicamente, el encapsulamiento, a pesar de ser uno de los conceptos básicos en la OO, no siempre se interpreta y se aplica correctamente. Especialmente en lo referente a encapsulamiento de atributos.

Diferenciemos operación y método a través de un ejemplo.

Consideremos tres objetos: polígono, círculo y punto.

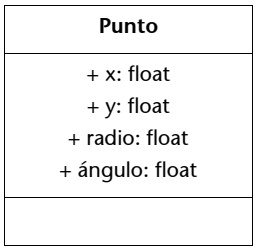
A los tres se les solicita la operación de imprimir. En esta situación, tenemos que:

- La operación solicitada es la misma, porque el significado del resultado es el mismo.
- Cada objeto ejecuta la operación de forma diferente; es decir, con un método diferente.
- Cada objeto, internamente, puede tener más de un método y selecciona el más apropiado según las circunstancias.

Las operaciones no son exclusivas de los tipos de objeto, los métodos sí. Una operación especifica "qué" hacer y un método "cómo" hacerlo. Esta diferencia permite tener múltiples métodos para una misma operación.

Veamos ahora la diferencia entre atributos y variables de instancia, que puede parecer más sutil.

Un atributo es la vista externa de una propiedad estática de un objeto. La representación interna puede variar, los atributos pueden implementarse también con métodos. Tomemos como ejemplo el objeto punto con los atributos que se muestran a continuación:



Los atributos de un punto pueden definirse en coordenadas angulares o rectangulares; en este caso, es posible conocer ambas representaciones. En la implementación de estos atributos, dos pueden ser variables de instancia y los otros dos se implementan como métodos, que se calculan a través de los primeros.

Desde el exterior no debe ser posible conocer la representación elegida internamente. Puede cambiarse la implementación de los atributos sin alterar la interfaz. En algunos casos puede incluso permitirse al sistema la elección de la representación interna de un atributo del mismo modo que una operación elige entre varios métodos disponibles.

2.4.1. Atributo frente a variable de instancia

Un atributo especifica una cualidad de un objeto; una variable de instancia especifica cómo se almacenan los valores para esa cualidad.

Los atributos

Un atributo puede ser almacenado en una variable o calculado por un método.

Consideremos tres objetos, nombre, foto, vídeo, de los que necesitamos conocer el tamaño y prever, así, el espacio necesario para almacenarlos en disco.

En esta situación tenemos que:

- El atributo es el mismo, porque su lectura tiene el mismo significado.
- Cada objeto implementa el atributo de manera diferente. Sin importar la implementación, externamente todos los atributos entregan el mismo tipo de valor. Por ejemplo:
  - El nombre puede utilizar un byte como variable de instancia, porque el tamaño de un nombre no puede ser mayor que 255 caracteres, o se puede implementar un método que calcule el tamaño en tiempo de ejecución.
  - La foto utilizará dos o cuatro bytes.
  - El vídeo puede almacenar el valor de tamaño en múltiplos de K.
- Cada objeto puede tener implementaciones alternativas que se adapten a las circunstancias.

Implementación del encapsulamiento

Clase	
implementación	**interfaz**
variables	operaciones de lectura/escritura
métodos	operaciones

Lamentablemente, los lenguajes de programación comúnmente utilizados no implementan mecanismos adecuados para el encapsulamiento de los atributos, llegando, incluso, a permitir el acceso público a variables de instancia. A continuación, analizaremos las graves consecuencias de este hecho.

Acceder a un atributo es, en realidad, una operación que puede ser de lectura o de escritura. Por este motivo, frecuentemente se define el encapsulamiento como la ocultación de todas las variables permitiendo el acceso del exterior sólo para operaciones. Cuando el lenguaje de programación no ofrece independencia de la implementación en los atributos, se deben definir una variable de instancia y dos métodos por cada atributo: *LeerAtributo* y *EscribirAtributo*.

Las bases de datos relacionales tienen perfectamente diferenciada la interfaz de la implementación en sus tipos de datos: la forma de almacenarlos es completamente independiente de la forma de consultarlos o guardarlos. No se conciben las operaciones como internas a los objetos.

El encapsulamiento ha sido considerado como un principio central de la orientación a objetos y atentar contra él significa para muchos romper con sus reglas fundamentales. Sin embargo, las bases de datos orientadas a objetos tienen entre sus funciones la realización de consultas, que necesita acceder a los atributos de los objetos. Dado que los objetos se implementan con variables, al accederlos se rompe el encapsulamiento.

La mayoría de los lenguajes orientados a objetos permiten romper el encapsulamiento de forma parcial, declarando variables como públicas. El encapsulamiento, en estos casos, se proporciona como un mecanismo opcional, ya que el usuario puede declarar todas las variables públicas y, por lo tanto, accesibles directamente.

Otros lenguajes implementan operaciones de lectura/escritura que permiten acceder a las variables sin romper el encapsulamiento.

2.5. Herencia: jerarquía de clases

La herencia se define como el mecanismo mediante el cual se utiliza la definición de una clase llamada “padre”, para definir una nueva clase llamada “hija” que puede heredar sus atributos y operaciones.

A las clases "hijo" también se les conoce como **subclases**, y a las clases "padre" como **superclases**. La relación de herencia entre clases genera lo que se llama **jerarquía de clases**.

Hablamos de herencia de tipo cuando la subclase hereda la interfaz de una superclase; es decir, los atributos y las operaciones. Hablamos de herencia estructural cuando la subclase hereda la implementación de la superclase; es decir, las variables de instancia y los métodos.

La herencia de tipo

En la herencia de tipo lo que hereda la subclase son los atributos de la superclase, pero no necesariamente su implementación, puesto que puede volver a implementarlos.

La herencia de tipo define relaciones **es-un** entre clases, donde la clase "hijo" tiene todas las propiedades del "padre", pero el "padre" no tiene todas las propiedades del "hijo".

Ejemplo

Un gato es-un animal. Todas las propiedades de la clase "animal" las tiene la clase "gato". Pero un animal no-es necesariamente un gato. Todas las propiedades de gato no las tienen todos los animales.

Consideremos una referencia mascota que es de tipo animal, en algún lenguaje de programación.

```
mimascota: Animal;
```

Puede hacer referencia a objetos de tipo animal, o tipos derivados de éste, como perro, gato o canario, por ejemplo.

```
mimascota = new Canario;
```

Se construye un nuevo canario y se hace referencia a él como mascota.

La propiedad de sustituir objetos que descienden del mismo padre se conoce como polimorfismo, y es un mecanismo muy importante de reutilización en la OO.

La referencia al tipo animal es una referencia polimorfa, ya que puede referirse a tipos derivados de animal. A través de una referencia polimorfa se pueden solicitar operaciones sin conocer el tipo exacto.

```
mimascota.comer();
```

La operación comer tiene el mismo significado para todos los animales. Como ya hemos comentado, cada uno utilizará un método distinto para ejecutar la operación.

Para conocer el tipo exacto del objeto en cuestión, se utiliza el operador de **información de tipo**. De este modo puede accederse a las propiedades específicas de un tipo de objeto que no están en los demás tipos.

En este ejemplo llamamos al operador información de tipo, **instancia-de**.

```
if (mimascota instancia-de Canario)
  mimascota.cantar();
```

Si la mascota es una instancia del tipo Canario entonces se le solicitará cantar, que es una propiedad que no tienen todas las mascotas.

Una clase puede heredar las propiedades de dos superclases mediante lo que se conoce como **herencia múltiple**.

En una herencia múltiple, puede ocurrir que en ambas superclases existan propiedades con los mismos nombres, situación que se denomina **colisión de nombres**. A continuación, se relacionan los posibles casos de colisión de nombres en la herencia de tipo:

- Los nombres son iguales porque se refieren a la misma propiedad (ya hemos visto ejemplos de ello: la operación imprimir y el atributo tamaño). En este caso no hay conflicto porque el significado está claro: es la misma propiedad, sólo hay que definir una implementación adecuada.

- Los nombres son iguales pero tienen significados diferentes. Esta situación es posible porque el modelado es una tarea subjetiva y se soluciona cambiando los nombres de las propiedades heredadas que tengan conflicto.

La herencia múltiple no comporta problemas para la herencia de tipo, puesto que no pretende la reutilización de código, sino el control conceptual de la complejidad de los sistemas mediante esquemas de clasificación.

Por lo que respecta a la herencia estructural, que, recordemos, consiste en que la subclase hereda las variables de instancia y los métodos de la superclase –es decir, la implementación–, la cosa cambia.

Para entender mejor la herencia estructural, diremos informalmente que representa una relación **funciona-como**. Por ejemplo, se puede utilizar para definir un avión tomando como superclase ave, de esta manera la capacidad de volar del ave queda implementada en el avión. Un avión no **es-un** ave, pero podemos decir que **funciona-como** ave.

Al aplicar la herencia de esta manera se dificulta la utilización del polimorfismo: aunque un objeto funcione internamente como otro, no se garantiza que externamente pueda tomar su lugar porque **funciona-como**.

El objetivo de la herencia estructural es la reutilización de código, aunque en algunos casos, como el ejemplo anterior, pueda hacer conceptualmente más complejos los sistemas.

Siempre que es posible aplicar la herencia de tipo, puede aplicarse la herencia estructural, por lo que la mayoría de los lenguajes de programación no hacen distinción entre los dos tipos de herencia.

Ejemplo

Si un canario **es-un** animal, entonces un canario **funciona-como** animal, más otras propiedades especificas de canario.

Los lenguajes de programación comúnmente no hacen distinción entre la herencia estructural y la herencia de tipo.

La **herencia estructural múltiple** permite heredar variables y métodos de varias superclases, pero surgen problemas que no son fáciles de resolver, especialmente con las variables de instancia.

Para resolver el conflicto de una variable de instancia duplicada, se puede optar por las siguientes soluciones:

- Cambiar los nombres, lo que puede provocar conflictos en los métodos que las utilizan.

- Eliminar una de las variables. Pero puede pasar que realicen alguna función independiente, en cuyo caso, sería un error eliminar una.

- No permitir herencia múltiple cuando hay variables duplicadas.

Como se puede observar, no es fácil solucionar conflictos entre variables de instancia, por ello muchos lenguajes optan por diversos mecanismos incluyendo la prohibición de la herencia múltiple.

2.5.1. Tipo y clase

Tenemos que advertir que la mayoría de lenguajes de programación no diferencian los conceptos de tipo y clase y que la diferencia que establecen algunos autores no es demasiado clara. De todas maneras, la tendencia sería definir dichos conceptos como sigue:

- Un tipo es un conjunto de objetos que comparten la misma interfaz.

- Una clase es un conjunto de objetos que comparten la misma implementación.

Una solución que se aplica es incluir en el lenguaje el concepto de **interfaz** que define solamente las operaciones de una clase, pero no ofrece alternativas para los atributos. Sin embargo, con la diferenciación entre clases e interfaces no se logra la diferenciación entre los dos tipos de herencia, pues las clases se utilizan para representar relaciones **es-un**.

2.6. Agregación: jerarquía de objetos

Los objetos son, por naturaleza, complejos; es decir, están compuestos de objetos más pequeños. Un sistema de información debe reflejar esta propiedad de los objetos de forma natural. En una base de datos relacional, un objeto complejo debe ser descompuesto en sus partes más simples para ser almacenado. Al extraerlo, es necesario ensamblar cada una de sus partes.

Ejemplo

Un automóvil está compuesto de carrocería, motor, ruedas, etc.

Por este motivo el modelo relacional comporta problemas cuando se utiliza en aplicaciones como el CAD, donde los objetos que se procesan son muy complejos.

Las bases de datos de objetos deben proporcionar la facilidad de obtener un objeto complejo en una sola consulta de forma transparente. En este caso, los

apuntadores son un mecanismo excelente para representar composición, ya que permiten acceder rápidamente a las partes componentes de un objeto, sin importar su lugar de almacenamiento.

Las bases de datos requieren independencia de la aplicación, lo que provoca un conflicto conceptual cuando se trabaja con objetos compuestos: las bases de datos deben almacenar información independiente de la aplicación para que nuevas aplicaciones puedan hacer diferentes interpretaciones de la información original; pero con los objetos compuestos esto no es tan sencillo, puesto que suelen tener una sola interpretación, o mejor dicho, una sola manera de ser consultado en una base de datos.

2.7. Persistencia

La persistencia se define como la capacidad de un objeto para sobrevivir al tiempo de ejecución de un programa. Para implementarla, se utiliza el almacenamiento secundario.

Se han propuesto varios mecanismos para implementar la persistencia en los lenguajes de programación, entre los que podemos destacar los siguientes:

- Archivos planos. Se crean archivos para almacenar los objetos en el formato deseado por el programador. Los objetos se cargan al abrir el programa y se guardan al finalizar. Ésta es la opción más accesible para todos los lenguajes de programación.

- Bases de datos relacionales. Los objetos son mapeados a tablas, un módulo del programa se encarga de hacer las transformaciones objeto-relacionales. Este enfoque consume mucho tiempo al realizar el mapeo. Existen algunas herramientas que realizan mapeos semiautomáticos.

- Bases de objetos. Los objetos son almacenados de forma natural en una base de objetos, y la consulta y recuperación es administrada por el gestor, de esta forma las aplicaciones no necesitan saber nada sobre los detalles de implementación.

- Persistencia transparente. Los objetos son almacenados y recuperados por el sistema cuando éste lo cree conveniente, sin que el usuario deba hacer ninguna solicitud explícita de consulta, actualización o recuperación de información a una base de objetos. No se requiere, por lo tanto, otro lenguaje para interactuar con las bases de datos.

2.8. PostgreSQL y la orientación a objetos

El argumento a favor de las bases de datos objeto-relacionales sostiene que permite realizar una migración gradual de sistemas relacionales a los orientados

a objetos y, en algunas circunstancias, coexistir ambos tipos de aplicaciones durante algún tiempo.

El problema de este enfoque es que no es fácil lograr la coexistencia de dos modelos de datos diferentes como son la orientación a objetos y el modelo relacional. Es necesario equilibrar de alguna manera los conceptos de uno y otro modelo sin que entren en conflicto.

Uno de los conceptos fundamentales en la orientación a objetos es el concepto de clase. Existen dos enfoques para asociar el concepto de clase con el modelo relacional:

1.er enfoque: las clases definen tipos de tablas
2.o enfoque: las clases definen tipos de columnas

Dado que en el modelo relacional las columnas están definidas por tipos de datos, lo más natural es hacer corresponder las columnas con las clases.

	1.er enfoque	2.o enfoque
Los objetos son	valores	tuplas
Las clases son	dominios	tablas

PostgreSQL implementa los objetos como tuplas y las clases como tablas. Aunque también es posible definir nuevos tipos de datos mediante los mecanismos de extensión.

Los mecanismos de extensión

No es habitual que el usuario utilice los mecanismos de extensión pues se consideran mecanismos avanzados.

Dado que las tablas son clases, pueden definirse como herencia de otras. Las tablas derivadas son polimorfas y heredan todos los atributos (columnas) de la tabla *padre* (incluida su clave primaria). Si no se manejan con precaución, las tablas polimorfas pueden conducir a errores de integridad al duplicar claves primarias. PostgreSQL soporta algunas extensiones del lenguaje SQL para crear y gestionar este tipo de tablas.

Veremos estos conceptos más en detalle en el subapartado 4.2 de esta unidad didáctica.

3. Acceso a un servidor PostgreSQL

3.1. La conexión con el servidor

Antes de intentar conectarse con el servidor, debemos asegurarnos de que está funcionando y que admite conexiones, locales (el SGBD se está ejecutando en la misma máquina que intenta la conexión) o remotas.

Una vez comprobado el correcto funcionamiento del servidor, debemos disponer de las credenciales necesarias para la conexión. Para simplificar, supondremos que disponemos de las credenciales* del administrador de la base de datos (normalmente, usuario PostgreSQL y su contraseña).

*** Distintos tipos de credenciales permiten distintos niveles de acceso.**

En el apartado que concierne a la administración de PostgreSQL se comenta detalladamente los aspectos relacionados con el sistema de usuarios, contraseñas y privilegios del SGBD.

3.2. El cliente psql

Para conectarse con un servidor, se requiere, obviamente, un programa cliente. Con la distribución de PostgreSQL se incluye un cliente, psql, fácil de utilizar, que permite la introducción interactiva de comandos en modo texto.

El siguiente paso es conocer el nombre de una base de datos residente en el servidor. El siguiente comando permite conocer las bases de datos residentes en el servidor:

```
~$ psql -l
List of databases
Name       | Owner    | Encoding
-----------+----------+-----------
demo       | postgres | SQL_ASCII
template0  | postgres | SQL_ASCII
template1  | postgres | SQL_ASCII
(3 rows)
~$
```

Para realizar una conexión, se requieren los siguientes datos:

- Servidor. Si no se especifica, se utiliza *localhost*.
- Usuario. Si no se especifica, se utiliza el nombre de usuario Unix que ejecuta psql.
- Base de datos.

Ejemplos del uso de psql para conectarse con un servidor de bases de datos

```
~$ psql -d demo
~$ psql demo
```

Las dos formas anteriores ejecutan psql con la base de datos *demo*.

```
~$ psql -d demo -U yo
~$ psql demo yo
~$ psql -h servidor.org -U usuario -d basedatos
```

A partir del fragmento anterior, el cliente psql mostrará algo similar a lo siguiente:

```
Welcome to psql, the PostgreSQL interactive terminal.
Type: \copyright for distribution terms
\h for help with SQL commands
\? for help on internal slash commands
\g or terminate with semicolon to execute query
\q to quit
demo=#
```

El símbolo '#', que significa que psql está listo para leer la entrada del usuario.

Notación

Hemos utilizado los dos comandos de ayuda que ofrece el lenguaje:

- **h** Explica la sintaxis de sentencias SQL.
- **?** Muestra los comandos internos disponibles.

Para salir de ayuda, se presiona la tecla 'q'.

Las sentencias SQL se envían directamente al servidor para su interpretación, los comandos internos tienen la forma *comando* y ofrecen opciones que no están incluidas en SQL y son interpretadas internamente por psql.

Para terminar la sesión con psql, utilizamos el comando **\q** o podemos presionar Ctrl-D.

3.3. Introducción de sentencias

Las sentencias SQL que escribamos en el cliente deberán terminar con ';' o bien con '\g':

```
demo=# select user;
   current_user
--------------
   postgres
(1 row)
demo=#
```

Cuando un comando ocupa más de una línea, el indicador cambia de forma y va señalando el elemento que aún no se ha completado.

```
demo=# select
demo-# user\g
   current_user
--------------
   postgres
(1 row)
demo=#
```

Indicadores de PostgreSQL	
Indicador	**Significado**
=#	Espera una nueva sentencia
-#	La sentencia aún no se ha terminado con ";" o \g
"#	Una cadena en comillas dobles no se ha cerrado
'#	Una cadena en comillas simples no se ha cerrado
(#	Un paréntesis no se ha cerrado

El cliente psql almacena la sentencia hasta que se le da la orden de enviarla al SGBD. Para visualizar el contenido del *buffer* donde ha almacenado la sentencia, disponemos de la orden '\p':

```
demo=> SELECT
demo-> 2 * 10 + 1
demo-> \p
   SELECT
   2 * 10 + 1
demo-> \g
   ?column?
   ----------
      21
   (1 row)
demo=>
```

El cliente también dispone de una orden que permite borrar completamente el *buffer* para empezar de nuevo con la sentencia:

```
demo=# select 'Hola'\r
Query buffer reset (cleared).
demo=#
```

3.3.1. Expresiones y variables

El cliente psql dispone de multitud de prestaciones avanzadas; entre ellas (como ya hemos comentado), el soporte para sustitución de variables similar al de los *shells* de Unix:

```
demo=>\set var1 demostracion
```

Esta sentencia crea la variable 'var1' y le asigna el valor 'demostración'. Para recuperar el valor de la variable, simplemente deberemos incluirla precedida de ':' en cualquier sentencia o bien ver su valor mediante la orden 'echo':

```
demo=# \echo :var1
demostracion
demo-#
```

De la misma forma, psql define algunas variables especiales que pueden ser útiles para conocer detalles del servidor al que estamos conectados:

```
demo=# \echo :DBNAME :ENCODING :HOST :PORT :USER;
demo LATIN9 localhost 5432 postgres
demo=#
```

El uso de variables puede ayudar en la ejecución de sentencias SQL:

```
demo=> \set var2 'mi_tabla'
demo=> SELECT * FROM :var2;
```

Se debe ser muy cuidadoso con el uso de las comillas y también es importante tener en cuenta que dentro de cadenas de caracteres no se sustituyen variables.

3.4. Proceso por lotes y formatos de salida

Además de ser interactivo, psql puede procesar comandos por lotes almacenados en un archivo del sistema operativo mediante la siguiente sintaxis:

```
$ psql demo -f demo.psql
```

Aunque el siguiente comando también funciona en el mismo sentido, no es recomendable usarlo porque de este modo, psql no muestra información de depuración importante, como los números de línea donde se localizan los errores, en caso de haberlos:

```
$ psql demo < demo.psql
```

El propio intérprete psql nos proporciona mecanismos para almacenar en fichero el resultado de las sentencias:

- Especificando el fichero destino* directamente al finalizar una sentencia:

*** Hemos almacenado el resultado en el fichero *'/tmp/a.txt'*.**

```
demo=# select user \g /tmp/a.txt
```

- Mediante una *pipe* enviamos la salida a un comando Unix:

```
demo=# select user \g | cat > /tmp/b.txt
```

Notación

A la orden '\o' se le debe especificar un fichero o bien un comando que irá recibiendo los resultados mediante una *pipe*.

Cuando se desee volver a la salida estándar STDOUT, simplemente se dará la orden '\o' sin ningún parámetro.

- Mediante la orden '\o' se puede indicar dónde debe ir la salida de las sentencias SQL que se ejecuten en adelante:

```
demo=# \o /tmp/sentencias.txt
demo=# select user;
demo=# select 1+1+4;
demo=# \o
demo=# select 1+1+4;
   ?column?
----------
   6
(1 row)
demo=#
```

- Se puede solicitar la ejecución de un solo comando y terminar inmediatamente mediante la siguiente forma:

```
$ psql -d demo -c "comando sql"
```

- Se puede especificar el formato de salida de los resultados de una sentencia. Por defecto, psql los muestra en forma tabular mediante texto. Para cambiarlo, se debe modificar el valor de la variable interna 'format' mediante la orden '\pset'. Veamos, en primer lugar, la especificación del formato de salida:

La salida del fichero

Al haber especificado que se quiere la salida en html, la podríamos redirigir a un fichero (ya hemos visto cómo hacerlo) y generar un archivo html que permitiese ver el resultado de la consulta mediante un navegador web convencional.

```
demo=# \pset format html
Output format is html.
demo=# select user;
<table border="1">
   <tr>
      <th align="center">current_user</th>
   </tr>
   <tr valign="top">
      <td align="left">postgres</td>
   </tr>
</table>
<p>(1 row)<br />
</p>
demo=#
```

Hay otros formatos de salida, como 'aligned', 'unaligned', 'html' y 'latex'. Por defecto, psql muestra el resultado en formato 'aligned'.

Tenemos también multitud de variables para ajustar los separadores entre columnas, el número de registros por página, el separador entre registros, título de la página html, etc. Veamos un ejemplo:

```
demo=# \pset format unaligned
Output format is unaligned.
demo=# \pset fieldsep ','
Field separator is ",".
demo=# select user, 1+2+3 as resultado;
current_user,resultado
postgres,6
(1 row)
demo=#
```

La salida de este fichero

Con esta configuración, y dirigiendo la salida a un fichero, generaríamos un fichero *CSV* listo para ser leído en una hoja de cálculo u otro programa de importación de datos.

Para poder realizar los ejemplos del resto del presente apartado, se debe procesar el contenido del fichero *demo.sql* tal como se transcribe a continuación.

Contenido del fichero *demo.psql*

```
--drop table productos;
--drop table proveedores;
--drop table precios;
--drop table ganancia;

create table productos (
   parte              varchar(20),
   tipo               varchar(20),
   especificación     varchar(20),
   psugerido          float(6),
   clave              serial,

   primary key(clave)
);

insert into productos (parte,tipo,especificación,psugerido) values
   ('Procesador','2 GHz','32 bits',null);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Procesador','2.4 GHz','32 bits',35);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Procesador','1.7 GHz','64 bits',205);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Procesador','3 GHz','64 bits',560);
insert into productos (parte,tipo,especificación,psugerido) values
   ('RAM','128MB','333 MHz',10);
insert into productos (parte,tipo,especificación,psugerido) values
   ('RAM','256MB','400 MHz',35);
```

```
insert into productos (parte,tipo,especificación,psugerido) values
   ('Disco Duro','80 GB','7200 rpm',60);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Disco Duro','120 GB','7200 rpm',78);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Disco Duro','200 GB','7200 rpm',110);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Disco Duro','40 GB','4200 rpm',null);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Monitor','1024x876','75 Hz',80);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Monitor','1024x876','60 Hz',67);

create table proveedores (
   empresa       varchar(20) not null,
   credito       bool,
   efectivo      bool,
   primary key   empresa)
);

insert into proveedores (empresa,efectivo) values ('Tecno-k', true );
insert into proveedores (empresa,credito) values ('Patito', true );
insert into proveedores (empresa,credito,efectivo) values
('Nacio-nal', true, true );

create table ganancia(
   venta      varchar(16),
   factor     decimal (4,2)
);

insert into ganancia values('Al por mayor',1.05);
insert into ganancia values('Al por menor',1.12);

create table precios (
   empresa                 varchar(20) not null,
   clave                   int not null,
   precio  float(6),

   foreign key (empresa)    references proveedores,
   foreign key (clave)      references productos
);

   insert into precios values ('Nacional',001,30.82);
   insert into precios values ('Nacional',002,32.73);
   insert into precios values ('Nacional',003,202.25);
   insert into precios values ('Nacional',005,9.76);
   insert into precios values ('Nacional',006,31.52);
   insert into precios values ('Nacional',007,58.41);
   insert into precios values ('Nacional',010,64.38);
   insert into precios values ('Patito',001,30.40);
   insert into precios values ('Patito',002,33.63);
   insert into precios values ('Patito',003,195.59);
   insert into precios values ('Patito',005,9.78);
```

```
insert into precios values ('Patito',006,32.44);
insert into precios values ('Patito',007,59.99);
insert into precios values ('Patito',010,62.02);
insert into precios values ('Tecno-k',003,198.34);
insert into precios values ('Tecno-k',005,9.27);
insert into precios values ('Tecno-k',006,34.85);
insert into precios values ('Tecno-k',007,59.95);
insert into precios values ('Tecno-k',010,61.22);
insert into precios values ('Tecno-k',012,62.29);
```

3.5. Usar bases de datos

La siguiente orden informa sobre las bases de datos actualmente en el SGBD.

```
demo=# \l
        List of databases
Name       |Owner     | Encoding
-----------+----------+----------
 demo      | postgres | LATIN9
 template0 | postgres | LATIN9
 template1 | postgres | LATIN9
(3 rows)
demo=#
```

La orden '\c' permite conectarse a una base de datos:

```
demo=# \c demo
You are now connected to database "demo".
demo=#
```

La consulta de la tabla que contiene la base de datos `demo` se realiza mediante la orden '\d':

```
demo=# \d
                List of relations
 Schema |        Name         |   Type   |  Owner
--------+---------------------+----------+----------
 public | ganancia            | table    | postgres
 public | precios             | table    | postgres
 public | productos           | table    | postgres
 public | productos_clave_seq | sequence | postgres
 public | proveedores         | table    | postgres
(5 rows)
```

La orden **\d** es útil para mostrar información sobre el SGBD: tablas, índices, objetos, variables, permisos, etc. Podéis obtener todas las variantes de esta sentencia introduciendo \? en el intérprete de comandos.

