

Apuntes de Sistemas Gestores de Bases de Datos

Curso 2006/2007
2º Curso de Administración de Sistemas Informáticos



Esta obra está bajo una licencia de Creative Commons.
Autor: Jorge Sánchez Asenjo (año 2007) <http://www.jorgesanchez.net>
e-mail: info@jorgesanchez.net

Esta obra está bajo una licencia de Reconocimiento-NoComercial-CompartirIgual de Creative Commons
Para ver una copia de esta licencia, visite:
<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>
o envíe una carta a:
Creative Commons, 559 Nathan Abbot



Reconocimiento-NoComercial-CompartirIgual 2.5 España

Usted es libre de:



copiar, distribuir y comunicar públicamente la obra



hacer obras derivadas

Bajo las condiciones siguientes:



Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciadador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).



No comercial. No puede utilizar esta obra para fines comerciales.



Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
- Apart from the remix rights granted under this license, nothing in this license impairs or restricts the author's moral rights.

Advertencia

Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.

Esto es un resumen legible por humanos del texto legal (la licencia completa) disponible en los idiomas siguientes:

Catalán Castellano Euskera Gallego

Para ver una copia completa de la licencia, acudir a la dirección
<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>

(Unidad 1)

Gestión de Datos

(1.1) introducción

(1.1.1) sistemas gestores de bases de datos

la necesidad de gestionar datos

En el mundo actual existe una cada vez mayor demanda de datos. Esta demanda siempre ha sido patente en empresas y sociedades, pero en estos años la demanda todavía se ha disparado más debido al acceso multitudinario a las redes integradas en Internet y a la aparición de pequeños dispositivos (móviles y PDA) que también requieren esa información.

En informática se conoce como **dato** a cualquier elemento informativo que tenga relevancia para un usuario. Desde su nacimiento, la informática se ha encargado de proporcionar herramientas que faciliten la gestión de los datos.

Antes de la aparición de las aplicaciones informáticas, las empresas tenían como únicas herramientas de gestión de datos a los cajones, carpetas y fichas en las que se almacenaban los datos. En este proceso manual, el tipo requerido para manipular estos datos era enorme. Sin embargo el proceso de aprendizaje era relativamente sencillo ya que se usaban elementos que el usuario reconocía perfectamente.

Por esa razón, la informática ha adaptado sus herramientas para que los elementos que el usuario maneja en el ordenador se parezcan a los que utilizaba manualmente. Por eso en informática se sigue hablando de ficheros, formularios, carpetas, directorios,....

componentes de un sistema de información electrónico

En el caso de una gestión electrónica de la información (lo que actualmente se considera un sistema de información), los componentes son:

- ◆ **Datos.** Se trata de la información relevante que almacena y gestiona el sistema de información
- ◆ **Hardware.** Equipamiento físico que se utiliza para gestionar los datos
- ◆ **Software.** Aplicaciones que permiten el funcionamiento adecuado del sistema

- ◆ **Recursos humanos.** Personal que maneja el sistema de información

(1.1.2) tipos de sistemas de información

En la evolución de los sistemas de información ha habido dos puntos determinantes, que han formado los dos tipos fundamentales de sistemas de información.

sistemas de información orientados al proceso

En estos sistemas de información se crean diversas aplicaciones (software) para gestionar diferentes aspectos del sistema. Cada aplicación realiza unas determinadas operaciones. Los datos de dichas aplicaciones se almacenan en archivos digitales dentro de las unidades de almacenamiento del ordenador (a veces en archivos binarios, o en hojas de cálculo, o incluso en archivos de texto).

Cada programa almacena y utiliza sus propios datos de forma un tanto caótica. La ventaja de este sistema (la única ventaja), es que los procesos eran independientes por lo que la modificación de uno no afectaba al resto. Pero tiene grandes inconvenientes:

- ◆ **Datos redundantes.** Ya que se repiten continuamente
- ◆ **Datos inconsistentes.** Ya que un proceso cambia sus datos y no el resto. Por lo que el mismo dato puede tener valores distintos según qué aplicación acceda a él.
- ◆ **Coste de almacenamiento elevado.** Al almacenarse varias veces el mismo dato, se requiere más espacio en los discos. Luego se agotarán antes.
- ◆ **Difícil acceso a los datos.** Cada vez que se requiera una consulta no prevista inicialmente, hay que modificar el código de las aplicaciones o incluso crear una nueva aplicación.
- ◆ **Dependencia de los datos a nivel físico.** Para poder saber cómo se almacenan los datos, es decir qué estructura se utiliza de los mismos, necesitamos ver el código de la aplicación; es decir el código y los datos no son independientes.
- ◆ **Tiempos de procesamiento elevados.** Al no poder optimizar el espacio de almacenamiento.
- ◆ **Dificultad para el acceso simultáneo a los datos.** Es casi imposible de conseguir ya que se utilizan archivos que no admiten esta posibilidad. Dos usuarios no pueden acceder a los datos de forma concurrente.
- ◆ **Dificultad para administrar seguridad.** Ya que cada aplicación se crea independientemente; es por tanto muy difícil establecer criterios de seguridad uniformes.

A estos sistemas se les llama sistemas de gestión de ficheros. Se consideran también así a los sistemas que utilizan programas ofimáticos (como **Word** y

Excel) para gestionar sus datos. De hecho estos sistemas producen los mismos (si no más) problemas.

sistemas de información orientados a los datos. bases de datos

En este tipo de sistemas los datos se centralizan en una **base de datos** común a todas las aplicaciones. Estos serán los sistemas que estudiaremos en este curso.

En esos sistemas los datos se almacenan en una única estructura lógica que es utilizable por las aplicaciones. A través de esa estructura se accede a los datos que son comunes a todas las aplicaciones.

(1.1.3) ventajas

- ◆ **Independencia de los datos y los programas y procesos.** Esto permite modificar los datos sin modificar el código de las aplicaciones.
- ◆ **Menor redundancia.** No hace falta tanta repetición de datos. Sólo se indica la forma en la que se relacionan los datos.
- ◆ **Integridad de los datos.** Mayor dificultad de perder los datos o de realizar incoherencias con ellos.
- ◆ **Mayor seguridad en los datos.** Al permitir limitar el acceso a los usuarios. Cada tipo de usuario podrá acceder a unas cosas..
- ◆ **Datos más documentados.** Gracias a los **metadatos** que permiten describir la información de la base de datos.
- ◆ **Acceso a los datos más eficiente.** La organización de los datos produce un resultado más óptimo en rendimiento.
- ◆ **Menor espacio de almacenamiento.** Gracias a una mejor estructuración de los datos.
- ◆ **Acceso simultáneo a los datos.** Es más fácil controlar el acceso de usuarios de forma concurrente.

desventajas

- ◆ **Instalación costosa.** El control y administración de bases de datos requiere de un software y hardware poderoso
- ◆ **Requiere personal cualificado.** Debido a la dificultad de manejo de este tipo de sistemas.
- ◆ **Implantación larga y difícil.** Debido a los puntos anteriores. La adaptación del personal es mucho más complicada y lleva bastante tiempo.
- ◆ **Ausencia de estándares reales.** Lo cual significa una excesiva dependencia hacia los sistemas comerciales del mercado. Aunque, hoy en día, una buena parte de esta tecnología está aceptada como estándar de hecho.

(1.1.4) objetivo de los sistemas gestores de bases de datos

Un sistema gestor de bases de datos o **SGBD** (aunque se suele utilizar más a menudo las siglas **DBMS** procedentes del inglés, **Data Base Management System**) es el software que permite a los usuarios procesar, describir, administrar y recuperar los datos almacenados en una base de datos.

En estos Sistemas se proporciona un conjunto coordinado de programas, procedimientos y lenguajes que permiten a los distintos usuarios realizar sus tareas habituales con los datos, garantizando además la seguridad de los mismos.

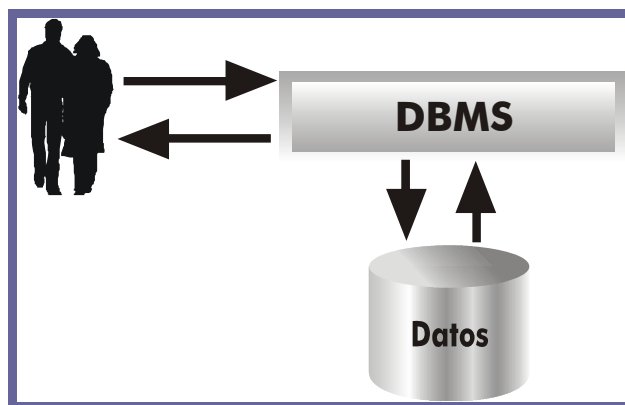


Ilustración 1, Esquema del funcionamiento y utilidad de un sistema gestor de bases de datos