Consulta de las columnas de cada una de las tablas:

```
demo-# \d proveedores
          Table "public.proveedores"
Column    |Type                    | Modifiers
----------+------------------------+-----------
 empresa  | character varying(20) | not null
 credito  | boolean               |
 efectivo | boolean               |
Indexes:
    "proveedores_pkey" primary key, btree (empresa)
```

Para crear una nueva base de datos, usaremos la sentencia `create database`:

```
mysql> create database prueba;
```

Para eliminar una base de datos, usaremos la sentencia `drop database`:

```
mysql> drop database prueba;
```

4. Creación y manipulación de tablas

4.1. Creación de tablas

Una vez conectados a una base de datos, la sentencia SQL **create table** permite crear las tablas que necesitemos:

```
demo=#     create table persona (
demo(#     nombre varchar(30),
demo(#     direccion varchar(30)
demo(# );
CREATE
```

El comando **drop table** permite eliminar tablas:

```
demo=# drop table persona;
```

La tabla recién creada aparece ahora en la lista de tablas de la base de datos en uso:

```
demo=# \dt
List of relations
Name     |Type   |Owner
---------+-------+--------
 persona | table | quiron
(1 row)
```

Podemos consultar su descripción mediante el comando *\d tabla*:

```
demo=# \d persona
Table "persona"
Column     |Type                   | Modifiers
-----------+-----------------------+-----------
    nombre | character varying(30) |
 direccion | character varying(30) |
```

La tabla está lista para insertar en ella algunos registros.

```
demo=# insert into persona values ( 'Alejandro Magno' , 'Babilonia' );
INSERT 24756 1
demo=# insert into persona values ( 'Federico García Lorca ' , 'Granada 65' );
INSERT 24757 1
```

El número con el que responde el comando *insert* se refiere al OID del registro insertado.


Este aspecto se explicará en detalle más adelante.

Las consultas se realizan con la sentencia `SQL` **`select.`** En este caso solicitamos que nos muestre todas las columnas de los registros en la tabla *persona*:

```
demo=# select * from persona;
nombre                 |direccion
-----------------------+------------
 Alejandro Magno       | Babilonia
 Federico García Lorca | Granada 65
(2 rows)
demo=#
```

Las tablas creadas en PostgreSQL incluyen, por defecto, varias columnas ocultas que almacenan información acerca del identificador de transacción en que pueden estar implicadas, la localización física del registro dentro de la tabla (para localizarla muy rápidamente) y, los más importantes, el OID y el TABLEOID. Estas últimas columnas están definidas con un tipo de datos especial llamado identificador de objeto (OID) que se implementa como un entero positivo de 32 bits. Cuando se inserta un nuevo registro en una tabla se le asigna un número consecutivo como OID, y el TABLEOID de la tabla que le corresponde.

En la programación orientada a objetos, el concepto de OID es de vital importancia, ya que se refiere a la identidad propia del objeto, lo que lo diferencia de los demás objetos.

Para observar las columnas ocultas, debemos hacer referencia a ellas específicamente en el comando *select*:

```
demo=# select oid, tableoid, * from persona;
oid    |tableoid  |nombre                 |direccion
-------+----------+-----------------------+------------
 17242 |    17240 | Alejandro Magno       | Babilonia
 17243 |    17240 | Federico García Lorca | Granada 65
(2 rows)
demo=#
```

Estas columnas se implementan para servir de identificadores en la realización de enlaces desde otras tablas.

Ejemplo de la utilización de OID para enlazar dos tablas

Retomamos la tabla *persona* y construimos una nueva tabla para almacenar los teléfonos.

```
demo=# create table telefono (
demo(# tipo char(10),
demo(# numero varchar(16),
demo(# propietario oid
demo(# );
CREATE
```

La tabla *teléfono* incluye la columna *propietario* de tipo OID, que almacenará la referencia a los registros de la tabla *persona*. Agreguemos dos teléfonos a 'Alejandro Magno', para ello utilizamos su OID que es 17242:

```
demo=# insert into telefono values( 'móvil' , '12345678', 17242 );
demo=# insert into telefono values( 'casa' , '987654', 17242 );
```

Las dos tablas están vinculadas por el OID de persona.

```
demo=# select * from telefono;
tipo        |numero       | propietario
------------+-------------+-------------
      móvil |    12345678 |   17242
       casa |      987654 |   17242
(2 rows)
```

La operación que nos permite unir las dos tablas es *join*, que en este caso une *teléfono* y *persona*, utilizando para ello la igualdad de las columnas *telefono.propietario* y *persona.oid*:

```
demo=# select * from telefono join persona on (telefono.propietario = perso-na.oid);
tipo        |numero       | propietario |nombre           | direccion
------------+-------------+-------------+-----------------+-----------
      móvil |    12345678 |       17242 | Alejandro Magno | Babilonia
       casa |      987654 |       17242 | Alejandro Magno | Babilonia
(2rows)
```

Los OID de PostgreSQL presentan algunas deficiencias:

- Todos los OID de una base de datos se generan a partir de una única secuencia centralizada, lo que provoca que en bases de datos con mucha actividad de inserción y eliminación de registros, el contador de 4 bytes se desborde y pueda entregar OID ya entregados. Esto sucede, por supuesto, con bases de datos muy grandes.

- Las tablas enlazadas mediante OID no tienen ninguna ventaja al utilizar operadores de composición en términos de eficiencia respecto a una clave primaria convencional.

- Los OID no mejoran el rendimiento. Son, en realidad, una columna con un número entero como valor.

Los desarrolladores de PostgreSQL proponen la siguiente alternativa para usar OID de forma absolutamente segura:

- Crear una restricción de tabla para que el OID sea único, al menos en cada tabla. El SGBD irá incrementando secuencialmente el OID hasta encontrar uno sin usar.
- Usar la combinación OID - TABLEOID si se necesita un identificador único para un registro válido en toda la base de datos.

Por los motivos anteriores, no es recomendable el uso de OID hasta que nuevas versiones de PostgreSQL los corrijan. En caso de usarlos, conviene seguir las recomendaciones anteriores.

Es posible crear tablas que no incluyan la columna OID mediante la siguiente notación:

```
create table persona (
nombre varchar(30),
direccion varchar(30)
)without oids;
```

4.2. Herencia

PostgreSQL ofrece como característica particular la herencia entre tablas, que permite definir una tabla que herede de otra previamente definida, según la definición de herencia que hemos visto en capítulos anteriores.

Retomemos la tabla *persona* definida como sigue:

```
create table persona (
nombre varchar (30),
direccion varchar (30)
);
```

A partir de esta definición, creamos la tabla *estudiante* como derivada de *persona*:

```
create table estudiante (
demo(# carrera varchar(50),
demo(# grupo char,
demo(# grado int
demo(# ) inherits ( persona );
CREATE
```

En la tabla *estudiante* se definen las columnas *carrera*, *grupo* y *grado*, pero al solicitar información de la estructura de la tabla observamos que también incluye las columnas definidas en *persona*:

```
demo=# \d estudiante
Table "estudiante"
Column     |Type                   | Modifiers
-----------+-----------------------+-----------
    nombre | character varying(30) |
 direccion | character varying(30) |
   carrera | character varying(50) |
     grupo |          character(1) |
     grado |               integer |
```

En este caso, a la tabla *persona* la llamamos **padre** y a la tabla *estudiante*, **hija**.

Cada registro de la tabla *estudiante* contiene 5 valores porque tiene 5 columnas:

```
demo=# insert into estudiante values (
demo(# 'Juan' ,
demo(# 'Treboles 21',
demo(# 'Ingenieria en Computacion',
demo(# 'A',
demo(# 3
demo(# );
INSERT 24781 1
```

La herencia no sólo permite que la tabla *hija* contenga las columnas de la tabla *padre*, sino que establece una relación conceptual **es-un**.

La consulta del contenido de la tabla *estudiante* mostrará, por supuesto, un solo registro. Es decir, no se heredan los datos, únicamente los campos (atributos) del objeto:

```
demo=# select * from estudiante;
nombre  |direccion    |carrera                    |grupo  | grado
--------+-------------+---------------------------+-------+-------
   Juan | Treboles 21 | Ingenieria en Computacion |     A | 3
(1 row)
```

Además, la consulta de la tabla *persona* mostrará un nuevo registro:

```
demo=# select * from persona;
nombre                 | direccion
-----------------------+------------
  Federico Garca Lorca | Granada 65
       Alejandro Magno | Babilonia
                  Juan | Treboles 21
(3 rows)
```

El último registro mostrado es el que fue insertado en tabla *estudiante*, sin embargo la herencia define una relación conceptual en la que un *estudiante* **es-una** *persona*. Por lo tanto, al consultar cuántas personas están registradas en la base de datos, se incluye en el resultado a todos los estudiantes. Para consultar sólo a las personas que no son estudiantes, podemos utilizar el modificador ONLY:

```
demo=# select * from only persona;
nombre                 | direccion
-----------------------+------------
 Alejandro Magno       | Babilonia
 Federico García Lorca | Granada 65
(2 rows)
demo=#
```

No es posible borrar una tabla *padre* si no se borran primero las tablas *hijo*.

```
demo=# drop table persona;
NOTICE: table estudiante depende de table persona
ERROR: no se puede eliminar table persona porque otros objetos dependen de él
HINT: Use DROP ... CASCADE para eliminar además los objetos dependientes.
```

Como es lógico, al borrar la fila del nuevo estudiante que hemos insertado, se borra de las dos tablas. Tanto si lo borramos desde la tabla *persona*, como si lo borramos desde la tabla *estudiante*.

4.3. Herencia y OID

Los OID permiten que se diferencien los registros de todas las tablas, aunque sean heredadas: nuestro estudiante tendrá el mismo OID en las dos tablas, ya que se trata de única instancia de la clase estudiante:

```
demo=# select oid,* from persona ;
oid    |nombre                 | direccion
-------+-----------------------+------------
 17242 | Alejandro Magno       | Babilonia
 17243 | Federico García Lorca | Granada 65
 17247 | Juan                  | Treboles 21
(3 rows)
demo=# select oid,* from estudiante ;
oid    |nombre  |direccion    |carrera                    |grupo  | grado
-------+--------+-------------+---------------------------+-------+-------
 17247 | Juan   | Treboles 21 | Ingenieria en Computación | A     |     3
(1 row)
```

Dado que no se recomienda el uso de OID en bases muy grandes, y debe incluirse explícitamente en las consultas para examinar su valor, es conveniente utilizar una secuencia compartida para padres y todos sus descendientes si se requiere un identificador.

En PostgreSQL, una alternativa para no utilizar los OID es crear una columna de tipo **serial** en la tabla *padre*, así será heredada en la *hija*. El tipo **serial** define una secuencia de valores que se irá incrementando de forma automática, y por lo tanto constituye una buena forma de crear claves primarias, al igual que el tipo AUTO_INCREMENT en MySQL.

```
demo=# create table persona (
demo(# id serial,
demo(# nombre varchar (30),
demo(# direccion varchar(30)
demo(# ) without oids;
NOTICE: CREATE TABLE will create implicit sequence 'persona_id_seq' for SERIAL column 'persona.
NOTICE: CREATE TABLE / UNIQUE will create implicit index 'persona_id_key' for table 'persona'
CREATE
```

La columna *id* se define como un entero y se incrementará utilizando la función *nextval()* tal como nos indica la información de la columna:

```
demo=# \d persona
Table "persona"
Column   |Type                |Modifiers
---------+--------------------+-------------------------------------------
       id|            integer  |not null default nextval('"persona_id_seq"'::text)
   nombre|character varying(30)|
direccion|character varying(30)|
Unique keys: persona_id_key
```

Al definir un tipo serial, hemos creado implícitamente una secuencia independiente de la tabla. Podemos consultar las secuencias de nuestra base de datos mediante el comando '\ds':

```
demo=# \ds
                List of relations
 Schema |Name                 |Type      |Owner
--------+---------------------+----------+----------
 public | productos_clave_seq | sequence | postgres
(1 row)
```

Creamos nuevamente la tabla *estudiante* heredando de *persona*:

```
create table estudiante (
demo(# carrera varchar(50),
demo(# grupo char,
demo(# grado int
demo(# ) inherits ( persona );
CREATE
```

El estudiante heredará la columna *id* y se incrementará utilizando la misma secuencia:

```
demo=# \d persona
Table "persona"
Column   |Type                 |Modifiers
---------+---------------------+-------------------------------------------
       id|             integer |not null default next-val('"persona_id_seq"'::text)
   nombre|character varying(30)|
direccion|character varying(30)|
 carrera |character varying(50)|
   grupo |         character(1)|
   grado |              integer|
```

Insertaremos en la tabla algunos registros de ejemplo, omitiendo el valor para la columna *id*:

```
demo=# insert into persona(nombre,direccion)
values ( 'Federico Garca Lorca' , 'Granada 65' );
demo=# insert into persona(nombre,direccion)
values ( 'Alejandro Magno' , 'Babilonia' );
demo=# insert into estudiante(nombre,direccion,carrera,grupo,grado)
values ( 'Elizabeth' , 'Pino 35', 'Psicologia' , 'B' , 5 );
```

La tabla *estudiante* contendrá un solo registro, pero su identificador es el número 3.

```
demo=# select * from estudiante;
id |nombre     |direccion  |carrera     |grupo  |grado
---+-----------+-----------+------------+-------+-------
3  | Elizabeth | Pino 35   | Psicologia |     B |     5
(1 row)
```

Todos los registros de *persona* siguen una misma secuencia sin importar si son padres o hijos:

```
demo=# select * from persona;
 id |nombre                |direccion
----+----------------------+------------
  1 | Federico Garca Lorca | Granada 65
  2 |      Alejandro Magno | Babilonia
  3 |            Elizabeth | Pino 35
(3 rows)
```

La herencia es útil para definir tablas que conceptualmente mantienen elementos en común, pero también requieren datos que los hacen diferentes. Uno de los elementos que conviene definir como comunes son los identificadores de registro.

4.4. Restricciones

Como ya sabemos, las restricciones permiten especificar condiciones que deberán cumplir tablas o columnas para mantener la integridad de sus datos. Algunas de las restricciones vendrán impuestas por el modelo concreto que se esté implementando, mientras que otras tendrán su origen en las reglas de negocio del cliente, los valores que pueden tomar algunos campos, etc.

Los valores que puede contener una columna están restringidos en primer lugar por el tipo de datos. Ésta no es la única restricción que se puede definir para los valores en una columna, PostgreSQL ofrece las restricciones siguientes:

Ejemplo

Una columna definida como **integer** no puede contener cadenas de caracteres.

- **null** y **not null**. En múltiples ocasiones el valor de una columna es desconocido, no es aplicable o no existe. En estos casos, los valores cero, cadena vacía o falso son inadecuados, por lo que utilizamos **null** para especificar

la ausencia de valor. Al definir una tabla podemos indicar qué columnas podrán contener valores nulos y cuáles no.

```
create table Persona (
nombre varchar(40) not null,
trabajo varchar(40) null,
correo varchar(20),
);
```

El *nombre* de una persona no puede ser nulo, y es posible que la persona no tenga *trabajo*. También es posible que no tenga *correo*, al no especificar una restricción **not null**, se asume que la columna puede contener valores nulos.

- **unique**. Esta restricción se utiliza cuando no queremos que los valores contenidos en una columna puedan duplicarse.

```
create table Persona (
nombre varchar(40) not null,
conyuge varchar(40) unique,
);
```

cónyuge no puede contener valores duplicados, no permitiremos que dos personas tengan simultáneamente el mismo cónyuge.

- **primary key**. Esta restricción especifica la columna o columnas que elegimos como clave primaria. Puede haber múltiples columnas *unique*, pero sólo debe haber una clave primaria. Los valores que son únicos pueden servir para identificar una fila de la tabla de forma unívoca, por lo que se les denomina *claves candidatas*.

```
create table Persona (
nss varchar(10) primary key,
conyuge varchar(40) unique,
);
```

Al definir una columna como *primary key*, se define implícitamente con *unique*. El nss (número de la seguridad social) no sólo es único, sino que lo utilizamos para identificar a las personas.

- **references y foreign key**. En el modelo relacional, establecemos las relaciones entre entidades mediante la inclusión de claves foráneas en otras relaciones. PostgreSQL y SQL ofrecen mecanismos para expresar y mantener esta integridad referencial. En el siguiente ejemplo, las *Mascotas* tienen como dueño a una *Persona*:

```
create table Mascota(
nombre varchar(20),
dueño varchar(10) references Persona,
);
```

Una referencia por defecto es a una clave primaria, por lo que *dueño* se refiere implícitamente al *nss* de *Persona*. Cuando se capturen los datos de una nueva *mascota*, PostgreSQL verificará que el valor de *dueño* haga referencia a un nss que exista en *Persona*, en caso contrario emitirá un mensaje de error. En otras palabras, no se permite asignar a una mascota un dueño que no exista.

También es posible especificar a qué columna de la tabla hace referencia:

```
create table Mascota(
nombre varchar(20),
dueño varchar(10) references Persona(nss),
);
```

o su equivalente:

```
create table Mascota(
nombre varchar(20),
dueño varchar(10),
FOREIGN KEY dueño references Persona(nss),
);
```

Podría darse el caso de que la clave primaria de la tabla referenciada tuviera más de una columna, en ese caso, la clave foránea también tendría que estar formada por el mismo número de columnas:

```
create table t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

Si no se especifica otra acción, por omisión la persona que tenga una mascota no puede ser eliminada, porque la mascota se quedaría sin dueño. Para poder eliminar una persona, antes se deben eliminar las mascotas que tenga. Este comportamiento no parece ser el más adecuado para el caso.

Para modificar este comportamiento disponemos de las reglas de integridad referencial del lenguaje SQL, que PostgreSQL también soporta. En el siguiente ejemplo se permite que al eliminar una persona, las mascotas simplemente se queden sin dueño.

```
create table Mascota (
dueño varchar(10) references Persona on delete set null,
);
```

En cláusula **on delete** se pueden especificar las siguientes acciones:

– **set null**. La referencia toma el valor NULL: si se elimina *Persona* su *Mascota* se quedará sin dueño.

– **set default**. La referencia toma el valor por omisión.

– **cascade**. La acción se efectúa en cascada: si se elimina *Persona* automáticamente se elimina su *Mascota*.

– **restrict**. No permite el borrado del registro: no se puede eliminar una *Persona* que tenga *Mascota*. Ésta es la acción que se toma por omisión.

Si se modifica la clave primaria de la tabla referenciada, se dispone de las mismas acciones que en el caso anterior, que especificaremos con la cláusula ON UPDATE.

- **check**. Esta restricción realiza la evaluación previa de una expresión lógica cuando se intenta realizar una asignación. Si el resultado es verdadero, acepta el valor para la columna, en caso contrario, emitirá un mensaje de error y rechazará el valor.

```
create table Persona (
edad int check( edad > 10 and edad < 80 ),
correo varchar(20) check( correo ~ '.+@.+\..+' ),
ciudad varchar(30) check( ciudad <> '' )
);
```

Se han restringido los valores que se aceptarán en la columna de la manera siguiente.

- Edad debe estar entre 11 y 79 años.
- Ciudad no debe una cadena vacía.
- Correo debe tener una arroba.

Comparación de expresiones regulares

El operador ~ realiza comparaciones de cadenas con expresiones regulares.
Las expresiones regulares son patrones de búsqueda muy flexibles desarrollados en el mundo Unix.

Cualquiera de esas restricciones puede tener nombre, de manera que se facilita la referencia a las restricciones específicas para borrarlas, modificarlas, etc. pues puede hacerse por nombres. Para dar nombre a una restricción, utilizamos la sintaxis siguiente:

```
constraint nombre_de_restricciÛn <restricción>
```

4.4.1. Restricciones de tabla

Cuando las restricciones se indican después de las definiciones de las columnas, y pueden afectar a varias de ellas simultáneamente, se dice que son restricciones de tabla:

```
create table Persona (
nss int,
nombre varchar(30),
pareja varchar(30),
jefe int,
correo varchar(20),
primary key (nss),
unique (pareja),
foreign key (jefe) references Persona,
check (correo ~ '@' )
);
```

Esta notación permite que la restricción pueda abarcar varias columnas.

```
create table Curso (
materia varchar(30),
grupo char(4),
dia int,
hora time,
aula int,
primary key (materia, grupo),
unique (dia, hora, aula)
);
```

Un curso se identifica por el grupo y la materia, y dos cursos no pueden estar en la misma aula el mismo día y a la misma hora.

Al igual que la restricción de columna, a las restricciones de tabla puede asignárseles un nombre:

```
create table Persona (
nss int,
nombre varchar(30),
pareja varchar(30),
jefe int,
correo varchar(20),
constraint identificador primary key (nss),
constraint monogamia unique (pareja),
constraint un_jefe foreign key (jefe) references Persona,
check (correo ~ '@' )
);
```

La sentencia `alter table` permite añadir (`add`) o quitar (`drop`) restricciones ya definidas:

```
alter table Persona drop constraint monogamia
alter table add constraint monogamia unique (pareja);
```

4.5. Indexación

PostgreSQL crea índices para las llaves primarias de todas las tablas. Cuando se necesite crear índices adicionales, utilizaremos la expresión del ejemplo siguiente:

```
create index persona_nombre_indice on Persona ( nombre );
```

4.6. Consulta de información de bases de datos y tablas

Como ya sabemos, el cliente **psql** ofrece varias alternativas para obtener información sobre la estructura de nuestra base de datos. En la siguiente tabla se muestran algunos comandos de mucha utilidad.

Comando	Descripción
\l	Lista las bases de datos
\d	Describe las tablas de la base de datos en uso
\ds	Lista las secuencias
\di	Lista los índices
\dv	Lista las vistas
\dp \z	Lista los privilegios sobre las tablas
\da	Lista las funciones de agregados
\df	Lista las funciones
\g archivo	Ejecuta los comandos de archivo
\H	Cambia el modo de salida HTML
\! comando	Ejecuta un comando del sistema operativo

Para obtener la lista de tablas de la base de datos *demo* hacemos lo siguiente:

```
demo=# \d
List of relations
Name                 |Type      | Owner
---------------------+----------+--------
            ganancia |    table | postgres
             precios |    table | postgres
           productos |    table | postgres
 productos_clave_seq | sequence | postgres
         proveedores |    table | postgres
(5 rows)
```

La estructura de la tabla *productos* se solicita de la siguiente manera.

```
demo=# \d productos
Table "productos"
Column        |Type                 | Modifiers
--------------+---------------------+-------------------------------------------
        parte |character varying(20)|
         tipo |character varying(20)|
especificación|character varying(20)|
    psugerido |               real |
       clave |integer            |not null default nextval('"productos_clave_seq"'::text)
Primary key: productos_pkey
Triggers: RI_ConstraintTrigger_17342,
RI_ConstraintTrigger_17344
```

En el ejemplo anterior podemos observar que la columna *clave* contiene dos modificadores:

- El primero especifica que no pueden asignarse valores nulos.
- El segundo especifica el valor por omisión que deberá asignarse a la columna.

En este caso, el valor será automáticamente calculado por la función *nextval()*, que toma como argumento la **secuencia*** *productos_clave_seq*.

*** Una secuencia es un nombre especial que permite la producción de series numéricas.**

El siguiente comando muestra las secuencias creadas en una base de datos:

```
demo=# \ds
List of relations
Name                 |Type      | Owner
---------------------+----------+--------
productos_clave_seq | sequence | quiron
(1 row)
```

Las secuencias se crean automáticamente cuando se declaran columnas de tipo serial.

En la estructura de la tabla productos encontramos también una clave primaria. PostgreSQL generará siempre un índice para cada tabla utilizando la clave primaria. La lista de los índices de la base de datos se obtiene de la siguiente forma:

```
demo=# \di
List of relations
Name              | Type  | Owner
------------------+-------+--------
productos_pkey    | index | quiron
proveedores_pkey  | index | quiron
(2 rows)
```

El conjunto de comandos proporcionados por **psql** que hemos presentado permite obtener información sobre la estructura de nuestra base de datos de una manera directa y sencilla y, también, es útil para explorar bases de datos que no conozcamos.

4.7. Tipos de datos

4.7.1. Tipos lógicos

PostgreSQL incorpora el tipo lógico **boolean**, también llamado **bool**. Ocupa un byte de espacio de almacenamiento y puede almacenar los valores falso y verdadero.

Valor	Nombre
Falso	false, 'f', 'n', 'no', 0
Verdadero	true, 't', 'y', 'yes', 1

PostgreSQL soporta los operadores lógicos siguientes: **and**, **or** y **not**.

Aunque los operadores de comparación se aplican sobre prácticamente todos los tipos de datos proporcionados por PostgreSQL, dado que su resultado es un valor lógico, describiremos su comportamiento en la siguiente tabla:

Operador	Descripción
>	Mayor que
<	Menor que
<=	Menor o igual que
>=	Mayor o igual que
<> !=	Distinto de

4.7.2. Tipos numéricos

PostgreSQL dispone de los tipos enteros **smallint**, **int** y **bigint** que se comportan como lo hacen los enteros en muchos lenguajes de programación.

Los números con punto flotante **real** y **double precisión** almacenan cantidades con decimales. Una característica de los números de punto flotante es que pierden exactitud conforme crecen o decrecen los valores.

Aunque esta pérdida de exactitud no suele tener importancia en la mayoría de las ocasiones, PostgreSQL incluye el tipo **numeric**, que permite almacenar cantidades muy grandes o muy pequeñas sin pérdida de información. Por supuesto, esta ventaja tiene un coste, los valores de tipo **numeric** ocupan un espacio de almacenamiento considerablemente grande y las operaciones se ejecutan sobre ellos muy lentamente. Por lo tanto, no es aconsejable utilizar el tipo **numeric** si no se necesita una alta precisión o se prima la velocidad de procesamiento.

Nombre	Tamaño	Otros nombres	Comentario
smallint	2 bytes	int2	
int	4 bytes	int4, integer	

Nombre	Tamaño	Otros nombres	Comentario
bigint	8 bytes	int8	
numeric(p,e)	11 + (p/2)		'p' es la precisión, 'e' es la escala
real	4 bytes	float, float4	
double precision	8 bytes	float8	
serial			No es un tipo, es un entero auto-incrementable

Serial

La declaración **serial** es un caso especial, ya que no se trata de un nuevo tipo. Cuando se utiliza como nombre de tipo de una columna, ésta tomará automáticamente valores consecutivos en cada nuevo registro.

Ejemplo de una tabla que define la columna *folio* como tipo **serial**.

```
create table Factura(
folio serial,
cliente varchar(30),
monto real
);
```

PostgreSQL respondería esta instrucción con dos mensajes:

- En el primero avisa que se ha creado una secuencia de nombre *factura_folio_seq*:

```
NOTICE: CREATE TABLE will create implicit sequence 'factura_folio_seq' for SERIAL column '
```

- En el segundo avisa de la creación de un índice único en la tabla utilizando la columna *folio*:

```
NOTICE: CREATE TABLE / UNIQUE will create implicit index 'factura_folio_key' for table 'factura'
CREATE
```

Si se declaran varias columnas con **serial** en una tabla, se creará una secuencia y un índice para cada una de ellas.

4.7.3. Operadores numéricos

PostgreSQL ofrece un conjunto predefinido de operadores numéricos, que presentamos en la siguiente tabla:

Símbolo	Operador
+	Adición
-	Substracción
*	Multiplicación
/	División
%	Módulo
^	Exponenciación
|/	Raíz cuadrada

Ejemplo

```
select |/ 9;
select 43 % 5;
select !! 7;
select 7!;
```

Símbolo	Operador
||/	Raíz cúbica
!	Factorial
!!	Factorial como operador fijo
@	Valor absoluto
&	AND binario
		OR binario
#	XOR binario
~	Negación binaria
<<	Corrimiento binario a la izquierda
>>	Corrimiento binario a la derecha

4.7.4. Tipos de caracteres

Los valores de cadena en PostgreSQL se delimitan por comillas simples.

Recordad

Las comillas dobles delimitan identificadores que contienen caracteres especiales.

```
demo=# select 'Hola mundo';
?column?
------------
Hola mundo
(1 row)
```

Se puede incluir una comilla simple dentro de una cadena con \' o '':

```
demo=# select 'Él dijo: ''Hola'' ';
?column?
------------------
Él dijo: 'Hola'
(1 row)
```

Las cadenas pueden contener caracteres especiales utilizando las llamadas secuencias de escape que inician con el caracter '\':

```
\n nueva línea
\r retorno de carro
\t tabulador
\b retroceso
\f cambio de página
\r retorno de carro
\\ el caracter \
```

Las secuencias de escape se sustituyen por el carácter correspondiente:

```
demo=# select 'Esto está en \n dos renglones';
?column?
------------------------------
Esto está en
dos renglones
(1 row)
```

PostgreSQL ofrece los tipos siguientes para cadenas de caracteres:

Tipo	Otros nombres	Descripción
char(n)	character(n)	Reserva *n* espacios para almacenar la cadena
varchar(n)	character var-ying(n)	Utiliza los espacios necesarios para almacenar una cadena menor o igual que *n*
text		Almacena cadenas de cualquier magnitud

4.7.5. Operadores

En la siguiente tabla se describen los operadores para cadenas de caracteres:

Operador	Descripción	¿Distingue mayúsculas y minúsculas?
||	Concatenación	-
~	Correspondencia a expresión regular	Sí
~*	Correspondencia a expresión regular	No
!~	No correspondencia a expresión regular	Sí
!~*	No correspondencia a expresión regular	-

En la siguiente tabla se muestran algunas funciones de uso común sobre cadenas de caracteres:

Bibliografía

Es recomendable consultar el manual para obtener la referencia completa de funciones.

Función	Descripción
Length(cadena)	Devuelve la longitud de la cadena
lower(cadena)	Convierte la cadena a minúsculas
ltrim(cadena,caracteres)	Elimina de la izquierda los caracteres especificados
substring(cadena from patrón)	Extrae la subcadena que cumple el patrón especificado

Sobre las cadenas también podemos utilizar los operadores de comparación que ya conocemos.

Ejemplo

En este caso, el resultado de la comparación *menor que* es VERDADERO:

```
demo=# select 'HOLA' < 'hola';
?column?
----------
t
(1 row)
```

4.7.6. Fechas y horas

En la siguiente tabla se muestran los tipos de datos referentes al tiempo que ofrece PostgreSQL:

Tipo de dato	Unidades	Tamaño	Descripción	Precisión
date	día-mes-año	4 bytes	Fecha	Día
time	hrs:min:seg:micro	4 bytes	Hora	Microsegundo
timestamp	día-mes-año hrs:min:seg:micro	8 bytes	Fecha más hora	Microsegundo
interval	second, minute, hour, day, week, month, year, decade, century, millennium*	12 bytes	Intervalo de tiempo	Microsegundo

*** También admite abreviaturas.**

Existe un tipo de dato **timez** que incluye los datos del tipo **time** y, además, la zona horaria. Su sintaxis es la siguiente:

```
hh:mm[:ss[.mmm]][am|pm][zzz]
```

El tipo de datos **date** almacena el día, mes y año de una fecha dada y se muestra por omisión con el formato siguiente: YYYY-MM-DD:

```
demo=# create table Persona ( nacimiento date );
CREATE
demo=# insert into persona values ( '2004-05-22' );
INSERT 17397 1
demo=# select * from persona;
nacimiento
------------
2004-05-22
(1 row)
```

Para cambiar el formato de presentación, debemos modificar la variable de entorno *datestyle*:

```
demo=# set datestyle = 'german';
SET VARIABLE
demo=# select * from persona;
nacimiento
------------
22.05.2004
(1 row)
```

Nombre del formato	Formato	Ejemplo
ISO	Año-mes-día	2004-05-22
GERMAN	Día.mes.año	22.05.2004
POSTGRES	día-mes-año	22-05-2004
SQL	mes/día/año	05/22/2004

4.7.7. *Arrays*

Definición

Los *arrays* no cumplen la primera forma normal de Cood, por lo que muchos los consideran inaceptables en el modelo relacional.

El tipo de datos ***array*** es una de las características especiales de PostgreSQL, permite el almacenamiento de más de un valor del mismo tipo en la misma columna.

```
demo=# create table Estudiante (
demo(# nombre varchar(30),
demo(# parciales int [3]
demo(# );
CREATE
```

La columna *parciales* acepta tres calificaciones de los estudiantes.

También es posible asignar un solo valor del *array*:

```
demo=# insert into Estudiante( nombre, parciales[2]) values ( 'Pedro' , '{90}');
INSERT 17418 1
demo=# select * from Estudiante ;
nombre      | parciales
------------+------------
John Lennon |
Juan        | {90,95,97}
Pedro       | {90}
(3 rows)
```

Los *arrays*, al igual que cualquier columna cuando no se especifica lo contrario, aceptan valores nulos:

```
demo=# insert into Estudiante values ( 'John Lennon ' );
INSERT 17416 1
demo=# insert into Estudiante values ( 'Juan' , '{90,95,97}' );
INSERT 17417 1
```

Los valores del array se escriben siempre entre llaves.

```
demo=# select * from Estudiante;
nombre        | parciales
--------------+------------
John Lennon   |
Juan          | {90,95,97}
(2 rows)
```

Para seleccionar un valor de un *array* en una consulta se especifica entre corchetes la celda que se va a desplegar:

```
demo=# select nombre, parciales[3] from Estudiante;
nombre        | parciales
--------------+-----------
John Lennon   |
Juan          | 97
Pedro         |
(3 rows)
```

Sólo Juan tiene calificación en el tercer parcial.

En muchos lenguajes de programación, los *array* se implementan con longitud fija, PostgreSQL permite aumentar su tamaño dinámicamente:

La columna *parciales* del registro Pablo incluye cuatro celdas y sólo la última tiene valor.

```
demo=# insert into Estudiante( nombre, parciales[4]) values ( 'Pablo' , '{70}');
INSERT 17419 1
demo=# select * from Estudiante;
nombre        | parciales
--------------+------------
John Lennon   |
Juan          | {90,95,97}
Pedro         | {90}
Pablo         | {70}
(4 rows)
```

Mediante la función array_dims() podemos conocer las dimensiones de un *array*:

```
demo=# select nombre, array_dims(parciales) from Estudiante;
nombre        | array_dims
--------------+------------
John Lennon   |
Juan          | [1:3]
Pedro         | [1:1]
Pablo         | [1:1]
(4 rows)
```

4.7.8. BLOB

El tipo de datos BLOB (Binary Large OBject) permite almacenar en una columna un objeto de gran tamaño. PostgreSQL no conoce nada sobre el tipo de información que se almacena en una columna BLOB, simplemente lo considera como una secuencia de bytes. Por este motivo, no se tienen operaciones sobre los tipos BLOB, con la excepción del operador de concatenación, que simplemente une el contenido de dos BLOB en uno.

Veamos cómo almacenar una fotografía en una tabla de personas mediante tipos BLOB.

Una primera manera de hacerlo es importando el contenido del archivo que contiene la imagen mediante la función **lo_import()**:

```
demo=# select lo_import('/home/quiron/mi-foto.jpg');
lo_import
-----------
17425
(1 row)
```

Esta función devuelve como resultado el OID del objeto insertado.¿Dónde se ha almacenado la fotografía si no hemos utilizado el comando *insert*? PostgreSQL mantiene una tabla de nombre **pg_largeobject** con el objetivo de almacenar BLOB. Podemos utilizar el OID para hacer referenciar al objeto en una tabla:

```
demo=# create table persona (
demo(# nombre varchar(30),
demo(# direccion varchar(30),
demo(# fotografia oid
demo(# );
CREATE
```

Los registros insertados en esta tabla llevan un número entero como OID que, en el siguiente caso, se ha obtenido solicitándolo a la función **lo_import()** anterior.

```
demo=# insert into persona values ( 'Julio' , 'Cedro 54', 17425);
```

La inserción anterior pudo haberse realizado en un solo paso:

```
demo=# insert into persona
values ( 'Julio' , 'Cedro 54', lo_import('/home/quiron/mi-foto.jpg'));
```

Para extraer el contenido de la columna, se utiliza la función **lo_export()**

```
demo=# select lo_export(17425,'/tmp/mi-foto.jpg');
lo_export
-----------
1
(1 row)
```

La función **lo_unlink()** permite borrar un BLOB almacenado en *pg_largeobject:*

```
select lo_unlink(17425);
```

Veamos el formato que utiliza PostgreSQL para visualizar BLOB.

Se ha recortado la salida para hacer más comprensible la tabla.

```
loid  | pageno | data
------+--------+------------------------------------------------------------------
17425 | 0      | \377\330\377\340\000\020JFIF\000\001\001\001\000H\000H\000\000\37
17425 | 1      | \256-}\306\267\032s[\336)\245\231\370|L\206\275\364\224\321\237\2
17425 | 2      | \341\226;\015l\232\033f\\\371\251\0323\003t\307\207~\035GB\271\17
(3 rows)
```

La fotografía se ha dividido en tres registros, que son las páginas 0, 1 y 2 del BLOB. Los tres registros tienen el mismo *loid*, lo que significa que son el mismo objeto. Obsérvese también, que los bytes son desplegados como caracteres de la forma '*ddd*', donde ddd son tres dígitos octales.

Para almacenar un BLOB en una tabla de forma directa; es decir, sin utilizar la tabla del sistema *pg_largeobject* utilizamos el tipo de dato **bytea**:

En el primer caso se está insertando un BLOB de un solo byte, el carácter ASCII cero.

En el segundo caso se están insertando 4 bytes, los tres primeros están representados directamente por los caracteres imprimibles ASCII y el tercero, el caracter ASCII 30 octal, como no es imprimible, se escribe en notación '*ddd*'.

```
demo=# create table persona (
demo(# nombre varchar(30),
demo(# direccion varchar(30),
demo(# fotografia bytea
demo(# );
CREATE
```

La columna fotografía es de tipo **bytea**, lo que permite almacenar objetos de gran tamaño, cerca de 1GB. Para insertar valores en estas columnas, hacemos lo siguiente:

```
demo=# insert into persona values ( 'Jorge' , 'Cerezo 55', '\\000');
INSERT 17436 1
demo=# insert into persona values ( 'Luis' , 'Encino 67', 'abc\\030');
INSERT 17437 1
```

La consulta se visualiza como sigue:

```
demo=# select * from persona;
nombre  | direccion | fotografia
--------+-----------+------------
Jorge   | Cerezo 55 | \000
Luis    | Encino 67 | abc\030
(2 rows)
```

Los caracteres en notación octal se muestran con una barra invertida y con dos tal como se escribieron. Esto es debido a que, en realidad, sólo llevan una barra invertida, pero por cuestiones de diseño PostgreSQL, las literales BLOB deben escribirse con doble barra invertida.

4.8. Modificación de la estructura de una tabla

Para modificar la estructura de una tabla una vez construida, disponemos de la sentencia SQL **alter table**.

Mediante esta sentencia, podemos llevar a cabo las operaciones siguientes:

- Agregar una columna.

```
demo=# alter table persona add edad int ;
ALTER
```

- Eliminar una columna.

```
demo=# ALTER TABLE products DROP COLUMN description;
```

- Fijar el valor por omisión de una columna.

```
demo=# alter table persona alter edad set default 15;
ALTER
```

- Eliminar el valor por omisión de una columna.

```
demo=# alter table persona alter edad drop default;
ALTER
```

- Renombrar una columna.

```
demo=# alter table persona rename direccion to dir;
ALTER
```

- Renombrar una tabla.

```
demo=# alter table persona rename to personal;
ALTER
```

5. Manipulación de datos

5.1. Consultas

Las consultas a la base de datos se realizan con el comando **select**, que se implementa en PostgreSQL cumpliendo en gran parte con el estándar SQL:

Notación

Omitiremos las referencias comunes a SQL y sólo se mostrarán algunas de las posibilidades de consulta con PostgreSQL. Por lo que respecta a las funciones auxiliares, se han visto algunas en el apartado de tipos de datos y, en todo caso, se recomienda la consulta de la documentación del producto para las operaciones más avanzadas.

```
demo=# select parte, tipo
demo-# from productos
demo-# where psugerido > 30
demo-# order by parte
demo-# limit 5
demo-# offset 3;
parte       | tipo
------------+----------
Monitor     | 1024x876
Monitor     | 1024x876
Procesador  | 2.4 GHz
Procesador  | 1.7 GHz
Procesador  | 3 GHz
(5 rows)
```

Al igual que MySQL, PostgreSQL admite la sentencia **explain** delante de **select** para examinar qué está ocurriendo durante una consulta:

Al igual que en el módulo de MySQL, vemos que no aprovecha los índices (básicamente porque no tenemos ninguno definido).

```
demo=# explain select productos.clave, parte||' '||tipo||' '||especificación as producto,
proveedores.empresa , precio  from productos natural join precios natural join proveedores;
                                    QUERY PLAN
------------------------------------------------------------------------------
Hash Join  (cost=45.00..120.11 rows=1000 width=104)
   Hash Cond: (("outer".empresa)::text = ("inner".empresa)::text)
   -> Hash Join  (cost=22.50..72.61 rows=1000 width=104)
      Hash Cond: ("outer".clave = "inner".clave)
      -> Seq Scan on precios  (cost=0.00..20.00 rows=1000 width=32)
      -> Hash (cost=20.00..20.00 rows=1000 width=76)
         -> Seq Scan on productos  (cost=0.00..20.00 rows=1000 width=76)
   -> Hash (cost=20.00..20.00 rows=1000 width=24)
      -> Seq Scan on proveedores  (cost=0.00..20.00 rows=1000 width=24)
(9 rows)
demo=#
```

Veamos como mejorar el rendimiento de este **select**:

```
demo=# create index empresa_idx on precios (empresa);
CREATE INDEX
demo=# create index clave_idx on precios (clave);
CREATE INDEX
demo=# explain select productos.clave, parte||' '||tipo||' '||especificación as producto,
proveedores.empresa , precio  from productos natural join precios natural join proveedores;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=29.00..56.90 rows=20 width=104)
   Hash Cond: ("outer".clave = "inner".clave)
   ->  Seq Scan on productos  (cost=0.00..20.00 rows=1000 width=76)
   ->  Hash  (cost=28.95..28.95 rows=20 width=32)
      ->  Hash Join  (cost=1.25..28.95 rows=20 width=32)
         Hash Cond: (("outer".empresa)::text = ("inner".empresa)::text)
         ->  Seq Scan on proveedores  (cost=0.00..20.00 rows=1000 width=24)
         ->  Hash  (cost=1.20..1.20 rows=20 width=32)
            ->  Seq Scan on precios  (cost=0.00..1.20 rows=20 width=32)
(9 rows)
demo=#
```

5.2. Actualizaciones e inserciones

PostgreSQL cumple con el estándar SQL en todos los sentidos para las sentencias de actualización, inserción y borrado. No define modificadores ni otros mecanismos para estas sentencias.

En determinadas cláusulas, y para tablas heredadas, es posible limitar el borrado o la actualización a la tabla *padre* (sin que se propague a los registros de las tablas hijas) con la cláusula **only**:

```
demo=# update only persona set nombre='Sr. '||nombre;
UPDATE 2
demo=# select * from persona;
nombre                    | direccion
--------------------------+-------------
 Sr. Alejandro Magno      | Babilonia
 Sr. Federico García Lorca | Granada 65
 Juan                     | Treboles 21
(3 rows)
demo=#
```

5.3. Transacciones

Definimos *transacción* como un conjunto de operaciones que tienen significado solamente al actuar juntas.

PostgreSQL ofrece soporte a transacciones, garantizando que ambas operaciones se realicen o que no se realice ninguna. Para iniciar una transacción, se utiliza el comando **begin** y para finalizarla, **commit**.

Ejemplo

Una compra puede ser una transacción que conste de dos operaciones:

- Insertar un registro del pago del producto
- Insertar el producto en el inventario.

No se debe insertar un producto que no se haya pagado, ni pagar un producto que no esté en el inventario, por lo tanto, las dos operaciones forman una transacción.

```
demo=# begin;
BEGIN
demo=# insert into productos values ('RAM','512MB','333 MHz',60);
INSERT 17459 1
demo=# select * from productos;
parte      | tipo     | especificación | psugerido | clave
-----------+----------+----------------+-----------+-------
Procesador | 2 Ghz    | 32 bits        |           | 1
Procesador | 2.4 GHz  | 32 bits        | 35        | 2
Procesador | 1.7 GHz  | 64 bits        | 205       | 3
Procesador | 3 GHz    | 64 bits        | 560       | 4
RAM        | 128MB    | 333 MHz        | 10        | 5
RAM        | 256MB    | 400 Mhz        | 35        | 6
Disco Duro | 80 GB    | 7200 rpm       | 60        | 7
Disco Duro | 120 GB   | 7200 rpm       | 78        | 8
Disco Duro | 200 GB   | 7200 rpm       | 110       | 9
Disco Duro | 40 GB    | 4200 rpm       |           | 10
Monitor    | 1024x876 | 75 Hz          | 80        | 11
Monitor    | 1024x876 | 60 Hz          | 67        | 12
RAM        | 512MB    | 333 MHz        | 60        | 13
(13 rows)
demo=# insert into precios values ('Patito',13,67);
INSERT 17460 1
```

El nuevo registro tiene como clave el 13 y, de momento, hasta que finalice la transacción, sólo puede verlo el usuario que lo ha insertado.

Insertamos un registro con el precio del proveedor Patito para el producto con clave 13.

```
demo=# commit;
```

Al cerrar la transacción, los registros insertados ya son visibles para todos los usuarios. Si por alguna razón, por ejemplo una caída del sistema, no se ejecuta el **commit**, la transacción se cancela. La forma explícita de cancelar una transacción es con el comando **rollback**.

Las transacciones de PostgreSQL

No hay ninguna característica en las transacciones de PostgreSQL que lo diferencien de lo que especifica el estándar. Ya hemos visto en apartados anteriores que las filas implicadas en una transacción mantienen unas columnas ocultas con información acerca de la propia transacción.

6. Funciones y disparadores

Como algunos de los gestores de bases de datos relacionales, comerciales líderes en el mercado, PostgreSQL puede incorporar múltiples lenguajes de programación a una base de datos en particular. Este hecho permite, por ejemplo:

- Almacenar procedimientos en la base de datos (***stored procedure***), que podrán lanzarse cuando convenga.
- Definir operadores propios.

PostgreSQL ofrece por defecto soporte para su propio lenguaje procedural, el PL/pgSQL. Para instalarlo, basta con invocar el comando **createlang** desde el sistema operativo, no desde la línea de psql.

PL/pgSQL

PL/pgSQL (*procedual language/postgreSQL*) es una extensión del SQL que permite la creación de procedimientos y funciones al estilo de los lenguajes tradicionales de programación.

```
$ createlang plpgsql demo
```

Mediante este comando, se ha instalado el lenguaje PL/pgSQL en la base de datos *demo*.

PostgreSQL también soporta otros lenguajes directamente, como PL/Tcl, PL/Perl y PL/Python.

6.1. Primer programa

Veamos el programa *HolaMundo* en PL/pgSQL:

```
demo=# create function HolaMundo() returns char
demo-# as ' begin return "Hola Mundo PostgreSQL" ; end; '
demo-# language 'plpgsql';
CREATE
```

La función tiene tres partes:

- El encabezado que define el nombre de la función y el tipo de retorno.
- El cuerpo de la función, que es una cadena de texto (por lo tanto, siempre va entre comillas dobles).
- La especificación del lenguaje utilizado.

La función recien creada tiene las mismas características que las integradas.

Puede solicitarse mediante el comando **select**:

```
demo=# select HolaMundo();
holamundo
-----------------------
Hola Mundo PostgreSQL
(1 row)
```

Puede eliminarse mediante el comando **drop function**.

```
demo=# drop function HolaMundo();
DROP
```

6.2. Variables

Las funciones pueden recibir parámetros, sólo es necesario especificar los tipos de datos. PostgreSQL asigna los nombres a los parámetros utilizando la secuencia $1, $2, $3...

En este ejemplo veremos todas las posibles maneras de declarar variables en una función.

La sentencia alias crea un nuevo nombre para una variable. La sentencia rename cambia el nombre de una variable.

```
create function mi_funcion(int,char) returns int
as '
declare -- declaración de variables locales
x int; -- x es de tipo entero
y int := 10; -- y tiene valor inicial de 10
z int not null; -- z no puede tomar valores nulos
a constant int := 20; -- a es constante
b alias for $1; -- El primer parámetro tiene dos nombres.
rename $1 to c; -- Cambia de nombre el segundo parámetro
begin
x := y + 30;
end;
' language 'plpgsql';
```

6.3. Sentencias

La estructura básica de una función es el **bloque**, que incluye dos partes, la declaración de variables y la sección de sentencias:

declare
secciÛn de variables
begin
secciÛn de sentencias
end;

Sentencia	Descripción
declare begin end	Bloque
:=	Asignación
select into	Asignación desde un **select**
Sentencias `sql`	Cualquier sentencia `sql`

Sentencia	Descripción
perform	Realiza una llamada a comando sql
execute	Interpreta una cadena como comando sql
exit	Termina la ejecución de un bloque
return	Termina la ejecución de una función
if	Ejecuta sentencias condicionalmente
loop	Repite la ejecución de un conjunto de sentencias
while	Repite un conjunto de sentencias mientras
for	Repite un conjunto de sentencias utilizando una variable de control
raise	Despliega un mensaje de error a advertencia

La sentencia de **asignación** utiliza el operador '**:=**' para almacenar los resultados de expresiones en variables. PostgreSQL proporciona otra sentencia para hacer asignaciones, select. Esta sentencia debe obtener como resultado un solo valor para que pueda ser almacenado en la variable:

```
select into x psugerido from productos where clave = 3;
```

La ejecución de comandos sql como **create**, **drop**, **insert** o **update** pueden hacerse sin ninguna sintaxis especial. La excepción es el comando **select**, que requiere ejecutarse con el comando **perform** a fin de que el resultado de la consulta sea descartado.

```
perform select psugerido from productos;
```

La sentencia **execute** también ejecuta un comando sql pero a partir de una cadena de texto. Esta sentencia comporta el problema de que su sintaxis no se verifica hasta la ejecución. Se puede utilizar, por ejemplo, para procesar parámetros como comandos sql:

```
execute $1
```

El comando **exit** termina la ejecución de un bloque. Se utiliza principalmente para romper ciclos.

La bifurcación, o ejecución condicional, se realiza mediante la sentencia **if**:

```
if ( $1 > 0 ) then
resultado := 'Positivo';
else
resultado := 'Negativo';
end if;
```

También puede utilizarse **if** con más de dos ramas:

```
if ( $1 > 0 ) then
resultado := 'Positivo';
elsif ( $1 < 0 ) then
resultado := 'Negativo';
else
resultado := 'Cero';
end if;
```

Con referencia a los bucles, PL/pgSQL ofrece tres opciones:

- El bucle **loop** es infinito, por lo que tiene una estructura muy simple. Por lo general se utiliza con alguna sentencia **if** para terminarlo:

```
cont := 0;
loop
if ( cont = 10 ) then
exit;
end if;
-- alguna acción
cont := cont + 1;
end loop;
```

- El bucle **while** incluye la condición al inicio del mismo, por lo que el control de su terminación es más claro:

```
cont := 0;
while cont != 10 loop
-- alguna acción
cont := cont + 1;
end loop;
```

- El bucle **for** permite realizar un número de iteraciones controladas por la variable del ciclo:

```
for cont in 1 .. 10 loop
-- alguna acción
end loop;
```

La sentencia **raise** permite enviar mensajes de tres niveles de severidad:

- **debug**. El mensaje se escribe en la bitácora del sistema (logs).
- **notice**. El mensaje se escribe en la bitácora y en el cliente psql.
- **exception** . El mensaje se escribe en la bitácora y aborta la transacción.

El mensaje puede incluir valores de variables mediante el carácter ' %':

- **raise debug** 'funcion(): ejecutada con éxito;
- **raise notice** 'El valor % se tomo por omisión', variable;
- **raise excepción** 'El valor % está fuera del rango permitido', variable;

6.4. Disparadores

Las funciones deben llamarse explícitamente para su ejecución o para incluirlas en consultas. Sin embargo, se puede definir que algunas funciones se ejecuten automáticamente cuando cierto evento tenga lugar en cierta tabla. Estas funciones se conocen como disparadores o *triggers* y se ejecutan mediante los comandos **insert**, **delete** y **uptade**.

Agregamos la tabla *historial* que almacena los productos descontinuados cuando se eliminan de la tabla *productos*.