El éxito del SGBD reside en mantener la seguridad e integridad de los datos. Lógicamente tiene que proporcionar herramientas a los distintos usuarios. Entre las herramientas que proporciona están:

- ◆ **Herramientas para la creación y especificación de los datos.** Así como la estructura de la base de datos.
- ◆ **Herramientas para administrar y crear la estructura física** requerida en las unidades de almacenamiento.
- ◆ **Herramientas para la manipulación de los datos** de las bases de datos, para añadir, modificar, suprimir o consultar datos.
- ◆ **Herramientas de recuperación** en caso de desastre
- ◆ **Herramientas para la creación de copias de seguridad**
- ◆ **Herramientas para la gestión de la comunicación** de la base de datos
- ◆ **Herramientas para la creación de aplicaciones** que utilicen esquemas externos de los datos
- ◆ **Herramientas de instalación** de la base de datos
- ◆ Herramientas para la exportación e importación de datos



(1.1.5) niveles de abstracción de una base de datos

introducción

En cualquier sistema de información se considera que se pueden observar los datos desde dos puntos vista:

- ◆ **Vista externa.** Esta es la visión de los datos que poseen los usuarios del Sistema de Información.
- ◆ **Vista física.** Esta es la forma en la que realmente están almacenados los datos.

En un Sistema de ficheros, los usuarios ven los datos desde las aplicaciones creadas por los programadores. Esa vista pueden ser formularios, informes visuales o en papel,... Pero la realidad física de esos datos, tal cual se almacenan en los discos, no la ven. Esa visión está reservada a los administrados.

En el caso de los Sistemas de Base de datos, se añade una tercera vista, que es la vista conceptual. Esa vista se sitúa entre la física y la externa. Se habla pues en Bases de datos de la utilización de tres esquemas para representar los datos.

esquema físico

Representa la forma en la que están almacenados los datos. Esta visión sólo la requiere el **administrador**. El administrador la necesita para poder gestionar más eficientemente la base de datos.

En este esquema es donde aparecen las unidades de disco, archivos y carpetas del sistema.

esquema conceptual

Se trata de un esquema teórico de los datos en la que figuran organizados en estructuras reconocibles del mundo real y en el que también aparece la forma de relacionarse los datos.

Esta estructura es utilizada por el **desarrollador**, que necesita conocerla para al crear aplicaciones saber como acceder. Todas las aplicaciones que se crean sobre la base de datos utilizan este esquema.

Realmente cuando se habla del diseño de la base de datos, se suele referir a este esquema. El esquema conceptual lo realiza el **diseñador** de la base de datos.

esquema externo

Se trata de la visión de los datos que poseen los **usuarios finales**. Esa visión es la que obtienen a través de las aplicaciones. Las aplicaciones creadas por los desarrolladores abstraen la realidad conceptual de modo que el usuario no conoce las relaciones entre los datos, como tampoco conoce todos los datos que realmente se almacenan.

Realmente cada aplicación produce un esquema externo diferente (aunque algunos pueden coincidir) o **vistas de usuario**. El conjunto de todas las vistas de usuario es lo que se denomina **esquema externo global**.

(1.2) componentes de los SGBD

(1.2.1) funciones. lenguajes de los SGBD

Los SGBD tienen que realizar tres tipos de funciones para ser considerados válidos.

función de descripción o definición

Permite al diseñador de la base de datos crear las estructuras apropiadas para integrar adecuadamente los datos. Esta función es la que permite definir las tres estructuras de la base de datos (relacionadas con sus tres esquemas).

- ◆ Estructura interna
- ◆ Estructura conceptual
- ◆ Estructura externa

Esta función se realiza mediante el **lenguaje de descripción de datos** o **DDL**. Mediante ese lenguaje:

- ◆ Se definen las estructuras de datos
- ◆ Se definen las relaciones entre los datos
- ◆ Se definen las reglas que han de cumplir los datos

función de manipulación

Permite modificar y utilizar los datos de la base de datos. Se realiza mediante el **lenguaje de modificación de datos** o **DML**. Mediante ese lenguaje se puede:

- ◆ Añadir datos
- ◆ Eliminar datos
- ◆ Modificar datos
- ◆ Buscar datos

Actualmente se suele distinguir la función de buscar datos respecto del resto. Para lo cual se proporciona un **lenguaje de consulta de datos** o **DQL**.

función de control

Mediante esta función los administradores poseen mecanismos para determinar las visiones de los datos permitidas a cada usuario, además de proporcionar elementos de creación y modificación de esos usuarios.