```
create table historial(
fecha date,
parte varchar(20),
tipo varchar(20),
especificacion varchar(20),
precio float(6)
);
```

Para poder utilizar una función como disparador, no debe recibir argumentos y debe retornar el tipo especial **trigger**:

```
create function respaldar_borrados() returns trigger as '
begin
insert into historial values (
now(),
old.parte,
old.tipo,
old.especificacion,
old.psugerido
);
return null;
end;
```

La variable old está predefinida por PostgreSQL y se refiere al registro con sus antiguos valores. Para referirse a los nuevos valores, se dispone de la variable new.

La función está lista para ser utilizada como disparador, sólo es necesario definirlo y asociarlo a la tabla y al evento deseado:

```
create trigger archivar
beofre delete
on productos
for each row execute procedure respaldar_borrados();
```

Acabamos de crear un disparador de nombre *archivar* que se activará cuando se ejecute el comando *delete* en la tabla *productos*. El usuario no necesita saber que se debe hacer una copia de seguridad de los registros borrados, se hace automáticamente.

Al crear el disparador, hemos especificado "before delete" al indicar la operación. PostgreSQL nos permite lanzar el disparador antes o después (*before, after*) que se efectúen las operaciones. Este matiz es importante, ya que, si este mismo disparador lo ejecutamos después de la operación, no veremos ninguna fila en la tabla. Es posible definir el mismo disparador para varias operaciones:

```
create trigger archivar
beofre delete or update
on productos
for each row execute procedure respaldar_borrados();
```

7. Administración de PostgreSQL

En las tareas administrativas como la instalación, la gestión de usuarios, las copias de seguridad, restauraciones y el uso de prestaciones internas avanzadas, es donde realmente se aprecian las diferencias entre gestores de bases de datos. PostgreSQL tiene fama de ser más complejo de administrar que sus competidores de código abierto, lo que se debe, sobre todo, a que ofrece más prestaciones (o más complejas).

El contenido de los siguientes apartados contempla las opciones de uso común para la administración de un servidor PostgreSQL. Existen tantas alternativas que no es posible incluirlas todas en este módulo, por lo que sólo se presentarán algunos temas de importancia para el administrador, desde una perspectiva general, que permita obtener una visión global de las posibilidades prácticas de las herramientas administrativas.

Bibliografía

El manual de PostgreSQL es la referencia principal que se debe tener siempre a mano para encontrar posibilidades y resolver dudas. En especial se recomienda leer los siguientes capítulos:

Capítulo III. Server Administration.

Capítulo V. Server Programming.

Capítulo VII. Internals.

De la misma manera, también son muy útiles las listas de correo que se describen en el sitio oficial www.postgresql.org.

7.1. Instalación

PostgreSQL está disponible para la mayoría de distribuciones de GNU/Linux. Su instalación es tan sencilla como ejecutar el instalador de paquetes correspondiente.

En Debian, el siguiente procedimiento instala el servidor y el cliente respectivamente:

```
# apt-get install postgresql
# apt-get install postgresql-client
```

En distribuciones basadas en RPM, los nombres de los paquetes son un poco diferentes:

```
# rpm -Uvh postgresql-server
# rpm -Uvh postgresql
```

Una vez instalado, se escribirá un *script* de inicio que permite lanzar y apagar el servicio PostgreSQL; de este modo, para iniciar el servicio, deberemos ejecutar el siguiente comando:

```
# /etc/init.d/postgresql start
```

Notación

Además del **start** también podremos utilizar los parámetros **restart, stop, reload** que permiten reiniciar, detener y recargar el servidor (releyendo su configuración), respectivamente.

Si se desea realizar una instalación a partir del código fuente, puede obtenerse del sitio oficial www.postgresql.org. A continuación, se describe el proceso de instalación de forma muy simplificada. En la práctica podrán encontrarse algunas diferencias; lo más recomendable es leer cuidadosamente la documentación incluida en los archivos INSTALL y README. Cualquier duda no resuelta por la documentación, puede consultarse en la lista de distribución.

```
# tar xzvf postgresql-7.4.6.tar.gz
# cd postgresql-7.4.6
# ./configure
# make
# make install
```

Con este proceso se instala la versión 7.4.6. El archivo se descomprime utilizando *tar*. Dentro del directorio recién creado se ejecuta *configure*, que realiza una comprobación de las dependencias de la aplicación. Antes de ejecutar *configure,* debemos instalar todos los paquetes que vamos a necesitar.

La compilación se realiza con *make* y, finalmente, los binarios producidos se copian en el sistema en los lugares convenientes con *make install*.

Después de instalados los binarios, se debe crear el usuario *postgres* (responsable de ejecutar el proceso *postmaster*) y el directorio donde se almacenarán los archivos de las bases de datos.

```
# adduser postgres
# cd /usr/local/pgsql
# mkdir data
# chown postgres data
```

Una vez creado el usuario *postgres*, éste debe inicializar la base de datos:

```
# su - postgres
# /usr/local/pgsql/initbd -D /usr/local/pgsql/data
```

initdb

El ejecutable *initdb* realiza el procedimiento necesario para inicializar la base de datos de postgres, en este caso, en el directorio /usr/local/pgsql/data.

El *postmaster* ya está listo para ejecutarse manualmente:

```
# /usr/local/pgsql/postmaster -D /usr/local/pgsql/data
```

Bibliografía

El proceso de compilación tiene múltiples opciones que se explican en la documentación incluida con las fuentes.

7.1.1. Internacionalización

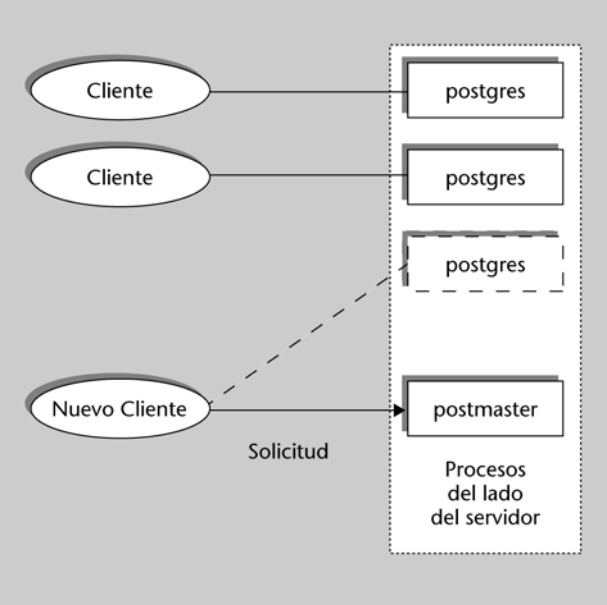
Por omisión, PostgreSQL no está compilado para soportar mensajes en español, por lo que es necesario compilarlo a partir de las fuentes incluyendo las

siguientes opciones de configuración, para que tanto el servidor como el cliente **psql** adopten la configuración establecida por el programa **setlocales** y las variables de entorno respectivas:

```
# configure --enable-nls -enable-locale
```

7.2. Arquitectura de PostgreSQL

El siguiente gráfico muestra de forma esquemática las entidades involucradas en el funcionamiento normal del gestor de bases de datos:



PostgreSQL está basado en una arquitectura cliente-servidor. El programa servidor se llama **postgres** y entre los muchos programas cliente tenemos, por ejemplo, **pgaccess** (un cliente gráfico) y **psql** (un cliente en modo texto).

Un proceso servidor *postgres* puede atender exclusivamente a un solo cliente; es decir, hacen falta tantos procesos servidor *postgres* como clientes haya. El proceso **postmaster** es el encargado de ejecutar un nuevo servidor para cada cliente que solicite una conexión.

Se llama **sitio** al equipo anfitrión (*host*) que almacena un conjunto de bases de datos PostgreSQL. En un *sitio* se ejecuta solamente un proceso *postmaster* y múltiples procesos *postgres*. Los clientes pueden ejecutarse en el mismo sitio o en equipos remotos conectados por TCP/IP.

Es posible restringir el acceso a usuarios o a direcciones IP modificando las opciones del archivo pg_hba.conf, que se encuentra en /etc/postgresql/pg_hba.conf.

Este archivo, junto con /etc/postgresql/postgresql.conf son particularmente importantes, porque algunos de sus parámetros de configuración por defecto

provocan multitud de problemas al conectar inicialmente y porque en ellos se especifican los mecanismos de autenticación que usará PostgreSQL para verificar las credenciales de los usuarios.

Para habilitar la conexión a PostgreSQL desde clientes remotos, debemos verificar el parámetro tcpip_socket = true en el fichero */etc/postgresql/postgresql.conf*.

A continuación, para examinar los métodos de autenticación y las posibilidades de conexión de clientes externos, debemos mirar el fichero */etc/postgresql/pg_hba.conf*, donde se explicita la acción que hay que emprender para cada conexión proveniente de cada *host* externo, o grupo de *hosts*.

7.3. El administrador de postgres

Al terminar la instalación, en el sistema operativo se habrá creado el usuario **postgres**, y en PostgreSQL se habrá creado un usuario con el mismo nombre.

Él es el único usuario existente en la base de datos y será el único que podrá crear nuevas bases de datos y nuevos usuarios.

Normalmente, al usuario *postgres* del sistema operativo no se le permitirá el acceso desde un *shell* ni tendrá contraseña asignada, por lo que deberemos convertirnos en el usuario *root*, para después convertirnos en el usuario *postgres* y realizar tareas en su nombre:

```
yo@localhost:~$ su
Password:
# su - postgres
postgres@localhost:~$
```

El usuario **postgres** puede crear nuevas bases de datos utilizando el comando **createdb**. En este caso, le indicamos que el usuario propietario de la misma será el usuario postgres:

```
postgres@localhost:~$ createdb demo --owner=postgres
create database
```

El usuario **postgres** puede crear nuevos usuarios utilizando el comando **createuser**:

Se ha creado el usuario yo con permisos para crear bases de datos y sin permisos para crear usuarios.

```
postgres@localhost:~$ createuser yo
Shall the new user be allowed to create databases? (y/n) y
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

Los siguientes comandos eliminan bases de datos y usuarios, respectivamente:

```
postgres@localhost:~$ dropdb demo
postgres@localhost:~$ dropuser yo
```

Es recomendable que se agreguen los usuarios necesarios para operar la instalación de PostgreSQL, y recurrir, así, lo menos posible al ingreso con *postgres*.

También disponemos de sentencias SQL para la creación de usuarios, grupos y privilegios:

```
demo=# create user marc password 'marc21';
CREATE USER
demo=# alter user marc password 'marc22';
ALTER USER
demo=# drop user marc;
DROP USER
```

Los grupos permiten asignar privilegios a varios usuarios y su gestión es sencilla:

```
create group migrupo;
```

Para añadir o quitar usuarios de un grupo, debemos usar:

```
alter group migrupo add user marc, ... ;
alter group migrupo drop user marc, ... ;
```

7.3.1. Privilegios

Cuando se crea un objeto en PostgreSQL, se le asigna un dueño. Por defecto, será el mismo usuario que lo ha creado. Para cambiar el dueño de una tabla, índice, secuencia, etc., debemos usar el comando *alter table*. El dueño del objeto es el único que puede hacer cambios sobre él, si queremos cambiar este comportamiento, deberemos asignar privilegios a otros usuarios.

Los privilegios se asignan y eliminan mediante las sentencias `grant` y `revoke`. PostgreSQL define los siguientes tipos de operaciones sobre las que podemos dar privilegios:

select, insert, update, delete, rule, references, trigger, create, temporary, execute, usage, y all privileges.

Presentamos algunas sentencias de trabajo con privilegios, que siguen al pie de la letra el estándar SQL:

```
grant all privileges on proveedores to marc;
grant select on precios to manuel;
grant update on precios to group migrupo;
revoke all privileges on precios to manuel;
grant select on ganacias from public;
```

7.4. Creación de tipos de datos

Entre las múltiples opciones para extender PostgreSQL, nos queda aún por ver la creación de tipos o dominios (según la nomenclatura del estándar SQL). PostgreSQL prevé dos tipos de datos definidos por el administrador:

- Un tipo de datos compuesto, para utilizar como tipo de retorno en las funciones definidas por el usuario.

- Un tipo de datos simple, para utilizar en las definiciones de columnas de las tablas.

A modo de ejemplo, veamos un tipo compuesto y la función que lo devuelve:

```
create type comptipo as (f1 int, f2 text);
create function gettipo() returns setof comptipo as
  'select id, nombre from clientes' language sql;
```

Para el tipo de datos simple, la definición es más compleja, pues se debe indicar a PostgreSQL funciones que tratan con este tipo que le permitirán usarlo en operaciones, asignaciones, etc.

A modo de ejemplo, vamos a crear el tipo "numero complejo", tal como lo hace la documentación de PostgreSQL. En primer lugar, debemos definir la estructura donde almacenaremos el tipo:

Tratar con el tipo de datos simple

Habitualmente, las funciones que tratarán con este tipo de datos se escribirán en C.

```
typedef struct Complex {
   double     x;
   double     y;
} Complex;
```

Después, las funciones que lo recibirán o devolverán:

```
PG_FUNCTION_INFO_V1(complex_in);
Datum
complex_in(PG_FUNCTION_ARGS)
{
   char    *str = PG_GETARG_CSTRING(0);
   double  x,
           y;
   Complex *result;

   if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
      ereport(ERROR,
           (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
           errmsg("invalid input syntax for complex: \"%s\"", str)));

   result = (Complex *) palloc(sizeof(Complex));
   result->x = x;
   result->y = y;
   PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
   Complex *complex = (Complex *) PG_GETARG_POINTER(0);
   char    *result;

   result = (char *) palloc(100);
   snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
   PG_RETURN_CSTRING(result);
}
```

Ahora estamos en condiciones de definir las funciones, y el tipo:

```
create function complex_in(cstring)
   returns complex
   as 'filename'
   language c immutable strict;

create function complex_out(complex)
   returns cstring
   as 'filename'
   language c immutable strict;

create type complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   alignment = double
);
```

El proceso es un poco farragoso, pero compensa por la gran flexibilidad que aporta al SGBD. A continuación, podríamos crear algunos operadores para trabajar con este tipo (suma, resta, etc.), mediante los pasos que ya conocemos.

La creación de un dominio en PostgreSQL consiste en un tipo (definido por el usuario o incluido en el SGBD), más un conjunto de restricciones. La sintaxis es la siguiente:

```
create domain country_code char(2) not null;
create domain complejo_positivo complex not null check
       (complejo_mayor(value,(0,0)))
```

Se ha creado un dominio basado en un tipo definido por el sistema donde la única restricción es su longitud.

Evidentemente, deberíamos haber definido el operador *complejo_mayor* que recibiera dos números complejos e indicara si el primero es mayor que el segundo.

7.5. Plantilla de creación de bases de datos

PostgreSQL tiene definidas dos bases de datos de sistema, *template0* y *template1* (que habremos visto en los ejemplos al listar las bases de datos del gestor), con un conjunto de objetos tales como los tipos de datos que soporta o los lenguajes de procedimiento instalados.

La base de datos *template0* se crea al instalar el servidor, con los objetos por defecto del mismo, y la base de datos *template1* se crea a continuación de la anterior con algunos objetos más particulares del entorno o sistema operativo donde se ha instalado PostgreSQL. Esto hace que sea muy recomendable heredar siempre de *template1* (que es el comportamiento por defecto).

Estas bases de datos se utilizan como "padres" del resto de bases de datos que se crean, de modo que, al crear una nueva base de datos, se copian todos los objetos de *template1*.

Ventajas de esta situación

Si queremos añadir algún objeto (una tabla, un tipo de datos, etc.) a todas las bases de datos, sólo tenemos que añadirlo a *template1*, y el resto de bases de datos que se hayan creado a partir de ésta lo tendrán disponible.

Si se instala un lenguaje sobre la base de datos *template1*, automáticamente todas las bases de datos también usarán el lenguaje. En las distribuciones de Linux es frecuente que se haya realizado de este modo, por lo que no hay necesidad de instalarlo.

Por supuesto, podemos escoger otra plantilla para crear bases de datos, especificándola en la sentencia:

```
create database nuevbd template plantillabd
```

7.6. Copias de seguridad

Hacer periódicamente copias de seguridad de la base de datos es una de las tareas principales del administrador de cualquier base de datos. En PostgreSQL, estas copias de seguridad se pueden hacer de dos maneras distintas:

- Volcando a fichero las sentencias SQL necesarias para recrear las bases de datos.
- Haciendo copia a nivel de fichero de la base de datos.

En el primer caso, disponemos de la utilidad pg_dump, que realiza un volcado de la base de datos solicitada de la siguiente manera:

```
$ pg_dump demo > fichero_salida.sql
```

pg_dump es un programa cliente de la base de datos (como **psql**), lo que significa que podemos utilizarlo para hacer copias de bases de datos remotas, siempre que tengamos privilegios para acceder a todas sus tablas. En la práctica, esto significa que debemos ser el usuario administrador de la base de datos para hacerlo.

Si nuestra base de datos usa los OID para referencias entre tablas, debemos indicárselo a **pg_dump** para que los vuelque también (`pg_dump -o`) en lugar de volver a crearlos cuando inserte los datos en el proceso de recuperación. Asimismo, si tenemos BLOB en alguna de nuestras tablas, también debemos indicárselo con el parámetro correspondiente (`pg_dump -b`) para que los incluya en el volcado.

Más información

Hay otras opciones interesantes que podemos consultar mediante el parámetro `-h`.

Para restaurar un volcado realizado con **pg_dump**, podemos utilizar directamente el cliente **psql**:

```
$ psql demo < fichero_salida.sql
```

Una vez recuperada una base de datos de este modo, se recomienda ejecutar la sentencia `analyze` para que el optimizador interno de consultas de PostgreSQL vuelva a calcular los índices, la densidad de las claves, etc.

pg_dump

pg_dump realiza la copia a partir de la base de datos de sistema *template0*, por lo que también se volcarán los tipos definidos, funciones, etc. de la base de datos. Cuando recuperemos esta base de datos, debemos crearla a partir de *template0* si hemos personalizado *template1* (y no de *template1* como lo haría por defecto la sentencia `create database`) para evitar duplicidades.

Las facilidades del sistema operativo Unix, permiten copiar una base de datos a otra en otro servidor de la siguiente manera:

```
$ pg_dump -h host1 demo | psql -h host2 demo
```

Para hacer la copia de seguridad a nivel de fichero, simplemente copiamos los ficheros binarios donde PostgreSQL almacena la base de datos (especificado en

tiempo de compilación, o en paquetes binarios, suele ser */var/lib/postgres/data*), o bien hacemos un archivo comprimido con ellos:

```
$ tar -cvzf copia_bd.tar.gz /var/lib/postgres/data
```

El servicio PostgreSQL debe estar parado antes de realizar la copia.

A menudo, en bases de datos grandes, este tipo de volcados origina ficheros que pueden exceder los límites del sistema operativo. En estos casos tendremos que utilizar técnicas de creación de volúmenes de tamaño fijo en los comandos **tar** u otros con los que estemos familiarizados.

7.7. Mantenimiento rutinario de la base de datos

Hay una serie de actividades que el administrador de un sistema gestor de bases de datos debe tener presentes constantemente, y que deberá realizar periódicamente. En el caso de PostgreSQL, éstas se limitan a un mantenimiento y limpieza de los identificadores internos y de las estadísticas de planificación de las consultas, a una reindexación periódica de las tablas, y al tratamiento de los ficheros de registro.

7.7.1. *vacuum*

El proceso que realiza la limpieza de la base de datos en PostgreSQL se llama *vacuum*. La necesidad de llevar a cabo procesos de *vacuum* periódicamente se justifica por los siguientes motivos:

- Recuperar el espacio de disco perdido en borrados y actualizaciones de datos.
- Actualizar las estadísticas de datos utilizados por el planificador de consultas SQL.
- Protegerse ante la pérdida de datos por reutilización de identificadores de transacción.

Para llevar a cabo un *vacuum*, deberemos ejecutar periódicamente las sentencias **vacuum** y **analyze**. En caso de que haya algún problema o acción adicional a realizar, el sistema nos lo indicará:

```
demo=# VACUUM;
WARNING: some databases have not been vacuumed in 1613770184 transactions
HINT: Better vacuum them within 533713463 transactions, or you may have a wraparound failure.
```

```
VACUUM
demo=# VACUUM VERBOSE ANALYZE;
INFO: haciendo vacuum a "public.ganancia"
INFO: "ganancia": se encontraron 0 versiones de filas eliminables y 2 no eliminables en 1 páginas
DETAIL: 0 versiones muertas de filas no pueden ser eliminadas aún.
Hubo 0 punteros de ítem sin uso.
0 páginas están completamente vacías.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO: analizando "public.ganancia"
INFO: "ganancia": 1 páginas, 2 filas muestreadas, se estiman 2 filas en total
VACUUM
```

7.7.2. Reindexación

La reindexación completa de la base de datos no es una tarea muy habitual, pero puede mejorar sustancialmente la velocidad de las consultas complejas en tablas con mucha actividad.

```
demo=# reindex database demo;
```

7.7.3. Ficheros de registro

Es una buena práctica mantener archivos de registro de la actividad del servidor. Por lo menos, de los errores que origina. Durante el desarrollo de aplicaciones puede ser muy útil disponer también de un registro de las consultas efectuadas, aunque en bases de datos de mucha actividad, disminuye el rendimiento del gestor y no es de mucha utilidad.

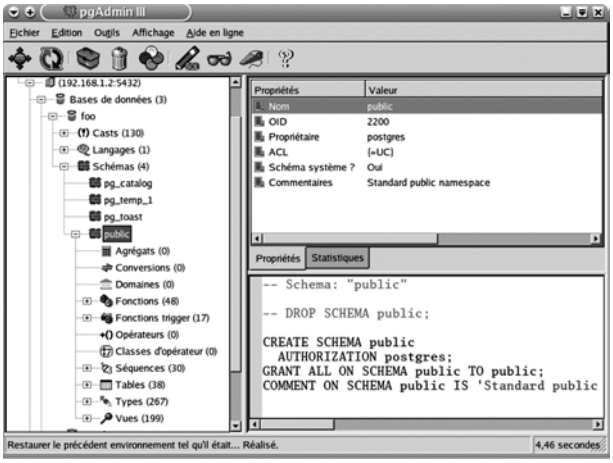
En cualquier caso, es conveniente disponer de mecanismos de rotación de los ficheros de registro; es decir, que cada cierto tiempo (12 horas, un día, una semana...), se haga una copia de estos ficheros y se empiecen unos nuevos, lo que nos permitirá mantener un histórico de éstos (tantos como ficheros podamos almacenar según el tamaño que tengan y nuestras limitaciones de espacio en disco).

PostgreSQL no proporciona directamente utilidades para realizar esta rotación, pero en la mayoría de sistemas Unix vienen incluidas utilidades como *logrotate* que realizan esta tarea a partir de una planificación temporal.

8. Cliente gráfico: pgAdmin3

El máximo exponente de cliente gráfico de PostgreSQL es el software pgAdmin3 que tiene licencia "Artist License", aprobada por la FSF.

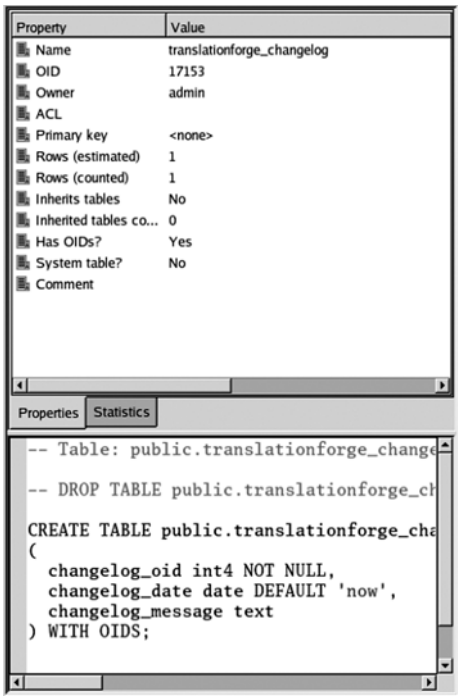
pgAdmin3 está disponible en http://www.pgadmin.org.



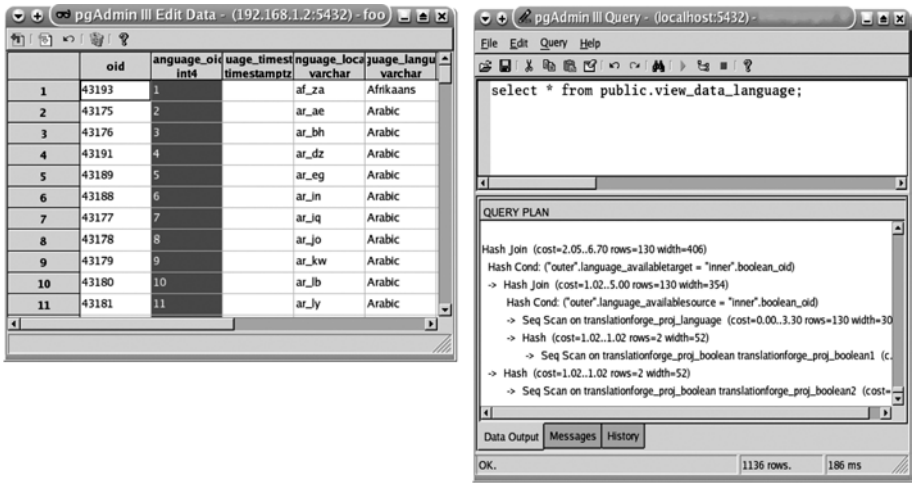
En pgAdmin3 podemos ver y trabajar con casi todos los objetos de la base de datos, examinar sus propiedades y realizar tareas administrativas.

- Agregados
- Casts
- Columnas
- Constraints
- Conversiones
- Bases de datos
- Dominios
- Funciones
- Grupos
- Índices
- Lenguajes (PLpgsql, PLpython, PLperl, etc.)
- Clases de operadores
- Operadores
- Servidores PostgreSQL
- Reglas
- Esquemas
- Secuencias
- Tablas
- Triggers
- Tipos de datos
- Usuarios
- Vistas

Una característica interesante de pgAdmin3 es que, cada vez que realizamos alguna modificación en un objeto, escribe la/s sentencia/s SQL correspondiente/s, lo que hace que, además de una herramienta muy útil, sea a la vez didáctica.



pgAdmin3 también incorpora funcionalidades para realizar consultas, examinar su ejecución (como el comando **explain**) y trabajar con los datos.



Todas estas características hacen de pgAdmin3 la única herramienta gráfica que necesitaremos para trabajar con PostgreSQL, tanto desde el punto de vista del usuario como del administrador. Evidentemente, las acciones que podemos realizar en cada momento vienen condicionadas por los permisos del usuario con el que nos conectemos a la base de datos.

Resumen

PostgreSQL implementa las características necesarias para competir con cualquier otra base de datos comercial, con la ventaja de tener una licencia de libre distribución BSD.

La migración de bases de datos alojadas en productos comerciales a PostgreSQL se facilita gracias a que soporta ampliamente el estándar SQL. PostgreSQL cuenta con una serie de características atractivas como son la herencia de tablas (clases), un rico conjunto de tipos de datos que incluyen arreglos, BLOB, tipos geométricos y de direcciónes de red. PostgreSQL incluye también el procesamiento de transacciones, integridad referencial y procedimientos almacenados. En concreto, hay procedimientos documentados para migrar los procedimientos almacenados desarrollados en lenguajes propietarios de bases de datos comerciales (PL/SQL) a PL/PGSQL.

La API se distribuye para varios lenguajes de programación como C/C++, Perl, PHP, Python, TCL/Tk y ODBC.

Por si esto fuera poco PostgreSQL es extensible. Es posible agregar nuevos tipos de datos y funciones al servidor que se comporten como los ya incorporados. También es posible insertar nuevos lenguajes de programación del lado del servidor para la creación de procedimientos almacenados. Todas estas ventajas hacen que muchos programadores lo elijan para el desarrollo de aplicaciones en todos los niveles.

Entre sus deficiencias principales podemos mencionar los OID. PostgreSQL está aún en evolución, se espera que en futuras versiones se incluyan nuevas características y mejoras al diseño interno del SGBD.

Bibliografía

Documentación de PostgreSQL de la distribución: http://www.postgresql.org/docs/

Silberschatz, A.; Korth, H.; Sudarshan,S. (2002). *Fundamentos de bases de datos* (4.ª ed.). Madrid: McGraw Hill.

Worsley, John C.; Drake, Joshua D. (2002). *Practical PostgreSQL*. O'Reilly.

Desarrollo de aplicaciones en conexión con bases de datos

Marc Gibert Ginestà

UOC
Formación de Posgrado

Índice

Introducción

Un curso de bases de datos quedaría incompleto si únicamente viéramos el funcionamiento y administración de los dos gestores anteriormente comentados. Uno de los principales objetivos de un SGBD es proporcionar un sistema de almacenamiento y consulta de datos al que se puedan conectar las aplicaciones que desarrollemos.

Así pues, en este capítulo vamos a abordar este tema, desde una perspectiva totalmente práctica, intentando exponer las bases para usar los SGBD vistos anteriormente desde algunos de los lenguajes de programación y conectores más usados. Los ejemplos proporcionados serán lo más simples posible para centrarnos en el tema que nos ocupa y no en las particularidades del lenguaje de programación en sí.

En primer lugar, veremos las herramientas que ofrece PHP para conectarse con bases de datos, y proporcionaremos algunos ejemplos.

A continuación, pasaremos a examinar la conexión JDBC a SGBD en general y a MySQL y PostgreSQL en particular, proporcionando también los ejemplos necesarios. También comentaremos algún aspecto avanzado como el de la persistencia de la conexión al SGBD.

Objetivos

El objetivo principal de esta unidad es conocer las diferentes técnicas de conexión a bases de datos que ofrecen PHP y Java.

Más concretamente, los objetivos que deberíais alcanzar al acabar el trabajo con la presente unidad son los siguientes:

- Conocer las posibilidades que PHP y Java ofrecen para la conexión y uso de bases de datos en general, y de MySQL y PostgreSQL en particular.

- Saber adaptar los programas desarrollados en estos lenguajes para que utilicen SGBD.

1. Conexión y uso de bases de datos en lenguaje PHP

El lenguaje de *script* PHP se ha popularizado extraordinariamente durante los últimos años, gracias a su sencillez y su sintaxis heredada de otros lenguajes como C, Perl o Visual Basic, que casi todos los desarrolladores ya conocían en mayor o menor grado.

Su fácil integración con los servidores web más populares (Apache, IIS, etc.), sin necesidad de recompilaciones o configuraciones complejas, ha contribuido también a que casi todos los proveedores de espacio web, desarrolladores de aplicaciones de software libre basadas en web, y proveedores de aplicaciones empresariales, lo usen para sus productos.

A lo largo de su corta historia ha progresado significativamente, y la versión 5.0 supone un paso adelante en la orientación a objetos, el tratamiento de excepciones y el trabajo con XML, lo que le hace parecerse en prestaciones a los lenguajes más maduros en el ámbito empresarial.

La versión 5.0 era la más actualizada en el momento de confección del presente material (finales del 2004). En este capítulo, no obstante, trabajaremos con la versión 4, ya que la versión 5.0 es muy reciente y los cambios que incorpora en cuanto a configuración, modelo de programación, etc. no les serán familiares a la mayoría de los estudiantes con conocimientos de PHP.

1.1. API nativa frente a API con abstracción

Desde la versión 2.0, PHP ha incorporado de forma nativa funciones para la conexión y uso de bases de datos. Al ser la rapidez una de las máximas de este lenguaje, y ante la ventaja de que proporciona mecanismos para la carga de librerías externas, se crearon unas librerías para cada motor de base de datos, que contenían las funciones necesarias para trabajar con él.

Estas API nativas son diferentes para cada SGBD, tanto en los nombres de las funciones que se utilizan para crear una conexión a la base de datos, lanzar una consulta, etc., como en el tratamiento de errores, resultados, etc.

Como actualmente hay aplicaciones web desarrolladas en PHP que usan la API concreta del SGBD para el que fueron pensadas, las revisaremos en este apartado.

Aunque se puede argumentar que al usar la API del SGBD concreto que utilicemos, dispondremos de operadores o funcionalidades específicas de ese motor que una librería estándar no puede proporcionar, con el paso del tiempo se ha visto que la utilización de estas API sólo está indicada (y aun así, no es recomendable) para aplicaciones que sepamos seguro que no van a cambiar el

SGBD con el que trabajan, ya que la revisión del código PHP, cuando hay un cambio de SGBD, es muy costosa y proclive a errores.

1.2. API nativa en MySQL

Para trabajar con la API nativa de MySQL en PHP, deberemos haber compilado el intérprete con soporte para este SGBD, o bien disponer ya del binario de PHP precompilado con el soporte incorporado.

En el caso de tenerlo que compilar, únicamente deberemos indicar como opción `--with-mysql`. Posteriormente, o en el caso de que ya dispongamos del binario, podemos validar que el soporte para MySQL está incluido correctamente en el intérprete con la ejecución del siguiente comando:

```
$ php -i | grep MySQL
supported databases => MySQL
MySQL Support => enabled
$
```

A partir de aquí, PHP proporciona unos parámetros de configuración que nos permitirán controlar algunos aspectos del funcionamiento de las conexiones con el SGBD, y las propias funciones de trabajo con la base de datos.

En cuanto a los parámetros, deberán situarse en el fichero `php.ini`, o bien configurarse para nuestra aplicación en concreto desde el servidor web. Destacan los siguientes:

`mysql.allow_persistent`: indica si vamos a permitir conexiones persistentes a MySQL. Los valores posibles son `true` o `false`.

`mysql.max_persistent`: número máximo de conexiones persistentes permitidas por proceso.

`mysql.max_links`: número máximo de conexiones permitidas por proceso, incluyendo las persistentes.

`mysql.connect_timeout`: tiempo que ha de transcurrir, en segundos, antes de que PHP abandone el intento de conexión al servidor.

Las conexiones persistentes son conexiones a la base de datos que se mantienen abiertas para evitar el tiempo de latencia que se pierde en conectar y desconectar. El intérprete, al ejecutar la sentencia de conexión a la base de datos, examina si hay alguna otra conexión abierta sin usar, y devuelve ésta en lugar de abrir una nueva. Lo mismo sucede al desconectar, el intérprete puede realizar la desconexión si hay suficientes conexiones aún abiertas, o bien mantener la conexión abierta para futuras consultas.

Por lo que respecta a la utilización de la API para la conexión y consulta de bases de datos, empezaremos con un ejemplo:

```
1  <?php
2  // Conectando y eligiendo la base de datos con que vamos a trabajar
3  $link = mysql_connect('host_mysql', 'usuario_mysql', 'password_mysql') or die
('No puedo conectarme: ' . mysql_error());
4  echo 'Conexión establecida';
5  mysql_select_db('mi_database',$link) or die('No he podido acceder a la base de
datos');
6
7  // Realizando una consulta SQL
8  $query = 'SELECT * FROM mi_tabla';
9  $result = mysql_query($query,$link) or die('Consulta errónea: ' . mysql_error());
10
11 // Mostramos los resultados en HTML
12 echo "<table>\n";
13 while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
14    echo "\t<tr>\n";
15    foreach ($line as $col_value) {
16    echo "\t\t<td>$col_value</td>\n";
17    }
18 echo "\t</tr>\n";
19 }
20 echo "</table>\n";
21
22 // Liberamos el resultset
23 mysql_free_result($result);
24
25 // Cerramos la conexión
26 mysql_close($link);
27 ?>
```

En las cinco primeras líneas se establece la conexión y se selecciona la base de datos con que se va a trabajar. El código es bastante explícito y la mayoría de errores al respecto suelen deberse a una mala configuración de los permisos del usuario sobre la base de datos con la que debe trabajar.

Conviene estar muy atento, sobre todo a las direcciones de origen de la conexión, ya que, aunque podemos usar `localhost` como nombre de equipo, si el intérprete y el SGBD están en el mismo servidor, suele ocurrir que PHP resuelve `localhost` al nombre real del equipo e intenta conectarse con esta identificación. Así pues, debemos examinar cuidadosamente los archivos de registro de MySQL y los usuarios y privilegios del mismo si falla la conexión.

Para establecer una conexión persistente, debemos utilizar la función `mysql_pconnect()` con los mismos parámetros.

A continuación, se utiliza la función `mysql_query()` para lanzar la consulta a la base de datos.

Utilidad de la sentencia de conexión

Hemos proporcionado la sentencia SQL a la función y el enlace nos ha devuelto la sentencia de conexión. Esta funcionalidad es absolutamente necesaria si se trabaja con varias conexiones simultáneamente, si únicamente hay una conexión establecida en el *script*, este parámetro es opcional.

La función `mysql_error()` es universal, y devuelve el último error ocurrido en el SGBD con nuestra conexión, o con la conexión `$link` que le indiquemos como parámetro.

La función `mysql_query()` puede devolver los siguientes resultados:

- FALSE si ha habido un error.
- Una referencia a una estructura si la sentencia es de consulta y ha tenido éxito.
- TRUE si la sentencia es de actualización, borrado o inserción y ha tenido éxito.

La función `mysql_affected_rows()` nos permite conocer el número de filas que se han visto afectadas por sentencias de actualización, borrado o inserción.

La función `mysql_num_rows()` nos permite conocer el número de filas devuelto por sentencias de consulta.

Una vez obtenido el recurso a partir de los resultados de la consulta, PHP proporciona multitud de formas de iterar sobre sus resultados o de acceder a uno de ellos directamente. Comentamos las más destacadas:

- `$fila = mysql_fetch_array($recurso, <tipo_de_array>)`

  Esta función va iterando sobre el recurso, devolviendo una fila cada vez, hasta que no quedan más filas, y devuelve FALSE. La forma del *array* devuelto dependerá del parámetro `<tipo_de_array>` que puede tomar estos valores:

  – MYSQL_NUM: devuelve un *array* con índices numéricos para los campos. Es decir, en `$fila[0]` tendremos el primer campo del SELECT, en `$fila[1]`, el segundo, etc.

  – MYSQL_ASSOC: devuelve un *array* asociativo en el que los índices son los nombres de campo o alias que hayamos indicado en la sentencia SQL.

  – MYSQL_BOTH: devuelve un *array* con los dos métodos de acceso.

Recuperando el ejemplo inicial, entre las líneas 7 y 21:

```
7  // Realizando una consulta SQL
8  $query = 'SELECT * FROM mi_tabla';
9  $result = mysql_query($query,$link) or die('Consulta errónea: ' . mysql_error());
10
11 // Mostramos los resultados en HTML
12 echo "<table>\n";
13 while ($line = mysql_fetch_array($result, MYSQL_BOTH)) {
14   echo "\t<tr>\n";
15   for($i=0;$i<sizeof($line);$i++) {
16     echo "\t\t<td>$line[$i]</td>\n";
17   }
18   echo "<td>Nombre: $line['nombre']</td>";
19   echo "\t</tr>\n";
20 }
21 echo "</table>\n";
```

- `$objeto = mysql_fetch_object($recurso)`

  Esta función va iterando sobre los resultados, devolviendo un objeto cada vez, de manera que el acceso a los datos de cada campo se realiza a través de las propiedades del objeto. Al igual que en el *array* asociativo, hay que vigilar con los nombres de los campos en consulta, evitando que devuelva campos con el mismo nombre fruto de combinaciones de varias tablas, ya que sólo podremos acceder al último de ellos:

  Volvemos sobre el ejemplo inicial

```
7  // Realizando una consulta SQL
8  $query = 'SELECT nombre,apellidos FROM mi_tabla';
9  $result = mysql_query($query,$link) or die('Consulta errónea: ' . mysql_error());
10
11 // Mostramos los resultados en HTML
12 echo "<table>\n";
13 while ($object = mysql_fetch_array($result, MYSQL_BOTH)) {
14   echo "\t<tr>\n";
15   echo "<td>Nombre: " . $object->nombre . "</td>";
16   echo "<td>Apellidos: " . $object->apellidos . "</td>";
17   echo "\t</tr>\n";
18 }
19 echo "</table>\n";
```

- `$valor = mysql_result($recurso,$numero_de_fila,$numero_de_campo)`

  Esta función consulta directamente un valor de un campo de una fila especificada. Puede ser útil si queremos conocer un resultado sin necesidad de realizar bucles innecesarios. Evidentemente, hemos de saber exactamente dónde se encuentra.

- `$exito = mysql_data_seek($recurso,$fila)`

  Esta función permite mover el puntero dentro de la hoja de resultados representada por `$recurso`, hasta la fila que deseemos. Puede ser útil para

avanzar o para retroceder y volver a recorrer la hoja de resultados sin tener que ejecutar la sentencia SQL de nuevo. Si la fila solicitada no existe en la hoja de resultados, el resultado será FALSE, al igual que si la hoja no contiene ningún resultado. Es decir, el valor de `$fila` debe estar entre 0 (la primera fila) y `mysql_num_rows()-1`, excepto cuando no hay ningún resultado, en cuyo caso devolverá FALSE.

Finalmente, comentaremos las funciones de liberación y desconexión. En el primer caso, PHP realiza un excelente trabajo liberando recursos de memoria cuando la ejecución en curso ya no los va a utilizar más. Aun así, si la consulta devuelve una hoja de datos muy grande, puede ser conveniente liberar el recurso cuando no lo necesitemos.

Por lo que respecta al cierre de la conexión, tampoco suele ser necesario, ya que PHP cierra todas las conexiones al finalizar la ejecución y, además, el cierre siempre está condicionado a la configuración de las conexiones persistentes. Tal como ya hemos comentado, si activamos las conexiones persistentes (o bien hemos conectado con `mysql_pconnect`), esta función no tiene ningún efecto y, en todo caso, será PHP quien decida cuándo se va a cerrar cada conexión.

Ya hemos comentado que las API específicas para cada motor incluyen un conjunto de funciones que podía ayudar a trabajar con sus aspectos particulares, a continuación, enumeramos las más importantes:

- `mysql_field_flags`, `mysql_field_name`, `mysql_field_table`, `mysql_field_type`: estas funciones reciben como parámetro un `$recurso` y un índice de campo dentro de la consulta ejecutada y devuelven información sobre el campo; en concreto, sus restricciones, nombre, tabla a la que corresponden y tipo de campo. Pueden ser muy útiles para trabajar con consultas genéricas sobre bases de datos y/o campos que no conocemos al realizar el *script*.

- `mysql_insert_id`: esta función devuelve el último identificador obtenido de una inserción en un campo autoincremental.

- `mysql_list_dbs`, `mysql_list_tables`, `mysql_list_fields`: con distintos parámetros, estas funciones permiten consultar datos de administración del motor de la base de datos.

Bibliografía

Hay otras funciones más específicas para obtener información sobre el cliente que origina la conexión, o sobre el propio servidor donde se está ejecutando. Conviene consultar la documentación para obtener información sobre usos más avanzados de esta API.

1.3. API nativa en PostgreSQL

Para trabajar con la API nativa de PostgreSQL en PHP, deberemos haber compilado el intérprete con soporte para este SGBD, o bien disponer ya del binario de PHP precompilado con el soporte incorporado.

En el caso de tenerlo que compilar, únicamente debemos indicar como opción `--with-pgsql`. Posteriormente, o en el caso de que ya dispongamos del binario, podemos validar que el soporte para PostgreSQL está incluido correctamente en el intérprete con la ejecución del siguiente comando:

```
$ php -i | grep PostgreSQL
PostgreSQL
PostgreSQL Support => enabled
PostgreSQL(libpq) Version => 7.4.6
$
```

A partir de aquí, PHP proporciona unos parámetros de configuración que nos permitirán controlar algunos aspectos del funcionamiento de las conexiones con el SGBD, y las propias funciones de trabajo con la base de datos.

En cuanto a los parámetros, deberán situarse en el fichero `php.ini` o bien configurarse para nuestra aplicación en concreto desde el servidor web. Destacan los siguientes:

- `pgsql.allow_persistent`: indica si vamos a permitir el uso de conexiones persistentes. Los valores son `true` o `false`.

- `pgsql.max_persistent`: número máximo de conexiones persistentes permitidas por proceso.

- `pgsql.max_links`: número máximo de conexiones permitidas por proceso, incluyendo las persistentes.

- `pgsql.auto_reset_persistent`: detecta automáticamente conexiones persistentes cerradas y las elimina.

Este parámetro disminuye ligeramente el rendimiento del sistema.

Por lo que respecta a la utilización de la API para la conexión y consulta de bases de datos, reproduciremos el anterior ejemplo:

```
1  <?php
2  // Conectando y eligiendo la base de datos con que vamos a trabajar
3  $link = pg_connect("host=host_pgsql port=5432 dbname=mi_database user=user_pgsql
password=pass_pgsql");
4  $stat = pg_connection_status($link);
5  if ($stat === 0) {
6     echo 'Conexión establecida';
```

```
7  } else {
8    die 'No puedo conectarme';
9  }
10
11 // Realizando una consulta SQL
12 $query = 'SELECT * FROM mi_tabla';
13 $result = pg_query($link,$query) or die('Consulta errónea: ' . pg_last_error());
14
15 // Mostramos los resultados en HTML
16 echo "<table>\n";
17 while ($line = pg_fetch_array($result, PGSQL_ASSOC)) {
18   echo "\t<tr>\n";
19   foreach ($line as $col_value) {
20   echo "\t\t<td>$col_value</td>\n";
21   }
22 echo "\t</tr>\n";
23 }
24 echo "</table>\n";
25
26 // Liberamos el resultset
27 pg_free_result($result);
28
29 // Cerramos la conexión
30 pg_close($link);
31 ?>
```

En las diez primeras líneas, establecemos la conexión y comprobamos que se ha realizado correctamente.

Recordad

La mayoría de los errores en este aspecto se originan por una mala configuración de los permisos de los usuarios en el SGBD. Una forma de probarlo que suele ofrecer más información es intentar la conexión con el cliente de la base de datos desde la línea de comandos, especificando el mismo usuario, base de datos y contraseña que estamos utilizando en el *script*.

A diferencia de MySQL, la selección de la base de datos se hace en el momento de la conexión. En cambio, la comprobación de la conexión es un poco más complicada.

Para establecer una conexión persistente, debemos utilizar la función `pg_pconnect()` con los mismos parámetros.

A continuación, se utiliza la función `pg_query()` para lanzar la consulta a la base de datos.

Utilidad de la sentencia de conexión

Aquí se aplican los mismos consejos que dábamos en el apartado anterior, y las mismas consideraciones en cuanto al parámetro `$link` y su opcionalidad si únicamente tenemos una conexión establecida con el mismo usuario y contraseña.

Para comprobar errores, la API de PostgreSQL distingue entre un error de conexión, y errores sobre los recursos devueltos. En el primer caso, deberemos usar `pg_connection_status()`, mientras que en el segundo podemos optar por `pg_last_error()` o bien `pg_result_error($recurso)` para obtener el mensaje de error que pueda haber devuelto un recurso en concreto.

La función pg_query() puede devolver los siguientes resultados:

- FALSE si ha habido un error.
- Una referencia a una estructura si la sentencia ha tenido éxito.

La función `pg_affected_rows($recurso)` nos permite conocer el número de filas que se han visto afectadas por sentencias de actualización, borrado o inserción. Esta función deberá recibir como parámetro el recurso devuelto por la función `pg_query()`.

La función `pg_num_rows($recurso)` nos permite conocer el número de filas devuelto por sentencias de consulta.

Una vez obtenido el recurso a partir de los resultados de la consulta, PHP proporciona multitud de formas de iterar sobre sus resultados o de acceder a uno de ellos directamente. Comentamos las más destacadas:

- `$fila = pg_fetch_array($recurso, <tipo_de_array>)`

  Esta función va iterando sobre el recurso, devolviendo una fila cada vez, hasta que no quedan más filas y devuelve FALSE. La forma del *array* devuelto, dependerá del parámetro `<tipo_de_array>` que puede tomar estos valores:

- PG_NUM: devuelve un *array* con índices numéricos para los campos. Es decir, en `$fila[0]` tendremos el primer campo del SELECT, en `$fila[1]`, el segundo, etc.

- PG_ASSOC: devuelve un *array* asociativo donde los índices son los nombres de campo o alias que hayamos indicado en la sentencia SQL.

- PG_BOTH: devuelve un *array* con los dos métodos de acceso.

Recuperando el ejemplo inicial, entre las líneas 11 y 24:

```
11 // Realizando una consulta SQL
12 $query = 'SELECT * FROM mi_tabla';
13 $result = pg_query($query,$link) or die('Consulta errónea: ' . pg_last_error());
14// Mostramos los resultados en HTML
15 echo "<table>\n";
16 while ($line = pg_fetch_array($result, PGSQL_BOTH)) {
17   echo "\t<tr>\n";
18   for($i=0;$i<sizeof($line);$i++) {
19     echo "\t\t<td>$line[$i]</td>\n";
20   }
21   echo "<td>Nombre: $line['nombre']</td>";
22   echo "\t</tr>\n";
23 }
24 echo "</table>\n";
```

- `$objeto = pg_fetch_object($recurso)`

  Esta función va iterando sobre los resultados, devolviendo un objeto cada vez, de forma que el acceso a los datos de cada campo se realiza por medio de las propiedades del objeto. Al igual que en el *array* asociativo, hay que vigilar con los nombres de los campos en consulta, evitando que devuelva campos con el mismo nombre fruto de combinaciones de varias tablas, ya que sólo podremos acceder al último de ellos.

  Volvemos sobre el ejemplo inicial

```
11 // Realizando una consulta SQL
12 $query = 'SELECT nombre,apellidos FROM mi_tabla';
13 $result = pg_query($query,$link) or die('Consulta errónea: ' . pg_last_error());
14 // Mostramos los resultados en HTML
15 echo "<table>\n";
16 while ($object = pg_fetch_array($result, PGSQL_BOTH)) {
17   echo "\t<tr>\n";
18   echo "<td>Nombre: " . $object->nombre . "</td>";
19   echo "<td>Apellidos: " . $object->apellidos . "</td>";
20   echo "\t</tr>\n";
21 }
22 echo "</table>\n";
```

  Podemos pasar a la función `pg_fetch_object()` un segundo parámetro para indicar la fila concreta que queremos obtener:

  `$resultado = pg_fetch_all($recurso)`

  Esta función devuelve toda la hoja de datos correspondiente a `$recurso`; es decir, una *array* con todas las filas y columnas que forman el resultado de la consulta.

- `$exito = pg_result_seek($recurso,$fila)`

  Esta función permite mover el puntero dentro de la hoja de resultados representada por `$recurso` hasta la fila que deseemos. Deben tomarse las mismas consideraciones que en la función `mysql_data_seek()`.

En cuanto a la liberación de recursos y la desconexión de la base de datos, es totalmente aplicable lo explicado para MySQL, incluyendo los aspectos relacionados con las conexiones persistentes.

Al igual que en MySQL, PHP también proporciona funciones específicas para trabajar con algunos aspectos particulares de PostgreSQL. Al tener éste más funcionalidad que se aleja de lo estándar debido a su soporte a objetos, estas

funciones cobrarán más importancia. A continuación comentamos las más destacadas:

- `pg_field_name`, `pg_field_num`, `pg_field_size`, `pg_field_type`: estas funciones proporcionan información sobre los campos que integran una consulta. Sus nombres son suficientemente explícitos acerca de su cometido.

- `pg_last_oid`: esta función nos devuelve el OID obtenido por la inserción de una tupla si el recurso que recibe como parámetro es el correspondiente a una sentencia INSERT. En caso contrario devuelve FALSE.

- `pg_lo_create`, `pg_lo_open`, `pg_lo_export`, `pg_lo_import`, `pg_lo_read`, `pg_lo_write`: estas funciones (entre otras) facilitan el trabajo con objetos grandes (LOB) en PostgreSQL.

```
<?php
  $database = pg_connect("dbname=jacarta");
  pg_query($database, "begin");
  $oid = pg_lo_create($database);
  echo "$oid\n";
  $handle = pg_lo_open($database, $oid, "w");
  echo "$handle\n";
  pg_lo_write($handle, "large object data");
  pg_lo_close($handle);
  pg_query($database, "commit");
?>
```

Las funciones `pg_lo_import` y `pg_lo_export` pueden tomar ficheros como parámetros, facilitando la inserción de objetos binarios en la base de datos.