Se suelen incluir aquí las tareas de copia de seguridad, carga de ficheros, auditoría, protección ante ataques externos, configuración del sistema,...

El lenguaje que implementa esta función es el **lenguaje de control de datos o DCL**.

(1.2.2) recursos humanos de las bases de datos

Intervienen (como ya se ha comentado) muchas personas en el desarrollo y manipulación de una base de datos. Habíamos seleccionado cuatro tipos de usuarios (administradores/as, desarrolladores, diseñadores/as y usuarios/as). Ahora vamos a desglosar aún más esta clasificación.

informáticos

Lógicamente son los profesionales que definen y preparan la base de datos. Pueden ser:

- ◆ **Directivos/as.** Son los organizadores y coordinadores del proyecto a desarrollar. Esto significa que son los encargados de decidir los recursos que se pueden utilizar, planificar el tiempo y las tareas, la atención al usuario y de dirigir las entrevistas y reuniones pertinentes.
- ◆ **Analistas.** Son los encargados de controlar el desarrollo de la base de datos aprobada por la dirección. Son además los diseñadores de la base de datos (especialmente de los esquemas interno y conceptual) y los coordinadores de la programación de la misma.
- ◆ **Administradores/as de las bases de datos.** Definen la seguridad de la base de datos y gestionan las copias de seguridad y la gestión física de la base de datos. Los analistas suelen tener esta funcionalidad cuando la base de datos está creada.
- ◆ **Desarrolladores/as o programadores/as.** Encargados de la realización de las aplicaciones de usuario de la base de datos.
- ◆ **Equipo de mantenimiento.** Encargados de dar soporte a los usuarios en el trabajo diario (suelen incorporar además tareas administrativas).

usuarios

- ◆ **Expertos/as.** Utilizan el lenguaje de manipulación de datos (DML) para acceder a la base de datos. Son usuarios que utilizan la base de datos para gestión avanzada de decisiones.
- ◆ **Habituales.** Utilizan las aplicaciones creadas por los desarrolladores para consultar y actualizar los datos. Son los que trabajan en la empresa a diario con estas herramientas y el objetivo fundamental de todo el desarrollo de la base de datos.

Sistemas Gestores de Bases de Datos

(Unidad 1) Sistemas Gestores de Bases de Datos

- ◆ **Ocasionales.** Son usuarios que utilizan un acceso mínimo a la base de datos a través de una aplicación que permite consultar ciertos datos. Serían por ejemplo los usuarios que consultan el horario de trenes a través de Internet.

(1.2.3) estructura multicapa

Un SGBD está en realidad formado por varias capas que actúan como interfaces entre el usuario y los datos. El propio ANSI/X3/SPARC introdujo una mejora de su modelo en 1988 a través de un grupo de trabajo llamado **UFTG (User Facilities Task Group)**, grupo de trabajo para las facilidades de usuario). Este modelo toma como objeto principal, al usuario habitual de la base de datos y orienta el funcionamiento de la base de datos de modo que este usuario ignora el funcionamiento externo.

Desde esta óptica para llegar a los datos hay que pasar una serie de capas que poco a poco van entrando más en la realidad física de la base de datos. Esa estructura se muestra en la siguiente figura:



Ilustración 2, Modelo de referencia de las facilidades de usuario

(1.2.4) núcleo

El núcleo de la base de datos es el encargado de traducir las operaciones que le llegan a instrucciones ejecutables por el sistema operativo en el lenguaje que éste último requiera.

(1.2.5) diccionario de datos

Se trata del elemento que posee todos los metadatos. Gracias a esta capa las solicitudes de los clientes se traducen en instrucciones que hacen referencia al esquema interno de la base de datos. La capa de acceso a datos es la que permite comunicar a las aplicaciones de usuario con el diccionario de datos a través de las herramientas de gestión de datos que incorpore el SGBD.

(1.2.6) facilidades de usuario

Son las herramientas que proporciona el SGBD a los usuarios para permitir un acceso más sencillo a los datos. Actúan de interfaz entre el usuario y la base de datos, y son el único elemento que maneja el usuario.