Nota

Hay otras funciones que tienen el mismo cometido que combinaciones de algunas de las funciones comentadas anteriormente (por ejemplo, `pg_select` o `pg_insert`, `pg_copy_from`), pero que no se comentan en este material por su extensión y por su poco uso.

1.4. Capa de abstracción PEAR::DB

El PEAR (PHP *extension and application repository*) se define como un marco de trabajo y un sistema de distribución de librerías reutilizables para PHP. Es similar en concepto al CPAN (*comprehensive perl archive network*) del lenguaje Perl o al PyPI (*Python package index*) de Python.

El PEAR pretende proporcionar una librería estructurada de código y librerías reutilizables, mantener un sistema para proporcionar a la comunidad herramientas para compartir sus desarrollos y fomentar un estilo de codificación estándar en PHP.

PEAR

Si nuestra instalación de PHP es reciente, ya dispondremos de PEAR instalado (a no ser que lo hayamos compilado con la opción `--without-pear`.

PEAR, debe ser el primer recurso para solventar cualquier carencia detectada en las funciones nativas de PHP. Como buena práctica general en el mundo del software libre, siempre es mejor usar, aprender o mejorar a partir de lo que

han hecho otros, que proponernos reinventar la rueda. Además, si hacemos mejoras a las librerías que usemos de PEAR, siempre podemos contribuir a esos cambios mediante las herramientas que nos proporciona.

En cierta forma, PEAR se comporta como un gestor de paquetes más de los que pueden incorporar las distribuciones GNU/Linux más recientes (como apt, yum o YOU). Este gestor de paquetes se compone del ejecutable 'pear' al que podemos proporcionar un conjunto de parámetros según las acciones que deseemos realizar:

```
$ pear list
Installed packages:
===================
Package         Version   State
Archive_Tar     1.2       stable
Console_Getopt  1.2       stable
DB              1.6.8     stable
http            1.3.3     stable
Mail            1.1.4     stable
Net_SMTP        1.2.6     stable
Net_Socket      1.0.5     stable
PEAR            1.3.4     stable
PhpDocumentor   1.3.0RC3  beta
XML_Beautifier  1.1       stable
XML_Parser      1.2.2     stable
XML_RPC         1.1.0     stable
XML_Util        1.1.1     stable
```

PEAR (y PHP) ya viene con un conjunto de paquetes instalados, lo que se denomina el PFC (PHP *foundation classes*). Estos paquetes proporcionan a PHP la mínima funcionalidad necesaria para que PEAR funcione y para que dispongamos de las librerías básicas de PHP.

A continuación presentamos las opciones más habituales de PEAR:

Comando	Resultado
`pear list`	Lista de los paquetes instalados.
`pear list-all`	Lista de todos los paquetes disponibles en PEAR.
`pear list-upgrades`	Lista de los paquetes instalados con actualización disponible.
`pear info <paquete>`	Proporciona información sobre el paquete.
`pear install <paquete>`	Descarga e instala el paquete.
`pear search <texto>`	Busca paquetes en el repositorio PEAR.
`pear upgrade <paquete>`	Actualiza el paquete si es necesario.
`pear upgrade-all`	Actualiza todos los paquetes instalados con actualización disponible.
`pear uninstall <paquete>`	Desinstala el paquete.

1.4.1. Capa de abstracción del motor de la base de datos

Parece evidente que, para la inmensa mayoría de aplicaciones basadas en PHP, el uso de su librería nativa de acceso a bases de datos va a condicionar el SGBD a usar con la aplicación. En aplicaciones comerciales, o que no pueden ni desean estar cerradas a un único motor, no será imprescindible disponer de unas funciones que encapsulen la comunicación con el SGBD y que sean independientes de éste en las interfaces que ofrecen, mientras que internamente llamarán a las funciones nativas del SGBD concreto con que se esté trabajando en cada momento.

Así pues, y buscando en PEAR, encontramos el módulo 'DB', una capa de abstracción y encapsulamiento de la comunicación con el SGBD. Al tener que incorporar todas las funcionalidades de los motores que soporta, el resultado será siempre el mínimo conjunto de prestaciones comunes a todos los SGBD. Las prestaciones más destacadas que ofrece la versión actual 1.6.8 son las siguientes:

Versiones

La versión 1.6.8. era la más actualizada en el momento de elaboración de este material (finales de 2004).

- Interfaz orientada a objetos.
- Una sintaxis común para identificar SGBD y cadenas de conexión.
- Emulación de "sentencias preparadas" en los motores que no las soportan.
- Códigos de errores comunes.
- Emulación de secuencias o autoincrementos en SGBD que no los soportan.
- Soporte para transacciones.
- Interfaz para obtener información del metadato (información sobre la tabla o la base de datos).
- Compatible con PHP4 y PHP5.
- Motores soportados: dbase, fbsql, interbase, informix, msql, mssql, mysql, mysqli, oci8, odbc, pgsql, sqlite y sybase.

```
1 <?php
2 // Incluimos la librería una vez instalada mediante PEAR
3 require_once 'DB.php';
4
5 // Creamos la conexión a la base de datos, en este caso PostgreSQL
6 $db =& DB::connect('pgsql://usuario:password@servidor/basededatos');
7
8 // Comprobamos error en la conexión
9 if (DB::isError($db)) {
10  die($db->getMessage());
11 }
12
```

```
13 // Realizamos la consulta:
14 $res =& $db->query('SELECT * FROM clients');
15
16 // Comprobamos que la consulta se ha realizado correctamente
17 if (DB::isError($res)) {
18 die($res->getMessage());
19 }
20
21 // Iteramos sobre los resultados
22 while ($row =& $res->fetchRow()) {
23  echo $row[0] . "\n";
24 }
25
26 // Liberamos la hoja de resultados
27 $res->free()
28
29 // Desconectamos de la base de datos
30 $db->disconnect();
31 ?>
```

La estructura del código y hasta la sintaxis de las sentencias es similar a los ejemplos nativos vistos anteriormente, exceptuando las partes de las sentencias que hacían referencia al motor de base de datos en particular.

A continuación, vamos a avanzar por el código ampliando la información sobre cada paso.

La conexión se especifica mediante una sintaxis de tipo DSN (*data source name*). Los DSN admiten multitud de variantes, dependiendo del motor al que nos conectemos, pero en casi todos los casos, tienen la forma siguiente:

```
motorphp://usuario:contraseña@servidor/basededatos?opcion=valor
```

- Conexión a MySQL

  ```
  mysql://usuario:password@servidor/basededatos
  ```

- Conexión a MySQL a través de un *socket* UNIX:

  ```
  mysql://usuario:password@unix(/camino/al/socket)/basededatos
  ```

- Conexión a PostgreSQL

  ```
  pgsql://usuario:password@servidor/basededatos
  ```

- Conexión a PostgreSQL en un puerto específico:

  ```
  pgsql://usuario:password@tcp(servidor:1234)/basededatos
  ```

En cualquier llamada a un método del paquete DB, éste puede devolver el objeto que le corresponde (una hoja de resultados, un objeto representando la conexión, etc.) o bien un objeto que represente el error que ha tenido la llamada. De ésta manera, para comprobar los errores que puede originar cada sentencia o intento de conexión, bastará con comprobar el tipo del objeto devuelto:

```
8  // Comprobamos error en la conexión
9  if (DB::isError($db)) {
10   die($db->getMessage());
11 }
```

La clase `DB_Error` ofrece varios métodos. A pesar de que el más utilizado es `getMessage()`, `getDebugInfo()` o `getCode()` pueden ampliar la información sobre el error.

Para realizar consultas, disponemos de dos mecanismos diferentes:

- Enviar la consulta directamente al SGBD.
- Indicar al gestor que prepare la ejecución de una sentencia SQL y posteriormente indicarle que la ejecute una o más veces.

En la mayoría de los casos, optaremos por el primer mecanismo y podremos proceder como sigue:

```
13 // Realizamos la consulta:
14 $res =& $db->query('SELECT * FROM clients');
```

Ejemplo

Pensemos en un conjunto de actualizaciones o inserciones seguidas o en un conjunto de consultas donde vamos cambiando un intervalo de fechas o el identificador del cliente sobre el que se realizan.

En ocasiones nos podemos encontrar ejecutando la misma consulta varias veces, con cambios en los datos o en el valor de las condiciones. En estos casos, es más indicado utilizar el segundo mecanismo.

Supongamos que tenemos que insertar un conjunto de clientes en nuestra base de datos. Las sentencias que ejecutaríamos serían parecidas a las siguientes:

```
INSERT INTO Clientes (nombre, nif) VALUES ('José Antonio Ramírez','29078922Z');
INSERT INTO Clientes (nombre, nif) VALUES ('Miriam Rodríguez','45725248T');
...
```

En lugar de esto, podemos indicarle al motor de la base de datos que prepare la sentencia, del modo siguiente:

```
$sth = $db->prepare('INSERT INTO Clientes (nombre,nif) VALUES (?,?)');
```

Utilizaremos la variable `$sth` que nos ha devuelto la sentencia `prepare` cada vez que ejecutemos el *query*:

```
$db->execute($sth, 'José Antonio Ramírez','29078922Z');
$db->execute($sth, 'Miriam Rodriguez','45725248T');
```

También podemos pasar una *array* con todos los valores:

```
$datos = array('Miriam Rodriguez','45725248T');
$db->execute($sth, $datos);
```

Y tenemos la opción de pasar una *array* de dos dimensiones con todas las sentencias a ejecutar, mediante el método `executeMultiple()`:

```
$todosDatos = array(array('José Antonio Ramírez','29078922Z'),
        array('Miriam Rodriguez','45725248T'));
$sth = $db->prepare('INSERT INTO Clientes (nombre,nif) VALUES (?, ?)');
$db->executeMultiple($sth, $todosDatos);
```

A continuación, examinaremos los métodos de iterar sobre los resultados. En el ejemplo inicial hemos utilizado la función `fetchRow()` de la manera siguiente:

```
21 // Iteramos sobre los resultados
22 while ($row =& $res->fetchRow()) {
23   echo $row[0] . "\n";
24 }
```

Pero también disponemos de la función `fetchInto()`, que recibe como parámetro el *array* donde queremos que se almacene el resultado:

```
21 // Iteramos sobre los resultados
22 while ($res->fetchRow()) {
23   echo $row[0] . "\n";
24 }
```

Tanto `fetchRow()` como `fetchInto()` aceptan otro parámetro para indicar el tipo de estructura de datos que va a almacenar en `$row`:

- DB_FETCHMODE_ORDERED: es la opción por defecto. Almacena el resultado en una *array* con índice numérico.

- DB_FETCHMODE_ASSOC: almacena el resultado en una *array* asociativa en el que las claves son el nombre del campo.

- DB_FETCHMODE_OBJECT: almacena el resultado en un objeto donde dispondremos de atributos con el nombre de cada campo para obtener el valor en cada iteración.

```
21 // Iteramos sobre los resultados en modo asociativo
22 while ($res->fetchInto (($row,DB_FETCHMODE_ASSOC)) {
23    echo $row['nombre'] . "\n";
24    echo $row['nif'] . "\n";
25 }
```

o bien:

```
21 // Iteramos sobre los resultados en modo objeto
22 while ($res->fetchInto (($row,DB_FETCHMODE_OBJECT)) {
23    echo $row->nombre . "\n";
24    echo $row->nif . "\n";
25 }
```

La función `fetchInto()` acepta un tercer parámetro para indicar el número de fila que queremos obtener, en caso de que no deseemos iterar sobre la propia función:

```
21 // Obtenemos la tercera fila de la consulta
22 $res->fetchInto ($row,DB_FETCHMODE_ASSOC,3); {
23    echo $row->nombre . "\n";
24    echo $row->nif . "\n";
```

Hay otros métodos para obtener diferentes vistas del resultado como los siguientes:

- `getAll()`: obtiene una *array* de dos dimensiones con toda la hoja de resultados. Tendría una forma como la siguiente:

```
Array
(
   [0] => Array
      (
         [cf] => Juan
         [nf] => 5
         [df] => 1991-01-11 21:31:41
      )
   [1] => Array
   (
      [cf] => Kyu
      [nf] => 10
      [df] => 1992-02-12 22:32:42
   )
)
```

- `getRow()`: devuelve sólo la primera fila de la hoja de resultados.
- `getCol()`: devuelve sólo la columna indicada de la hoja de resultados.

De la misma manera que en las librerías nativas, hay métodos que proporcionan información sobre la consulta en sí:

- `numRows()`: número de filas de la hoja de resultados.
- `numCols()`: número de columnas de la hoja de resultados.
- `affectedRows()`: número de filas de la tabla afectadas por la sentencia de actualización, inserción o borrado.

1.4.2. Transacciones

PEAR::DB proporciona mecanismos para tratar las transacciones independientemente del SGBD con que trabajemos.

Como ya hemos comentado, la operativa con las transacciones está relacionada con las sentencias `begin`, `commit` y `rollback` de SQL. PEAR::DB envuelve estas sentencias en llamadas a métodos suyos, del modo siguiente:

```
// Desactivamos el comportamiento de COMMIT automático.
$db->autocommit(false);
..
..
if (...) {
   $db->commit();
} else {
   $db->rollback();
}
```

Atención

En MySQL sólo funcionará el soporte de transacciones si la base de datos está almacenada con el mecanismo InnoDB.

En PostgreSQL no hay restricción alguna.

En ningún sitio...

... se hace un `begin`. Al desactivar el `autocommit` (que está activado por defecto) todas las sentencias pasarán a formar parte de una transacción, que se registrará como definitiva en la base de datos al llamar al método `commit()` o bien se desechará al llamar al método `rollback()`, volviendo la base de datos al estado en el que estaba después del último `commit()`.

1.4.3. Secuencias

PEAR::DB incorpora un mecanismo propio de secuencias (AUTO_INCREMENT en MySQL), que es independiente de la base de datos utilizada y que puede ser de gran utilidad en identificadores, claves primarias, etc. El único requisito es que se usen sus métodos de trabajo con secuencias, siempre que se esté trabajando con esa base de datos; es decir, no se debe crear la secuencia en el SGBD y, después, trabajar con ella con los métodos que ofrece PEAR::DB. Si la secuencia la creamos mediante la base de datos, entonces deberemos trabajar con ella con las funciones extendidas de SQL que nos proporcione ese SGBD (en PostgreSQL la función `nextval()` o en MySQL la inserción del valor 0 en un campo AUTO_INCREMENT).

Disponemos de las siguientes funciones para trabajar con secuencias:

- `createSequence($nombre_de_secuencia)`: crea la secuencia o devuelve un objeto DB_Error en caso contrario.

- `nextId($nombre_de_secuencia)`: devuelve el siguiente identificador de la secuencia.

- `dropSequence($nombre_de_secuencia)`: borra la secuencia.

```
// Creamos la secuencia:
$tmp = $db->createSequence('miSecuencia');
if (DB::isError($tmp)) {
  die($tmp->getMessage());
}

// Obtenemos el siguiente identificador
$id = $db->nextId('mySequence');
if (DB::isError($id)) {
  die($id->getMessage());
}

// Usamos el identificador en una sentencia
$res =& $db->query("INSERT INTO miTabla (id, texto) VALUES ($id, 'Hola')");

// Borramos la secuencia
$tmp = $db->dropSequence('mySequence');

if (DB::isError($tmp)) {
  die($tmp->getMessage());
}
```

Finalmente, en el aspecto relacionado con los metadatos de las tablas, PEAR::DB ofrece la función `tableInfo()`, que proporciona información detallada sobre una tabla o sobre las columnas de una hoja de resultados obtenida de una consulta.

```
$info = $db->tableInfo('nombretabla');
print_r($info);
```

O bien:

```
$res =& $db->query('SELECT * FROM nombretabla');
$info = $db->tableInfo($res);
print_r($info);
```

El resultado será similar al siguiente:

```
[0] => Array (
  [table] => nombretabla
  [name] => nombre
```

```
    [type] => string
    [len] => 255
    [flags] =>
)
[1] => Array (
    [table] => nombretabla
    [name] => nif
    [type] => string
    [len] => 20
    [flags] => primary key not null
)
```

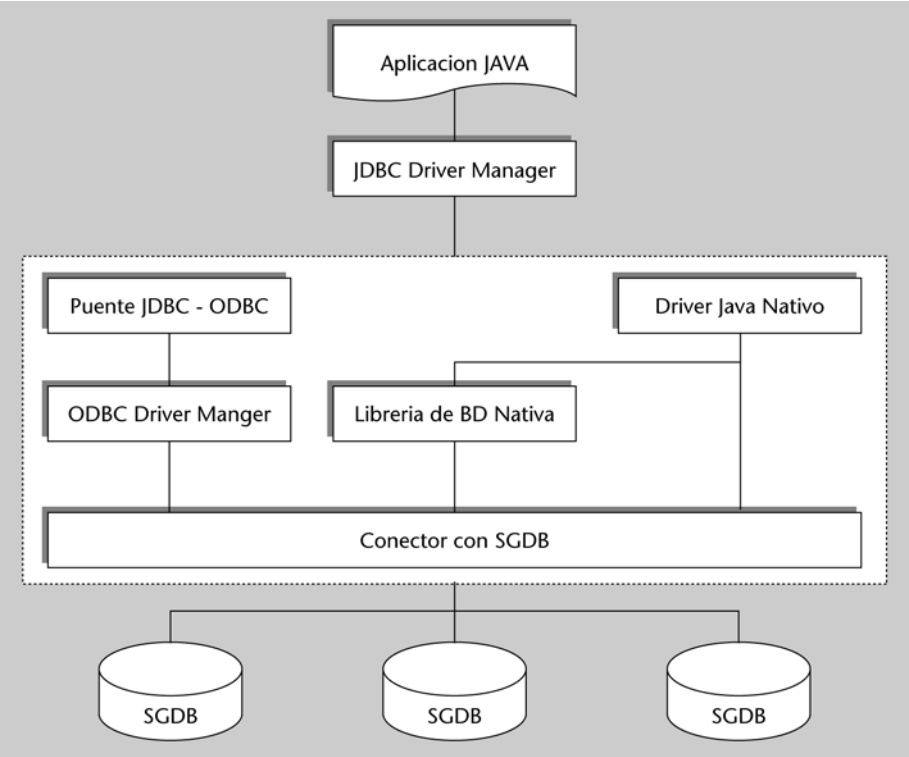
Funcionalidades

Hay funcionalidades más avanzadas de esta librería que aumentan continuamente.
De todas formas, con las presentadas basta para identificar las ventajas de trabajar con una capa de abstracción del motor de base de datos donde se almacenan los datos de nuestra aplicación.

2. Conexión y uso de bases de datos en lenguaje Java

El acceso a bases de datos desde Java se realiza mediante el estándar JDBC (*Java data base conectivity*), que permite un acceso uniforme a las bases de datos independientemente del SGBD. De esta manera, las aplicaciones escritas en Java no necesitan conocer las especificaciones de un SGBD en particular, basta con comprender el funcionamiento de JDBC. Cada SGBD que quiera utilizarse con JDBC debe contar con un adaptador o controlador.

La estructura de JDBC se puede expresar gráficamente como sigue:



Hay *drivers* para la mayoría de SGBD, tanto de software libre como de código abierto. Además, hay *drivers* para trabajar con otros tipos de datos (hojas de cálculo, ficheros de texto, etc.) como si fueran SGBD sobre los que podemos realizar consultas SQL.

Para usar la API JDBC con un SGBD en particular, necesitaremos el *driver* concreto del motor de base de datos, que media entre la tecnología JDBC y la base de datos. Dependiendo de múltiples factores, el *driver* puede estar escrito completamente en Java, o bien haber usado métodos JNI (*Java native interface*) para interactuar con otros lenguajes o sistemas.

La última versión de desarrollo de la API JDBC proporciona también un puente para conectarse a SGBD que dispongan de *drivers* ODBC (*open database*

connectivity). Este estándar es muy común sobre todo en entornos Microsoft y sólo debería usarse si no disponemos del *driver* nativo para nuestro SGBD.

En el caso concreto de MySQL y PostgreSQL, no tendremos ningún problema en encontrar los drivers JDBC:

- MySQL Connector/J: es el *driver* oficial para MySQL y se distribuye bajo licencia GPL. Es un *driver* nativo escrito completamente en Java.

- JDBC para PostgreSQL: es el *driver* oficial para PostgreSQL y se distribuye bajo licencia BSD. Es un *driver* nativo escrito completamente en Java.

Tanto uno como otro, en su distribución en formato binario, consisten en un fichero .jar (*Java archive*) que debemos situar en el CLASSPATH de nuestro programa para poder incluir sus clases.

Java incluye la posibilidad de cargar clases de forma dinámica. Éste es el caso de los controladores de bases de datos: antes de realizar cualquier interacción con las clases de JDBC, es preciso registrar el controlador. Esta tarea se realiza con el siguiente código:

```
String controlador = "com.mysql.jdbc.Driver"
Class.forName(controlador).newInstance();
```

o bien:

```
Class.forName("org.postgresql.Driver");
```

A partir de este momento, JDBC está capacitado para interactuar con MySQL o PostgreSQL.

2.1. Acceder al SGBD con JDBC

La interfaz JDBC está definida en la librería java.sql. Vamos a importar a nuestra aplicación Java todas las clases definidas en ella.

```
import java.sql.*;
```

Puesto que JDBC puede realizar conexiones con múltiples SGDB, la clase `DriverManager` configura los detalles de la interacción con cada uno en particular. Esta clase es la responsable de realizar la conexión, entregando un objeto de la clase `Connection`.

```
String url="jdbc:mysql://localhost/demo";
String usuario="yo"
String contasenia="contraseña"
Connection conexion = DriverManager.getConnection (url,usuario,contrasenia);
```

El destino de la conexión se especifica mediante un URL de JDBC con la sintaxis siguiente:

```
jdbc:<protocolo_sgbd>:<subnombre>
```

La parte `protocolo_sgdb` de la URL especifica el tipo de SGBD con el que se realizará la conexión, la clase `DriverManager` cargará el módulo correspondiente a tal efecto.

El `subnombre` tiene una sintaxis específica para cada SGDB que tanto para MySQL como para PostgreSQL es *//servidor/base_de_datos*.

Las sentencias en JDBC también son objetos que deberemos crear a partir de una conexión:

```
Statement sentenciaSQL = conexion.createStatement();
```

Al ejecutar una sentencia, el SGBD entrega unos resultados que JDBC también representa en forma de objeto, en este caso de la clase `ResultSet`:

La variable *res* contiene el resultado de la ejecución de la sentencia, y proporciona un cursor que permite leer las filas una a una.

```
ResultSet res = sentencia.executeQuery("SELECT * FROM tabla");
```

Para acceder a los datos de cada columna de la hoja de resultados, la clase `ResultSet` dispone de varios métodos según el tipo de la información de la columna:

- getArray()
- getClob()
- getString()
- getAsciiStream()
- getDate()
- getTime()
- getBigDecimal()
- getDouble()
- getTimestamp()
- getBinaryStream()
- getFloat()
- getURL()
- getBlob()
- getInt()
- getBoolean()
- getLong()
- getByte()
- getObject()
- getObject()
- getBytes()
- getBytes()
- getRef()
- getRef()
- getCraracterStream()
- getShort()

```
import java.sql.*;
// Atención, no debe importarse com.mysql.jdbc ya que se carga dinámicamente!!

public static void main(String[] args) {

  // Cargamos el driver JDBC para MySQL
  String controlador = "com.mysql.jdbc.Driver"
  Class.forName(controlador).newInstance();

  // Conectamos con la BD
  String url="jdbc:mysql://localhost/uoc";
  String usuario="yo"
  String contasenia="contraseña"
  Connection conexion = DriverManager.getConnection (url,usuario,contrasenia);

  // Creamos una sentencia SQL
  Statement sentenciaSQL = conexion.createStatement();
  // Ejecutamos la sentencia
  ResultSet res = sentencia.executeQuery("SELECT * FROM tabla");

  // Iteramos sobre la hoja de resultados
  while (res.next()) {
    // Obtenemos el campo 'nombre' en forma de String
    System.out.println(res.getString("nombre") );
  }

  // Finalmente, se liberan los recursos utilizados.
  res.close();
  sentencia.close();
  conexion.close();
}
```

En el ejemplo anterior no se ha previsto nada para tratar con los errores que puedan producirse, porque en Java el tratamiento de errores se realiza a través de `Exceptions`. En JDBC se han previsto excepciones para los errores que pueden producirse a lo largo de todo el uso de la API: conexión, ejecución de la sentencia, etc.

Revisemos el ejemplo, utilizando excepciones para tratar los errores.

```
import java.sql.*;
// Atención, no debe importarse com.mysql.jdbc ya que se carga
// dinámicamente!!

public static void main(String[] args) {

  try {
    // Cargamos el driver JDBC para MySQL
    String controlador = "com.mysql.jdbc.Driver"
    Class.forName(controlador).newInstance();
  } catch (Exception e) {
    System.err.println("No puedo cargar el controlador de MySQL ...");
    e.printStackTrace();
  }

  try {
    // Conectamos con la BD
    String url="jdbc:mysql://localhost/uoc";
    String usuario="yo"
    String contasenia="contraseña"
    Connection conexion = DriverManager.getConnection (url,usuario,contrasenia);
```

```
        // Creamos una sentencia SQL
        Statement sentenciaSQL = conexion.createStatement();
        // Ejecutamos la sentencia
        ResultSet res = sentencia.executeQuery("SELECT * FROM tabla");

        // Iteramos sobre la hoja de resultados
        while (res.next()) {
          // Obtenemos el campo 'nombre' en forma de String
          System.out.println(res.getString("nombre") );
        }

        // Finalmente, se liberan los recursos utilizados.
        res.close();
        sentencia.close();
        conexion.close();
      } catch (SQLException e) {
        System.out.println("Excepción del SQL: " + e.getMessage());
        System.out.println("Estado del SQL: " + e.getSQLState());
        System.out.println("Error del Proveedor: " + e.getErrorCode());
      }
    }
```

Mientras que la operación `executeQuery()` de la clase `Statement` devuelve un objeto `ResultSet`, la operación `executeUpdate`() sólo devuelve su éxito o fracaso. Las sentencias SQL que se utilizan con `executeUpdate`() son **insert**, **update**, o **delete**, porque no devuelven ningún resultado.

Errores

Todos los errores de JDBC se informan a través de `SQLExeption`.