(1.2.7) funcionamiento del SGBD

El esquema siguiente presenta el funcionamiento típico de un SGBD:

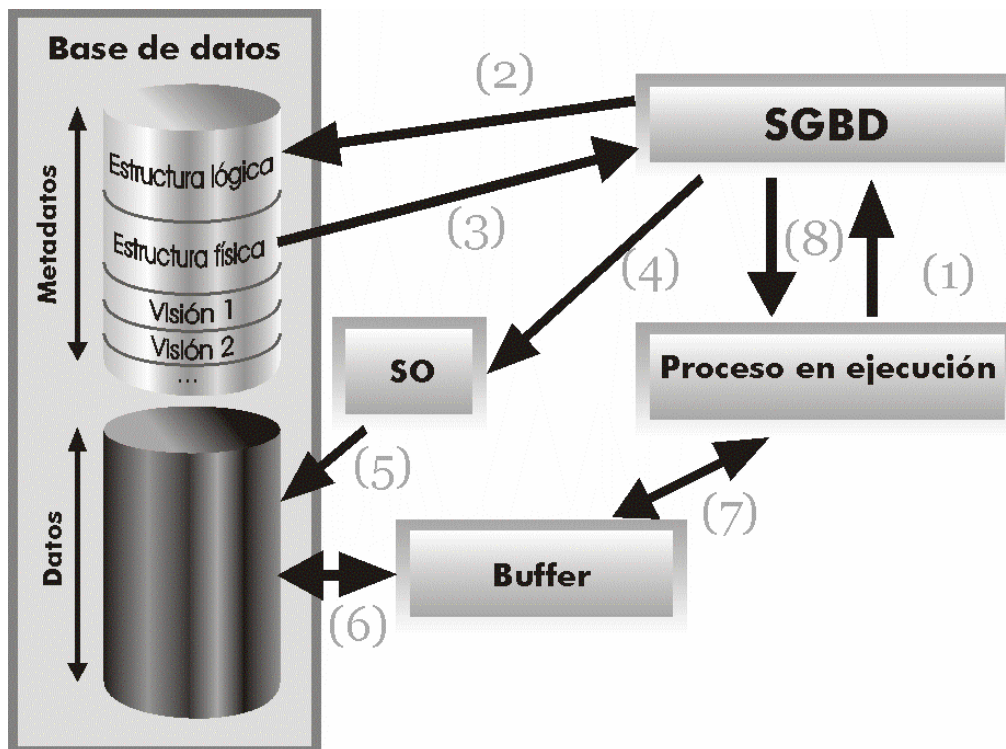


Ilustración 3, Esquema del funcionamiento de un SGBD

El esquema anterior reproduce la comunicación entre un proceso de usuario que desea acceder a los datos y el SGBD:

- (1) El proceso lanzado por el usuario llama al SGBD indicando la porción de la base de datos que se desea tratar

Sistemas Gestores de Bases de Datos

(Unidad 1) Sistemas Gestores de Bases de Datos

- (2) El SGBD traduce la llamada a términos del esquema lógico de la base de datos. Accede al esquema lógico comprobando derechos de acceso y la traducción física
- (3) El SGBD obtiene el esquema físico
- (4) El SGBD traduce la llamada a los métodos de acceso del Sistema Operativo que permiten acceder a los datos requeridos
- (5) El Sistema Operativo accede a los datos tras traducir las órdenes dadas por el SGBD
- (6) Los datos pasan del disco a una memoria intermedia o buffer. En ese buffer se almacenarán los datos según se vayan recibiendo
- (7) Los datos pasan del buffer al área de trabajo del usuario (ATU) del proceso del usuario.
- (8) El SGBD devuelve indicadores en los que manifiesta si ha habido errores o advertencias a tener en cuenta. Esto se indica al área de comunicaciones del proceso de usuario. Si las indicaciones son satisfactorias, los datos de la ATU serán utilizables por el proceso de usuario.

(1.3) arquitectura de los SGBD. estándares

Es uno de los aspectos que todavía sigue pendiente. Desde la aparición de los primeros gestores de base de datos se intentó llegar a un acuerdo para que hubiera una estructura común para todos ellos, a fin de que el aprendizaje y manejo de este software fuera más provechoso y eficiente.

El acuerdo nunca se ha conseguido del todo, no hay estándares aceptados del todo. Aunque sí hay unas cuentas propuestas de estándares que sí funcionan como tales.