`SQLWarning` presenta las advertencias de acceso a las bases de datos.

```
public void Insertar_persona(String nombre, direccion, telefono){
Statement sentencia = conexion.CreateStatement();
sentencia.executeUpdate( "insert into personas values("
+ nombre + ","
+ domicilio + ","
+ telefono + ")" );
}
```

2.2. Sentencias preparadas

Las sentencias preparadas de JDBC permiten la "precompilación" del código SQL antes de ser ejecutado, permitiendo consultas o actualizaciones más eficientes. En el momento de compilar la sentencia SQL, se analiza cuál es la estrategia adecuada según las tablas, las columnas, los índices y las condiciones de búsqueda implicados. Este proceso, obviamente, consume tiempo de procesador, pero al realizar la compilación una sola vez, se logra mejorar el rendimiento en siguientes consultas iguales con valores diferentes.

Otra ventaja de las sentencias preparadas es que permiten la parametrización: la sentencia SQL se escribe una vez, indicando las posiciones de los datos que van a cambiar y, cada vez que se utilice, le proporcionaremos los argumentos necesarios que serán sustituidos en los lugares correspondientes. Los parámetros se especifican con el carácter '?'.

```
public class Actualizacion{
  private PreparedStatement sentencia;

  public void prepararInsercion(){
    String sql = "insert into personas values ( ?, ? ,? )";
    sentencia = conexion.prepareStatement(sql);
  }

  public void insertarPersona(String nombre, dirección, telefono)
{
    sentencia.setString(1, nombre);
    sentencia.setString(2, direccion);
    sentencia.setString(3, telefono);
    sentencia.executeUpdate();
  }
}
```

Al utilizar esta clase, obviamente, deberemos llamar primero al método que prepara la inserción, y posteriormente llamar tantas veces como sea necesario al método `insertarPersona`.

Se definen tres parámetros en la sentencia SQL, a los cuales se hace referencia mediante números enteros consecutivos:

```
String sql = "insert into personas values ( ?, ? ,? )";
```

Ejemplo

El segundo de los tres parámetros se especifica con `sentencia.setString(2, direccion);`

La clase `PreparedStatement` incluye un conjunto de operaciones de la forma `setXXXX()`, donde `XXXX` es el tipo de dato para los campos de la tabla. Una de esas operaciones es precisamente `setString()` que inserta la variable en un campo de tipo cadena.

2.3. Transacciones

La API JDBC incluye soporte para transacciones, de forma que se pueda deshacer un conjunto de operaciones relacionadas en caso necesario. Este comportamiento es responsabilidad de la clase `Connection`.

Por omisión, cada sentencia se ejecuta en el momento en que se solicita y no se puede deshacer. Podemos cambiar este comportamiento con la operación siguiente:

```
conexion.setAutoCommit(false);
```

Después de esta operación, es necesario llamar a `commit()` para que todas las sentencias SQL pendientes se hagan definitivas:

```
sentencia.executeUpdate(...);
...
sentencia.executeUpdate(...);
...
conexion.commit(); // Se hacen permanentes las dos actualizaciones anteriores
```

En caso contrario, desharemos todas las actualizaciones después del último `commit()`:

```
sentencia.executeUpdate(...);
...
sentencia.executeUpdate(...);
...
sentencia.executeUpdate(...);
...
conexion.rollback(); // Cancela las tres últimas actualizaciones
```

Resumen

Hemos presentado algunas de las formas más habituales de conectarse a los SGBD que hemos visto en módulos anteriores desde PHP y Java.

Hemos podido comprobar que no existen demasiadas variaciones ni restricciones entre MySQL y PostgreSQL en cuanto a su acceso desde lenguajes de programación, sino al contrario, los esfuerzos se encaminan en homogeneizar el desarrollo e independizarlo del SGBD con el que trabajamos.

En PHP, hemos repasado los métodos nativos y visto sus particularidades. Hemos comprobado que, a no ser que necesitemos características propias y muy avanzadas de un SGBD, no es aconsejable usar esos métodos por los problemas que nos puede ocasionar un cambio de gestor de base de datos en el futuro. Aun así, es interesante revisarlos porque encontraremos muchas aplicaciones de software libre desarrolladas en PHP que los utilizan.

PEAR::DB es un ejemplo de librería de abstracción (no es la única) bien hecha y con el soporte de la fundación que mantiene PHP. Tiene todo lo que podemos desear y actualmente es completamente estable y usable en entornos empresariales.

En Java, hemos visto JDBC. Aunque la API da mucho más de sí, creemos que hemos cumplido los objetivos de este capítulo, sin entrar en conceptos que sólo programadores expertos en Java podrían apreciar.

Así pues, se han proporcionado los elementos de referencia y los ejemplos necesarios para trabajar con bases de datos en nuestras aplicaciones.

Bibliografía

Documentación de PHP: http://www.php.net/docs.php

PEAR :: The PHP Extension and Application Repository: http://pear.php.net/

Pear::DB Database Abstraction Layer: http://pear.php.net/package/DB

DBC Technology: http://java.sun.com/products/jdbc/

MySQL Connector/J: http://dev.mysql.com/downloads/connector/j

PostgreSQL JDBC Driver: http://jdbc.postgresql.org/

Caso de estudio

Marc Gibert Ginesta

UOC
Formación de Posgrado

Índice

Introducción

Este módulo forma parte del curso "Bases de datos" del itinerario "Administrador web y comercio electrónico" dentro del Máster Internacional de Software Libre de la Universitat Oberta de Catalunya.

El módulo está estructurado en apartados que corresponden al resto de los módulos de la asignatura, de modo que el estudiante puede ir siguiendo este caso de estudio a medida que va progresando en el curso.

Aunque algunos de los módulos ya disponen de ejercicios de autoevaluación, el caso de estudio presenta una visión completa de un proyecto de bases de datos y proporciona una visión práctica de cada uno de ellos.

Objetivos

Los objetivos que deberíais alcanzar al finalizar el trabajo con la presente unidad son los siguientes:

- Comprender desde un punto de vista práctico los conceptos explicados en las unidades didácticas teóricas.

- Disponer de un modelo de referencia para emprender proyectos de bases de datos.

- Adquirir el criterio suficiente para identificar las actividades clave y tomar decisiones en un proyecto que implique el uso de bases de datos.

1. Presentación del caso de estudio

Acabamos de entrar a trabajar en una pequeña empresa –SuOrdenadorAMedida, S.L.– dedicada a la venta de ordenadores a particulares y otras empresas. Cuando nos hicieron la entrevista de trabajo, comentamos nuestra pasión por el software libre y en concreto hicimos hincapié en nuestro conocimiento de los motores de bases de datos libres y las ventajas que podían aportar respecto de los gestores propietarios. No sabemos si eso fue lo que convenció a la persona de recursos humanos o no, pero, en todo caso, y visto el resultado de la reunión que hemos tenido en nuestro primer día de trabajo, vamos a tener que aplicar nuestros conocimientos a fondo.

Nos han contado que, hasta ahora, la gestión de la empresa se llevaba a cabo con programas propietarios de gestión y contabilidad, pero que debido a problemas con la empresa que desarrollaba estos programas, se está considerando la migración de la gestión administrativa y de operaciones a entornos abiertos. Para acabar de decidirse, nos proponen que empecemos por renovar el sistema de gestión de peticiones e incidencias por parte de los clientes, de modo que esté basada en software libre.

Actualmente, las peticiones e incidencias se reciben telefónicamente, por correo electrónico o en persona en alguno de los locales que tiene la empresa. La persona que atiende al teléfono o lee los correos electrónicos de plantea una serie de preguntas al cliente y escribe en una plantilla de documento las respuestas. A continuación, se imprime el documento y se deja en una bandeja que recogen los técnicos cada mañana.

A medida que los técnicos van avanzando en la solución de la incidencia (o han llamado al cliente para pedir más datos), van apuntando las acciones y el estado del problema en la hoja que recogieron, hasta que la incidencia queda resuelta. En ese momento, la dejan en una bandeja que recoge cada mañana el personal de administración, que se pone en contacto con el cliente y factura el importe correspondiente a las horas de trabajo y componentes sustituidos.

Es obvio que este sistema presenta numerosas deficiencias, y que el rendimiento tanto de los técnicos, como del personal administrativo y de atención al cliente podría aumentar enormemente si muchos de estos procesos fueran automáticos, centralizados y, a poder ser, conectados con el resto del sistema de información de la empresa.

Éste es, a grandes rasgos, el problema que se nos plantea, y que utilizaremos como caso de estudio para aplicar los conocimientos adquiridos durante el desarrollo del curso.

2. El modelo relacional y el álgebra relacional

Visto el proyecto planteado, decidimos hacer las cosas bien hechas para, de paso, impresionar a nuestro jefe con nuestros conocimientos en bases de datos. El primer paso será presentarle un documento que describa el modelo relacional que vamos a utilizar, en el que incluiremos algunas consultas de muestra para que pueda comprobar qué será capaz de hacer con nuestro proyecto cuando esté acabado.

2.1. Determinar las relaciones

En primer lugar determinaremos las relaciones, sus atributos y los dominios de cada uno de ellos:

```
PETICION(referencia, cliente, resumen, estado, fecharecepcion, fechainicio, fechafin,
tiempoempleado)
NOTA_PETICION(peticion, nota, fecha, empleado)
MATERIAL_PETICION(nombrematerial, peticion, cantidad, precio)
CLIENTE(nombre, nif, telefono, email)
EMPLEADO(nombre, nif)
```

En la relación `PETICION`, hemos decidido que convendría tener una referencia interna de la petición, que nos ayudará al hablar de ella con el cliente (si tuviese varias abiertas) y evitará confusiones al trabajar. El resto de atributos son bastante explícitos.

Como una petición puede evolucionar con el tiempo, a medida que se piden más datos al cliente, la incidencia va evolucionando, etc., hemos creado las relaciones `NOTA_PETICION` y `MATERIAL_PETICION` para reflejarlo.

También hemos tenido que definir las relaciones `CLIENTE` y `EMPLEADO` para poder relacionarlas con las peticiones y las notas que se vayan generando durante su resolución.

A continuación vamos a definir los dominios de los atributos:

```
PETICION:
dominio(referencia)=números
dominio(cliente)=NIF
```

```
dominio(resumen)=texto
dominio(estado)=estados
dominio(fecharecepcion)=fechayhora
dominio(fechainicio)=fechayhora
dominio(fechafin)=fechayhora
dominio(tiempoempleado)=horasyminutos

NOTA_PETICION:
dominio(peticion)=números
dominio(nota)=texto
dominio(fecha)=fechayhora
dominio(empleado)=NIF

MATERIAL_PETICION:
dominio(nombrematerial)=nombreMaterial
dominio(peticion)=números
dominio(precio)=precio
dominio(cantidad)=números

CLIENTE:
dominio(nombre)=nombreCliente
dominio(nif)=NIF
dominio(telefono)=teléfonos
dominio(email)=emails

EMPLEADO:
dominio(nombre)=nombreEmpleado
dominio(nif)=NIF
```

Al definir los dominios de cada atributo, ya nos hemos avanzado en la toma de algunas decisiones: al decidir, por ejemplo, que el dominio del atributo `empleado` en la relación `NOTA_PETICION` es `NIF`, estamos implícitamente determinando que la clave primaria de la relación `EMPLEADO` será del dominio NIF y que usaremos un atributo de este dominio para referirnos a él.

Este proceso descrito indicando directamente su resultado, normalmente es fruto de una revisión de las entidades a medida que se van definiendo y analizando las necesidades de éstas.

Nota

La regla de integridad del modelo correspondiente a la clave primaria obligará a que no existan dos notas sobre la misma petición hechas en la misma fecha y hora por parte del mismo empleado, lo cual es perfectamente lícito y coherente.

2.2. Definición de claves

Aunque algunas claves ya se intuyen a partir de los atributos de las relaciones, vamos a determinarlas para completar el caso.

```
PETICION:
Claves candidatas: {referencia}
Clave primaria: {referencia}

NOTA_PETICION:
Claves candidatas: {peticion,fecha,empleado}
Clave primaria: {peticion, fecha,empleado}

MATERIAL_PETICION:
Claves candidatas: {nombrematerial,peticion}
Clave primaria: {nombrematerial,peticion}

CLIENTE:
Claves candidatas: {nif}
Clave primaria: {nif}

EMPLEADO:
Claves candidatas: {nif}
Clave primaria: {nif}
```

En todos los casos sólo tenemos una clave candidata y, por lo tanto, no caben dudas a la hora de escoger la clave primaria. Esto no tiene por que ser así: en la relación `EMPLEADO`, podríamos haber incluido más atributos (número de la seguridad social, un número de empleado interno, etc.) que serían claves candidatas susceptibles de ser clave primaria.

Ahora podemos reescribir las relaciones:

```
PETICION(referencia, cliente, resumen, estado, fecharecepcion, fechainicio, fechafin,
tiempoempleado)
NOTA_PETICION(peticion, nota, fecha, empleado)
MATERIAL_PETICION(nombrematerial, peticion, cantidad, precio)
CLIENTE(nombre, nif, telefono, email)
EMPLEADO(nombre, nif)
```

Las claves foráneas ya se intuyen a partir de las relaciones, aunque vamos a comentarlas para completar el caso:

`NOTA_PETICION:`

Tiene de clave foránea el atributo `{peticion}`, que establece la relación (y pertenece al mismo dominio) con el atributo `{referencia}` de la relación `PETICION`.

También tiene la clave foránea `{empleado}`, que establece la relación con `EMPLEADO` a partir de su clave primaria `{nif}`.

`MATERIAL_PETICION:`

Tiene de clave foránea el atributo `{peticion}`, que establece la relación (y pertenece al mismo dominio) con el atributo `{referencia}` de la relación `PETICION`.

2.3. Reglas de integridad

En este punto, no es necesario preocuparse por las reglas de integridad del modelo que tratan sobre la clave primaria, ya que nos vendrán impuestas en el momento de crear las tablas en el SGBD.

Es conveniente, no obstante, fijar las decisiones sobre la integridad referencial; en concreto, qué vamos a hacer en caso de restricción. Así pues, para cada relación que tiene una clave primaria referenciada desde otra, deberemos decidir qué política cabe aplicar en caso de modificación o borrado:

PETICION

- Modificación del atributo `{referencia}` referenciado desde `NOTA_PETICION` y `MATERIAL_PETICION`: aquí podemos optar por la restricción o por la actualización en cascada (más cómoda, aunque no todos los SGBD la implementan, como veremos más adelante).

- Borrado del atributo `{referencia}`. Aquí optaremos por una política de restricción. Si la petición tiene notas asociadas o materiales, significa que ha habido alguna actividad y, por lo tanto, no deberíamos poder borrarla. Si se desea anularla, ya estableceremos un estado de la misma que lo indique.

CLIENTE

- Modificación del atributo `{nif}` referenciado desde `PETICION`. Es probable que si un cliente cambia de NIF (por un cambio del tipo de sociedad, etc.) deseemos mantener sus peticiones. Aquí la política debe ser de actualización en cascada.

- Borrado del atributo `{nif}`. Es posible que si queremos borrar un cliente, es porque hemos terminado toda relación con él y, por lo tanto, es coherente utilizar aquí la política de anulación.

EMPLEADO

- Modificación del atributo `{nif}` referenciado desde `NOTA_PETICION`. No es probable que un empleado cambie su NIF, salvo caso de error. Aun así, en caso de que se produzca, es preferible la actualización en cascada.

- Borrado del atributo `{nif}`. Aunque eliminemos un empleado si termina su relación con la empresa, no deberíamos eliminar sus notas. La mejor opción es la restricción.

2.4. Álgebra relacional

Para probar el modelo, una buena opción es intentar realizar algunas consultas sobre él y ver si obtenemos los resultados deseados. En nuestro caso, vamos a realizar las siguientes consultas:

- Obtención de una petición junto con los datos del cliente:

```
R:= PETICION [cliente=nif] CLIENTE
```

- Obtención de una petición con todas sus notas:

```
NP(peticionnota, nota, fechanota, empleado):=NOTA_PETICION(peticion, nota, fecha, empleado)
R:=PETICION[peticion=peticionnota]NOTA_PETICION
```

- Obtención de los datos de todos los empleados que han participado en la petición 5:

```
NP:=NOTA_PETICION[peticion=5]
RA:=EMPLEADO[nif=empleado]NP
R:=RA[nombre,nif]
```

Ejercicio

Os sugerimos que intentéis más operaciones sobre el modelo para familiarizaros con él.

3. El lenguaje SQL

Una vez terminado el modelo relacional, decidimos completar la documentación que veníamos realizando con las sentencias SQL correspondientes. Así, veremos en qué se concretará el modelo relacional.

Como aún no sabemos en qué sistema gestor de base de datos vamos a implantar la solución, decidimos simplemente anotar las sentencias según el estándar SQL92, y, posteriormente, ya examinaremos las particularidades del sistema gestor escogido para adaptarlas.

3.1. Sentencias de definición

- Creación de la base de datos

```
CREATE SCHEMA GESTION_PETICIONES;
```

Atención

En algunos casos es conveniente la definición de dominios para facilitar el trabajo posterior de mantenimiento de la coherencia de la base de datos. No es aconsejable definir dominios para cada dominio relacional, pero sí en los casos en que una columna puede tomar una serie de valores determinados.

- Definición de dominios

```
CREATE DOMAIN dom_estados AS CHAR (20)
   CONSTRAINT estados_validos
   CHECK (VALUE IN ('Nueva','Se necesitan más datos','Aceptada','Confirmada',
'Resuelta',
'Cerrada'))
   DEFAULT 'Nueva';
```

- Creación de las tablas

```
CREATE TABLE PETICION ( referencia INTEGER NOT NULL,
   cliente INTEGER NOT NULL,
   resumen CHARACTER VARYING (2048),
   estado dom_estados NOT NULL,
   fecharecepcion TIMESTAMP NOT NULL,
   fechainicio TIMESTAMP, fechafin TIMESTAMP,
   tiempoempleado TIME, PRIMARY KEY (referencia),
   FOREIGN KEY cliente REFERENCES CLIENTE(nif)
      ON DELETE CASCADE
      ON UPDATE CASCADE,
   CHECK (fecharecepcion < fechainicio),
   CHECK (fechainicio < fechafin) );
```

Atención

Aquí deberemos tener en cuenta las reglas de integridad, ya que habrá que explicitar la política escogida como restricción.

```
CREATE TABLE NOTA_PETICION ( peticion INTEGER NOT NULL,
   nota CHARACTER VARYING (64000),
   fecha TIMESTAMP NOT NULL,
   empleado CHARACTER (9),
   FOREIGN KEY (peticion) REFERENCES PETICION(referencia)
      ON DELETE NO ACTION
      ON UPDATE CASCADE,
   FOREIGN KEY (empleado) REFERENCES EMPLEADO(nif)
      ON DELETE NO ACTION
      ON UPDATE CASCADE );

CREATE TABLE MATERIAL_PETICION ( nombrematerial CHARACTER
VARYING (100) NOT NULL,
   peticion INTEGER NOT NULL,
   precio DECIMAL(8,2),
   cantidad INTEGER,
   FOREIGN KEY (peticion) REFERENCES PETICION(referencia)
      ON DELETE NO ACTION
      ON UPDATE CASCADE );

CREATE TABLE CLIENTE ( nombre CHARACTER VARYING (100) NOT
NULL,
   nif CHARACTER (9) NOT NULL,
   telefono CHARACTER (15),
   email CHARACTER (50),
   PRIMARY KEY (nif) );

CREATE TABLE EMPLEADO (nombre CHARACTER VARYING (100) NOT
NULL,
   nif CHARACTER (9) NOT NULL,
   PRIMARY KEY (nif) );
```

- Creación de vistas

Función de vistas

Las vistas agilizarán las consultas que prevemos que van a ser mas frecuentes.

– Peticiones pendientes:

```
CREATE VIEW peticiones_pendientes (referencia, nombre_cliente, resumen, estado, duracion,
fecharecepcion) AS (
SELECT P.referencia, C.nombre, P.resumen, P.estado, (P.fechainicio     P.fecharecepcion),
P.fecharecepcion
FROM PETICION P JOIN CLIENTE C ON P.cliente = C.nif
WHERE estado NOT IN ('Resuelta','Cerrada') ORDER BY fecharecepcion )
```

– Tiempo y precio de los materiales empleados para las peticiones terminadas en el mes en curso:

```
CREATE VIEW peticiones_terminadas (referencia, nombre_cliente, resumen,
tiempo_empleado, importe_materiales) AS (
     SELECT P.referencia, C.nombre, P.resumen, P.tiempoempleado, SUM(M.precio)
     FROM PETICION P, CLIENTE C, MATERIAL_PETICION M
     WHERE P.cliente=C.nif AND M.peticion=P.referencia AND estado='Resuelta'
     GROUP BY P.referencia)
```

3.2. Sentencias de manipulación

A continuación, decidimos indicar algunas sentencias de manipulación corrientes para completar la documentación. De esta manera, cuando empecemos el desarrollo, tendremos mucho más claras estas operaciones sobre la base de datos:

- Nuevo cliente:

```
INSERT INTO CLIENTE VALUES ('Juan Pérez','42389338A','912223354','juanperez@gmail.com');
```

- Nueva petición:

```
INSERT INTO PETICION VALUES (5,'42389338A', 'No le arranca el ordenador',
'Nueva',CURRENT_TIMESTAMP,NULL,NULL,NULL);
```

- Cambio de estado de la petición, añadimos una nota y un material:

```
UPDATE PETICION SET estado='Aceptada' WHERE referencia=5;
INSERT INTO NOTA_PETICION VALUES (5,'Parece un problema del disco duro. Vamos examinarlo
más a fondo.',CURRENT_TIMESTAMP,'35485411G');

INSERT INTO MATERIAL_PETICION VALUES ('Disco duro 20Gb',5,250.00,1);
```

- Materiales solicitados en la petición 5:

```
SELECT nombrematerial, cantidad, precio FROM MATERIAL_PETICION WHERE peticion=5
```

- Número de peticiones abiertas del cliente '42389338A':

```
SELECT COUNT(*) FROM PETICION WHERE cliente='42389338A' AND estado NOT IN ('Resuelta',
'Cerrada');
```

La creación de vistas del apartado anterior nos ha mostrado también algunas consultas complejas que repetimos a continuación:

- Peticiones abiertas:

```
SELECT P.referencia, C.nombre, P.resumen, P.estado, (P.fechainicio   P.fecharecepcion),
P.fecharecepcion FROM PETICION P JOIN CLIENTE C ON P.cliente = C.nif WHERE estado NOT IN
('Resuelta','Cerrada') ORDER BY fecharecepcion;
```

- Tiempo y precio de los materiales empleados para las peticiones terminadas en el mes en curso:

```
SELECT P.referencia, C.nombre, P.resumen, P.tiempoempleado, SUM(M.precio) FROM PETICION
P, CLIENTE C, MATERIAL_PETICION M WHERE P.cliente=C.nif AND M.peticion=P.referencia AND
estado='Resuelta' GROUP BY P.referencia;
```

Finalmente, vamos a practicar con las consultas que realizamos en álgebra relacional en el apartado anterior:

- Obtención de una petición junto con los datos del cliente:

```
R:= PETICION [cliente=nif] CLIENTE
SELECT * FROM PETICION JOIN CLIENTE ON PETICION.cliente=CLIENTE.nif;
```

- Obtención de una petición con todas sus notas:

```
NP(peticionnota,nota,fechanota,empleado):=NOTA_PETICION (peticion,nota,fecha,empleado)
R:=PETICION[referencia=peticionnota]NP
SELECT PETICION.*, peticion AS peticionnota, nota, fecha as fechanota, empleado FROM
PETICION JOIN NOTA_PETICION ON referencia=peticionnota;
```

- Obtención de los datos de todos los empleados que han participado en la petición 5:

```
NP:=NOTA_PETICION[peticion=5]
RA:=EMPLEADO[nif=empleado]NP
R:=RA[nombre,nif]
SELECT E.nombre, E.nif FROM EMPLEADO E, NOTA_PETICION N WHERE E.nif=NOTA_PETICION.empleado
AND NOTA_PETICION.peticion=5;
```

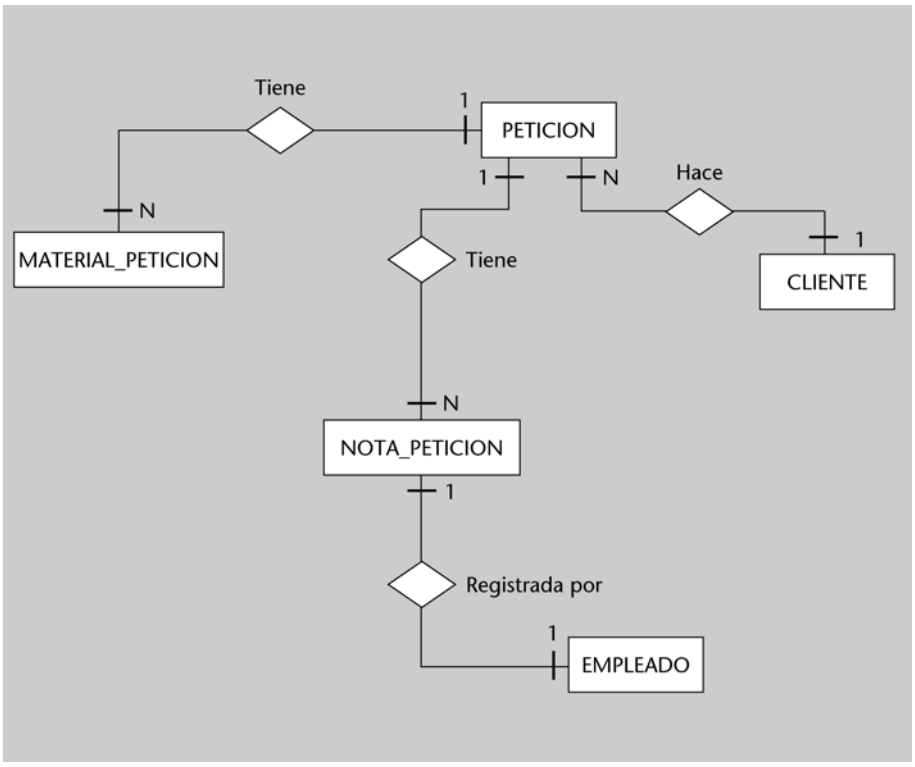
4. Introducción al diseño de bases de datos

Aunque las sentencias SQL de creación de tablas son bastante claras para un usuario técnico, de cara a la reunión previa a la toma de decisión sobre el SGBD concreto en el que vamos a implantar la solución, necesitaremos algo más.

Teniendo en cuenta que entre los asistentes a la reunión no hay más técnicos especializados en bases de datos que nosotros, hemos pensado que disponer un modelo entidad-relación del sistema nos ayudará a comunicar mejor la estructura que estamos planteando y, de paso, a demostrar (o, si es necesario, corregir) que el modelo relacional que planteamos al inicio es el correcto.