(1.3.1) organismos de estandarización

Los intentos por conseguir una estandarización han estado promovidos por organismos de todo tipo. Algunos son estatales, otros privados y otros promovidos por los propios usuarios. Los tres que han tenido gran relevancia en el campo de las bases de datos son **ANSI/SPARC/X3**, **CODASYL** y **ODMG**. Los organismos grandes (que recogen grandes responsabilidades) dividen sus tareas en comités, y éstos en grupos de trabajo que se encargan de temas concretos.

(1.3.2) SC21

- ◆ **ISO (International Organization for Standardization)**. Es un organismo internacional de definición de estándares de gran prestigio.

- ◆ **IEC (International Electrotechnical Commission)**. Organismo de definición de normas en ambientes electrónicos
- ◆ **JTC1 (Joint Technical Committee)**. Comité formado por los dos organismos anteriores encargado de diversos proyectos. En el campo de las bases de datos, el subcomité **SC21** (en el que participan otros organismos nacionales, como el español AENOR) posee un grupo de trabajo llamado **WG 3** que se dedica a las bases de datos. Este grupo de trabajo es el que define la estandarización del lenguaje SQL entre otras cuestiones.

(1.3.3) DBTG Codasyl

Codasyl (**Conference on Data System Languages**) es el nombre de una conferencia de finales de los años 60 en la que participaron organismos privados y públicos del gobierno de Estados Unidos con la finalidad de definir estándares (Codasyl definió el lenguaje COBOL) para la informática de gestión.

De ahí salió DBTG (**Data Base Task Group**, grupo de tareas para bases de datos) grupo que definió el modelo en red de bases de datos que desde entonces se llama Codasyl o DBTG y que fue aceptado por la ANSI

(1.3.4) ANSI/X3/SPARC

ANSI (American National Science Institute) es un organismo científico de Estados Unidos que ha definido diversos estándares en el campo de las bases de datos. **X3** es la parte de ANSI encargada de los estándares en el mundo de la electrónica. Finalmente **SPARC, System Planning and Repairs Committee**, comité de planificación de sistemas y reparaciones es una subsección de X3 encargada de los estándares en Sistemas Informáticos en especial del campo de las bases de datos. Su logro fundamental ha sido definir un modelo de referencia para las bases de datos (que se estudiará posteriormente).