4.1. Diseño conceptual: el modelo ER

Vamos a plantear en primer lugar el modelo obtenido y, después, comentaremos los aspectos más interesantes:



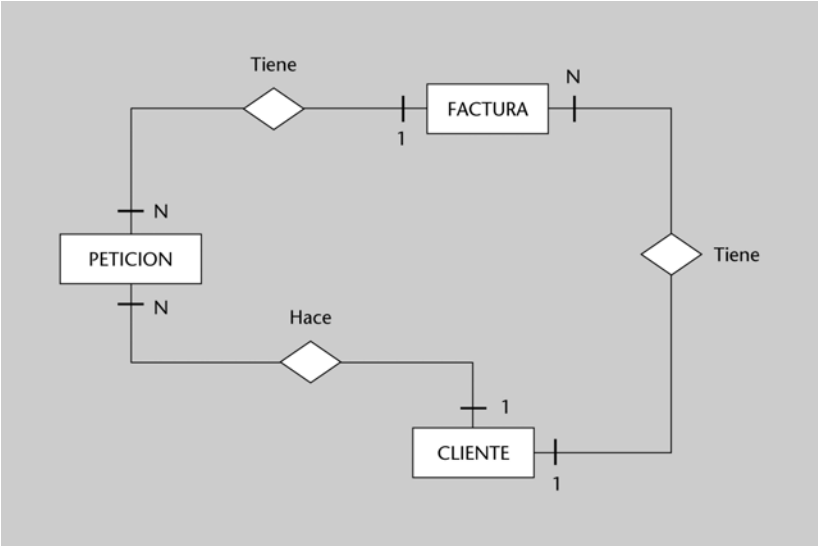
El modelo, expresado de este modo, es mucho más comprensible por parte de personal no técnico o no especializado en tecnologías de bases de datos.

A partir de la expresión gráfica del modelo, identificamos limitaciones o puntos de mejora, que anotamos a continuación:

- Probablemente, debemos incluir información de facturación a clientes por las peticiones realizadas.
- Probablemente, habrá peticiones que puedan agruparse en una entidad superior (un proyecto o trabajo), o bien peticiones relacionadas entre ellas (peticiones que deban resolverse antes que otras).

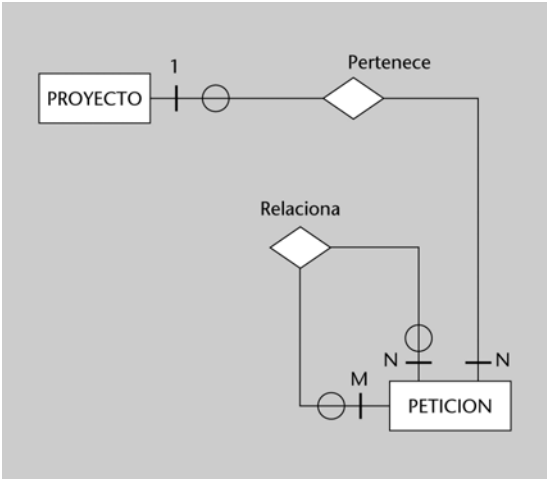
Una vez identificadas estas limitaciones, vamos a ampliar el modelo para corregirlas e impresionar a nuestros superiores de cara a la reunión.

- Información de facturación a clientes:



Lo único que hemos introducido ha sido la entidad `FACTURA`, que se relaciona con *N* peticiones y con un único cliente. Un `CLIENTE` puede tener varias facturas asociadas, pero una petición sólo puede pertenecer a una única factura.

- Grupos de peticiones y relaciones entre ellas.



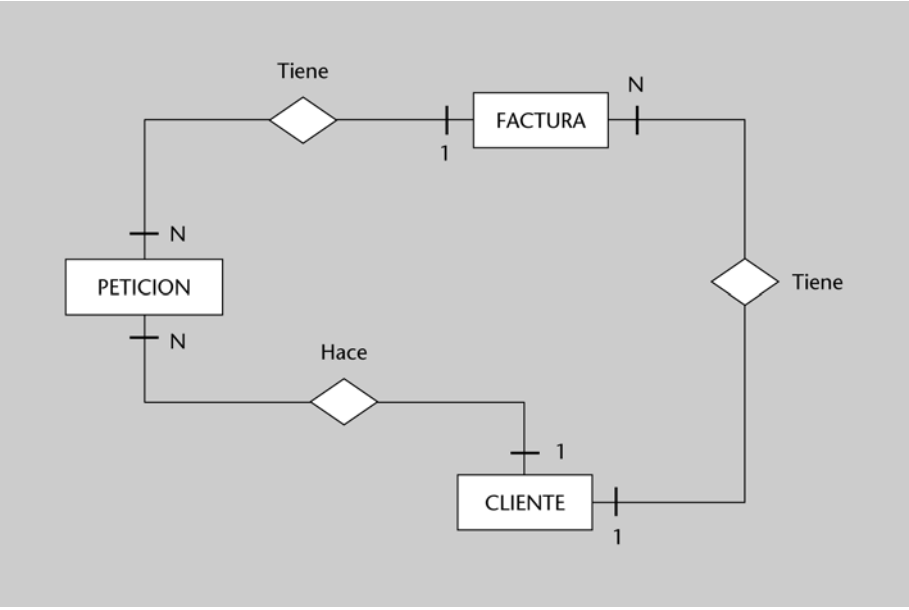
La pertenencia a un proyecto será opcional, y así lo indicamos en el diagrama. Por lo que respecta a las relaciones de peticiones entre ellas, se trata de una interrelación recursiva. Si queremos contemplar casos como los siguientes, debemos expresar la relación como M:N recursiva y opcional. En la interrelación `RELACIONA`, debemos contemplar algún atributo que indique de qué tipo de relación se trata en cada caso:

- Una petición depende de una o más peticiones.
- Una petición bloquea a una o más peticiones.
- Una petición es la duplicada de una o más peticiones.
- Una petición está relacionada con una o más peticiones.

4.2. Diseño lógico: la transformación del modelo ER al modelo relacional

En el apartado anterior sugerimos unas ampliaciones sobre el modelo ER que proporcionaban más prestaciones al proyecto. A continuación, vamos a realizar la transformación al modelo relacional de estas ampliaciones:

- Información de facturación a clientes.



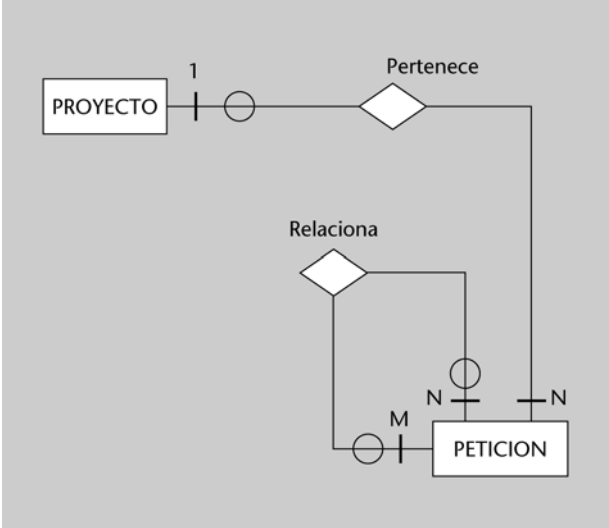
Según las transformaciones vistas en el módulo "El lenguaje SQL", la entidad `FACTURA` se transforma en la relación `FACTURA`, con los siguientes atributos:

```
FACTURA(numfactura, fecha, cliente)
```

Donde `cliente` es una clave foránea que corresponde a la interrelación `TIENE` entre `CLIENTE` y `FACTURA`. Un cliente puede tener *N* facturas, pero una factura pertenece sólo a un único cliente.

La interrelación entre FACTURA y PETICION del tipo 1:N se transforma también en una nueva clave foránea, que aparece siempre en el lado *N* de la interrelación; o sea, en la relación PETICION. Si existen peticiones que no deban facturarse (porque se han cerrado sin resolverse, o eran duplicadas de otras, etc.), su clave foránea tomaría el valor NULO.

- Grupos de peticiones y relaciones entre ellas.



Por una parte, la entidad proyecto debe transformarse en la relación PROYECTO, con atributos como los siguientes:

```
PROYECTO(codigo, nombre, fechainicio, fechafin)
```

La relación 1:N entre PROYECTO y PETICIÓN se transformará en la inserción de una nueva clave foránea en la relación PETICION, que podrá tener valor NULO si la petición no pertenece a ningún proyecto; es decir, si se trata de una petición aislada. La relación PETICION quedaría así:

```
PETICION(referencia, cliente, resumen, estado, fecharecepcion, fechainicio, fechafin,
tiempoempleado, factura, proyecto)
```

Por lo que respecta a las relaciones entre peticiones, se trata de una interrelación recursiva N:M, y por lo tanto se transformará en una nueva relación, PETICION_RELACION:

```
PETICION_RELACION(referencia_peticion1, referencia_peticion2, tiporelacion)
```

En este caso, y según el valor que pueda tomar el atributo `{tiporelacion}`, tendrá importancia o no qué referencia de petición aparece en cada atributo de la relación.

Ejemplo

Si la relación debe indicar simplemente que dos peticiones están relacionadas, entonces no importa qué referencia sea, la 1 o la 2.

En cambio, si el atributo `{tiporelacion}` indica un bloqueo o una dependencia entre relaciones (porque una debe resolverse antes que otra, por ejemplo), entonces sí tiene sentido qué referencia de petición se almacena en el atributo 1 y cuál en el 2. En todo caso, esta tarea corresponderá a la solución que se adapte y trabaje con la base de datos en último término, no al propio modelo.

5. Bases de datos en MySQL

Una vez hemos terminado el proceso de diseño de nuestra solución, en cuanto a su sistema de información, es hora de implantarlo sobre un sistema gestor de bases de datos.

Ya que disponemos de dos alternativas (MySQL y PostgreSQL), y no nos corresponde tomar la decisión final (sólo hacer la recomendación), vamos a elaborar una lista con los aspectos clave en la toma de decisiones y a puntuar, o comentar, cada SGBD según los ítems siguientes:

- Modelo de licencia, precio.
- Soporte por parte del fabricante.
- Conexión desde PHP.
- Prestaciones en creación de las estructuras (tablas, índices, etc.).
- Prestaciones en tipos de datos.
- Prestaciones en consultas simples.
- Prestaciones en consultas complejas.
- Prestaciones en manipulación de datos.
- Facilidad en la administración de usuarios.
- Facilidad en la gestión de copias de seguridad.

Nota

Mostraremos esta lista en forma de tabla, y al final elaboraremos unas conclusiones. Aunque no realicemos una ponderación de cada aspecto de la lista anterior, la simple comparación nos servirá para llegar a una conclusión rápida.

Concepto	Valoración	Comentarios
Modelo de licencia, precio	2	Aunque no nos planteamos vender nuestra solución, ni comercializarla con una licencia propietaria, la licencia dual de MySQL siempre será un aspecto que tendremos que tener en cuenta si alguien se interesa por nuestra aplicación.
Soporte por parte del fabricante	3	Tenemos tanto la opción de contratar soporte en varias modalidades, como la de optar por consultar a la amplísima gama de usuarios del producto. En todo caso, en ambas situaciones obtendremos un excelente soporte.
Conexión desde PHP	3	PHP siempre ha incluido soporte para este SGBD bien con funciones especiales dedicadas que aprovechan al máximo sus características, o bien con librerías PEAR como DB que nos abstraen del SGBD y que soportan MySQL a la perfección. Aunque recientemente ha habido algún problema con la licencia y parecía que PHP no incluiría soporte para MySQL en sus últimas versiones, MySQL ha hecho una excepción con PHP (que sin duda ha contribuido mucho a la popularización de MySQL) por el bien de la comunidad y de sus usuarios.
Prestaciones en creación de las estructuras (tablas, índices, etc.)	2	MySQL es francamente fácil de manejar en este aspecto, y aunque no ofrece todas las prestaciones contempladas en el estándar, es "satisfactorio" para la aplicación que estamos planeando.
Prestaciones en tipos de datos	2	Los tipos de datos soportados por MySQL así como los operadores incluidos en el SGBD son más que suficientes para nuestra aplicación.
Prestaciones en consultas simples	3	Ésa es precisamente la característica que hace que MySQL sea uno de los SGBD mejor posicionados.

Clave de valoración

- 1: no satisfactorio
- 2: satisfactorio
- 3: muy satisfactorio

Concepto	Valoración	Comentarios
Prestaciones en consultas complejas	2	Hasta hace poco, MySQL no soportaba subconsultas y esto implicaba un mayor esfuerzo por parte de los programadores. Ahora ya las soporta y a un nivel igual al de sus competidores.
Prestaciones en manipulación de datos	3	MySQL incluye multitud de opciones no estándares para cargar datos externos, insertar o actualizar sobre la base de consultas complejas y la utilización de operadores como condiciones para la manipulación.
Facilidad en la administración de usuarios	3	Soporta muy bien el estándar en cuanto a la creación de usuarios y la gestión de sus privilegios con GRANT y REVOKE. Además, todos estos datos están accesibles en tablas de sistema, lo que hace muy sencilla la verificación de permisos.
Facilidad en la gestión de copias de seguridad	3	Disponemos tanto de herramientas de volcado, como la posibilidad de copia binaria de la base de datos. Además, dada su popularidad varios fabricantes de soluciones de copias de seguridad proporcionan conectores para realizar *backups* de la base de datos en caliente.
Conclusión	2,6	Estamos ante un "más que satisfactorio" SGBD para la solución que nos planteamos. No hay ninguna carencia insalvable.

6. Bases de datos en PostgreSQL

Concepto	Valoración	Comentarios
Modelo de licencia, precio	3	La licencia BSD no nos limita en ningún aspecto. Simplemente tendremos que incluir la nota sobre la misma en nuestro software, tanto si lo queremos comercializar como si no.
Soporte por parte del fabricante	2	PostgreSQL no ofrece soporte directamente, aunque sí que proporciona los mecanismos para que la comunidad lo ofrezca (listas de correo, IRC, enlaces, etc.). También tiene una lista (corta) de empresas que ofrecen soporte profesional de PostgreSQL. En España sólo hay una, y no es una empresa de desarrollo de software.
Conexión desde PHP	3	PHP siempre ha incluido soporte para este SGBD, bien con funciones especiales dedicadas que aprovechan al máximo sus características o bien con librerías PEAR como DB, que nos abstraen del SGBD y que soportan PostgreSQL a la perfección.
Prestaciones en creación de las estructuras (tablas, índices, etc.)	3	PostgreSQL es muy potente en este aspecto, ofrece prácticamente todas las prestaciones contempladas en el estándar, y tiene un fantástico sistema de extensión.
Prestaciones en tipos de datos	3	Los tipos de datos soportados por PostgreSQL, así como los operadores incluidos en el SGBD son más que suficientes para nuestra aplicación. Además, su sistema de extensiones y definición de tipos y dominios incluye casi todo lo que podamos necesitar.
Prestaciones en consultas simples	3	Por supuesto, PostgreSQL no decepciona en este punto.
Prestaciones en consultas complejas	3	PostgreSQL ha soportado subconsultas, vistas y todo lo que podamos necesitar en nuestra aplicación desde hace varios años. Su implementación de éstas es ya muy estable.
Prestaciones en manipulación de datos	3	PostgreSQL incluye multitud de opciones no estándares para cargar datos externos, insertar o actualizar sobre la base de consultas complejas y la utilización de operadores como condiciones para la manipulación.
Facilidad en la administración de usuarios	2	Soporta bastante bien el estándar en cuanto a la creación de usuarios y la gestión de sus privilegios con GRANT y REVOKE. Su sistema múltiple de autenticación lo hace demasiado complejo en este aspecto.
Facilidad en la gestión de copias de seguridad	3	Disponemos tanto de herramientas de volcado, como de la posibilidad de copia binaria de la base de datos.
Conclusión	2,8	Estamos ante el SGBD casi ideal. Sólo le falta facilitar la gestión de usuarios y mejorar su soporte.

Clave de valoración

- 1: no satisfactorio
- 2: satisfactorio
- 3: muy satisfactorio

7. Desarrollo de aplicaciones en conexión con bases de datos

En la reunión mantenida con la dirección se examinaron muy cuidadosamente los análisis de los SGBD seleccionados. Al ser la diferencia de valoración tan leve, no fue fácil tomar una decisión, pero al final se decidió la implementación de la solución sobre el SGBD PostgreSQL.

Se decidió, también, hacer la implementación en PHP, abstrayéndonos del SGBD con el que trabajáramos. Así, en caso de que la mayor dificultad en la administración de PostgreSQL nos hiciera rectificar la decisión en el futuro, el tiempo de puesta en marcha del cambio sería mínimo.

Antes de iniciar la implantación, vamos a realizar unas pruebas conceptuales de la propia implementación que, después, pasaremos a un equipo de desarrollo interno para que haga el resto. En concreto, tomaremos algunas de las consultas vistas en el capítulo 3 y programaremos los *scripts* PHP de las páginas correspondientes, documentándolas al máximo para facilitar el trabajo al equipo de desarrollo.

En primer lugar, crearemos un fichero .php con la conexión a la base de datos, para incluirlo en todos los PHP que lo vayan a necesitar, y evitar, así, tener que repetir código cada vez. Esta acción también ayudará a mantener centralizados los datos de la conexión y, en caso de que debiéramos cambiar el usuario o la contraseña o cualquier otro dato de la conexión, sólo tendríamos que actualizar dicho fichero.

datosconexion.php

```
<?php
// Incluimos la librería una vez instalada mediante PEAR
require_once 'DB.php';

// Creamos la conexión a la base de datos, en este caso PostgreSQL
$db =& DB::connect('pgsql://usuario:password@servidor/basededatos');

// Comprobamos error en la conexión
if (DB::isError($db)) {
   die($db->getMessage());
}
?>
```

a. Nuevo cliente. Página de resultado de la inserción de un nuevo cliente

```
<?php
// Incluimos el fichero con los datos de la conexión.
include_one 'datosconexion.php';

// Utilizamos el método quoteSmart() para evitar que determinados caracteres
// (intencionados o no) puedan romper la sintaxis de la sentencia SQL.
// El método insertará automáticamente comillas alrededor de las cadenas
// de texto, o tratará los valores NULL correctamente según el SGBD, etc.
$db->query("INSERT INTO CLIENTE VALUES ("
   . $db->quoteSmart($_REQUEST['nombre']) . ","
   . $db->quoteSmart($_REQUEST['dni']). ","
   . $db->quoteSmart($_REQUEST['telefono']) . ","
   . $db->quoteSmart($_REQUEST['email']) . ")");

if (DB::isError($db)) {
   echo "<h2>Error al insertar el cliente</h2>";
   die($db->getMessage());
}

$db->disconnect();
```

c. Cambio de estado de la petición:

```
<?php
include_once 'datosconexion.php';

// Vamos a trabajar con todas las operaciones de esta página en forma de
// transacción, ya que, si se produce un error al insertar una nota o un
// material, la petición puede quedar en un estado erróneo.
$db->autoCommit(false);

// Le decimos a PHP que almacene en un buffer la salida, para poder así
// rectificar en caso de producirse un error.
ob_start();

// Suponemos que los estados nos llegan directamente con los valores
// soportados por el dominio, por ejemplo, a partir de los valores
// fijos de un desplegable.

// Suponemos que la fecha de inicio y fecha de fin nos llegan en formato
// español dd/mm/yyyy y los convertimos a YYYY-mm-dd según ISO 8601.

// Suponemos que el tiempo empleado nos llega en dos campos, horas y minutos, de
// forma que concatenándolos e insertando un ":" en medio, obtenemos una
// hora en formato ISO 8601.
```

```
if (isset($_REQUEST['fechainicio']) && !empty($_REQUEST['fechainicio'])) {
   $fechainicio_array=split("/",$_REQUEST['fechainicio']);
   $fechainciioDB=date("Y-M-d", mktime(0, 0, 0, $fechainicio[1], $fechainicio[0],
$fechainicio[2]));
} else {
   $fechainicioDB=NULL;
}

if (isset($_REQUEST['fechafin']) && !empty($_REQUEST['fechafin'])) {
   $fechafin_array=split("/",$_REQUEST['fechafin']);
   $fechafinDB=date("Y-M-d", mktime(0, 0, 0, $fechafin[1], $fechafin[0], $fechafin[2]));
} else {
   $fechafinDB=NULL;
}

$db->query("UPDATE PETICION SET "
   . "cliente=" . $db->quoteSmart($_REQUEST['cliente']) . ","
   . "resumen=" . $db->quoteSmart($_REQUEST['resumen']) . ","
   . "estado=" . $db->quoteSmart($_REQUEST['estado']) . ","
   . "fechainicio=" . $db->quoteSmart($fechainicioDB) . ","
   . "fechafin" . $db->quoteSmart($fechafinDB) . ","
   . "tiempoempleado=". $db->quoteSmart($_REQUEST['hora'] . ":" . $_REQUEST['minutos']));

if (DB::isError($db)) {
   echo "<h2>Error al insertar el cliente</h2>";
   ob_flush();
   die($db->getMessage());
} else {
   echo "<h2>Petición actualizada correctamente</h2>";
}

// Comprobamos si han añadido alguna nota
if (isset($_REQUEST['texto_nota']) && !empty($_REQUEST['texto_nota'])) {
   // Tenemos el identificador de petición en $_REQUEST['referencia']
   $db->query("INSERT INTO NOTA_PETICION VALUES ("
   . $db->quoteSmart($_REQUEST['referencia']) . ","
   . $db->quoteSmart($_REQUEST['texto_nota']) . ","
   . $db->quoteSmart(date("Y-M-d",mktime())) . ","
   . $db->quoteSmart($_REQUEST['nifEmpleado']) . ")");
   if (DB::isError($db)) {
      ob_clean();
      echo "<h2>Error al insertar la nota. Datos de la petición no actualizados</h2>";
      ob_flush();
      $db->rollback();
      die($db->getMessage());
   } else {
      echo "<h3>Nota actualizada correctamente</h2>";
   }
}
```

```
// Comprobamos si han añadido algún material+
if (isset($_REQUEST['nombrematerial']) && !empty($_REQUEST['nombrematerial'])) {
   // Tenemos el identificador de la petición en $_REQUEST['referencia']
   $db->query("INSERT INTO MATERIAL_PETICION VALUES "
   . $db->quoteSmart($_REQUEST['nombrematerial']) . ","
   . $db->quoteSmart($_REQUEST['referencia']) . ","
   . $db->quoteSmart($_REQUEST['precio']) . ","
   . $db->quoteSmart($_REQUEST['cantidad']) . ")");
   if (DB::isError($db)) {
      ob_clean();
      echo "<h2>Error al insertar el material. Datos de la petición no actualizados</h2>";
      ob_flush();
      $db->rollback();
      die($db->getMessage());
   } else {
      echo "<h3>Material actualizado correctamente</h2>";
   }
}

$db->commit();
$ob_flush();

$db->disconnect();
?>
```

f. y g. Peticiones abiertas de un cliente y resumen de los materiales usados en cada una

```
<?php
include_once 'datosconexion.php';

// Buscamos las peticiones abiertas de un cliente
$res=$db->query("SELECT P.referencia, P.resumen, P.estado, P.fecharecepcion FROM
PETICION P JOIN CLIENTE C ON P.cliente=C.nif WHERE ESTADO NOT IN ('Resuelta',
'Cerrada') ORDER BY fecharecepcion");

if (DB::isError($db)) {
   die($db->getMessage());
}

// Antes de empezar la iteración por las peticiones, vamos a preparar la consulta
// correspondiente a los materiales empleados en cada una.
$queryMaterial=$db->prepare("SELECT SUM(M.precio) as precioMateriales,
COUNT(M.nombrematerial) numMateriales FROM MATERIAL_PETICION M WHERE M.peticion=? ");

if (DB::isError($db)) {
   die($db->getMessage());
}
```

```
echo "<table>";
echo "<tr><th>Referencia</th><th>Resumen</th><th>Estado</th><th>Duración</th><th>
Fecharecepción</th><th>Precio materiales</th><th>Núm. materiales</th></tr>";
while($res->fetchInto($row,DB_FETCHMODE_ASSOC)) {
   echo "<tr>";
   echo "<td>" . $row['referencia'] . "</td>";
   echo "<td>" . $row['resumen'] . "</td>";
   echo "<td>" . $row['estado'] . "</td>";
   echo "<td>" . $row['fecharecepcion'] . "</td>";
   $resMaterial=$db->execute($queryMaterial,$row['referencia']);
   if (DB::isError($db)) {
      die($db->getMessage());
   } else {
      $res->fetchInto($rowMaterial,DB_FETCHMODE_ASSOC);
      echo "<td>" . $rowMaterial['precioMateriales'] . "</td>";
      echo "<td>" . $rowMaterial['numMateriales'] . "</td>";
   }
   echo "</tr>";
}

echo "</table>";
?>
```

Mediante estos tres ejemplos, disponemos ya de una base tanto de código, como de estilo y mecanismos de comprobación de error, para desarrollar el resto de la aplicación, sin tener en cuenta su diseño.

Hemos intentado escoger consultas y operaciones representativas del funcionamiento de la aplicación y, a la vez, que se correspondieran con las vistas en apartados anteriores. Además, hemos introducido algunas funciones PHP que suelen utilizarse en combinación con el trabajo en bases de datos para tipos concretos, y para el tratamiento de errores, para evitar que el usuario reciba información confusa en la página de resultados.

Resumen

En esta unidad hemos visto las actividades más destacadas de las fases iniciales de un proyecto de desarrollo con conexión a bases de datos.

Más que resolver el propio caso, se trataba de identificar los aspectos clave de los casos reales y enlazarlos con el contenido del resto de unidades del curso.

Habréis podido identificar qué actividades son más relevantes para el resultado final del proyecto, en cuáles conviene invertir más tiempo y cuáles no son tan críticas para los objetivos de éste u otro caso similar.

Si habéis seguido la planificación sugerida y repasado cada unidad a la que que se hacía referencia, habréis podido comprender la aplicación práctica del material y se habrán alcanzado los objetivos que nos proponíamos al redactar esta unidad didáctica.

También es posible leer este caso de estudio como un capítulo final del curso, donde se desarrolla un ejemplo completo. Se ha intentado redactarlo con este doble cometido.

Apéndice:GNU Free Documentation License

UOC
Formación de Posgrado

GNU Free Documentation License

GNU Free Documentation License
Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software. We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions

stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent.

An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions

whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition.

Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material.

If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for

example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number.

Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit.

When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form.

Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their

copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.


UOC
La universidad virtual
Formación de Posgrado