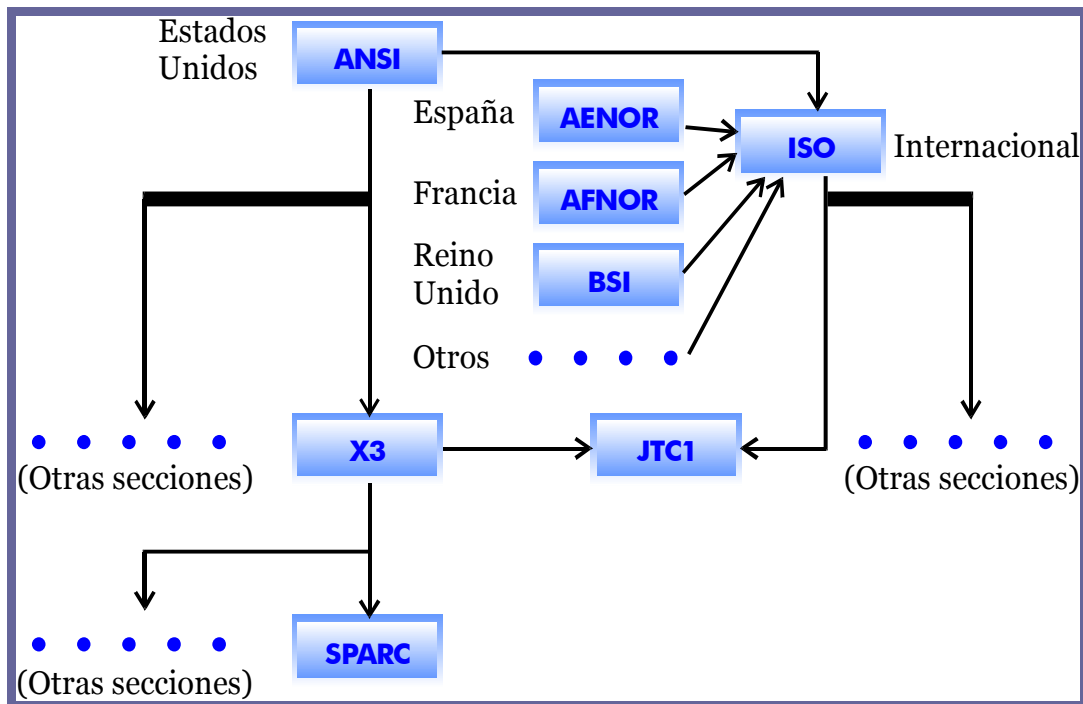


Ilustración 4, Relación entre los organismos de estandarización

En la actualidad ANSI para Estados Unidos e ISO para todo el mundo son nombres equivalentes en cuanto a estandarización de bases de datos, puesto que se habla ya de un único modelo de sistema de bases de datos.

(1.3.5) Modelo ANSI/X3/SPARC

El grupo ANSI ha marcado la referencia para la construcción de SGBD. Se basa en los estudios anteriores en los que se indicaban tres niveles de abstracción de la base de datos. ANSI profundiza más en esta idea y define cómo debe ser el proceso de creación y utilización de estos niveles.

En el modelo ANSI se indica que hay tres modelos (**externo**, **conceptual** e **interno**) entendiéndose por modelo las normas que permiten crear esquemas (diseños de la base de datos). Los esquemas externos reflejan la información preparada para el usuario final, el esquema conceptual refleja los datos y relaciones de la base de datos y el esquema interno la preparación de los datos para ser almacenados.

El esquema conceptual contiene la información lógica de la base de datos. Su estructuración y las relaciones que hay entre los datos. Se trata de la propuesta teórica de los datos (es quizá la más importante).

El esquema interno contiene información sobre cómo están almacenados los datos en disco. Es el esquema más cercano a la organización real de los datos.

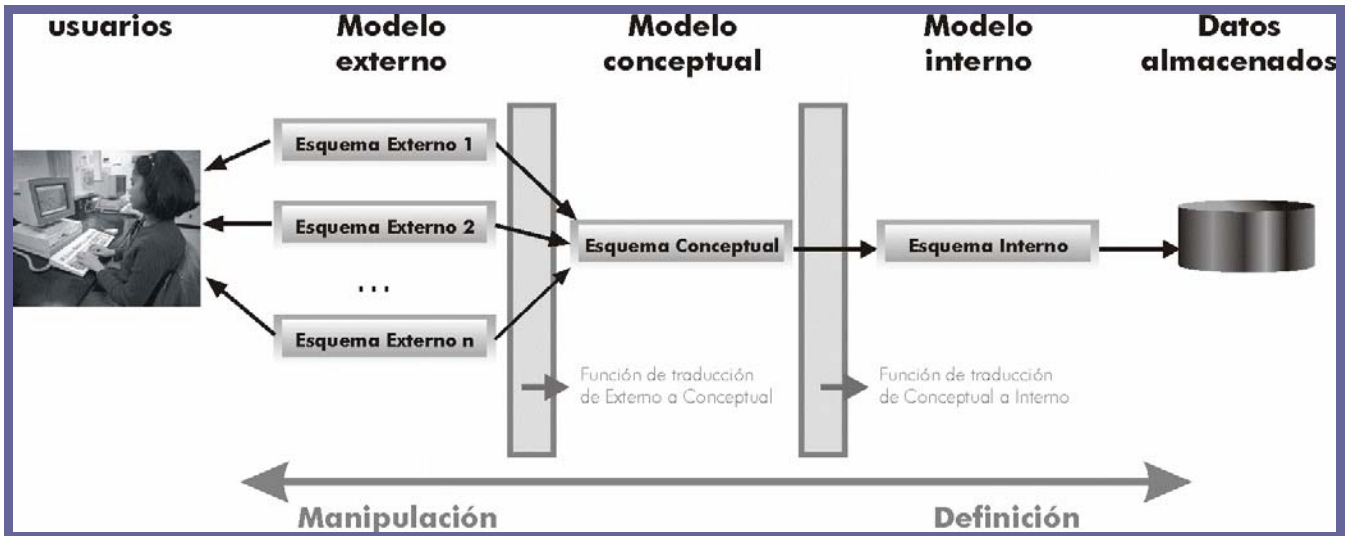


Ilustración 5, Niveles en el modelo ANSI

LEYENDA

- Funciones humanas
- Metadatos
- Funciones del programa
- Interfaces

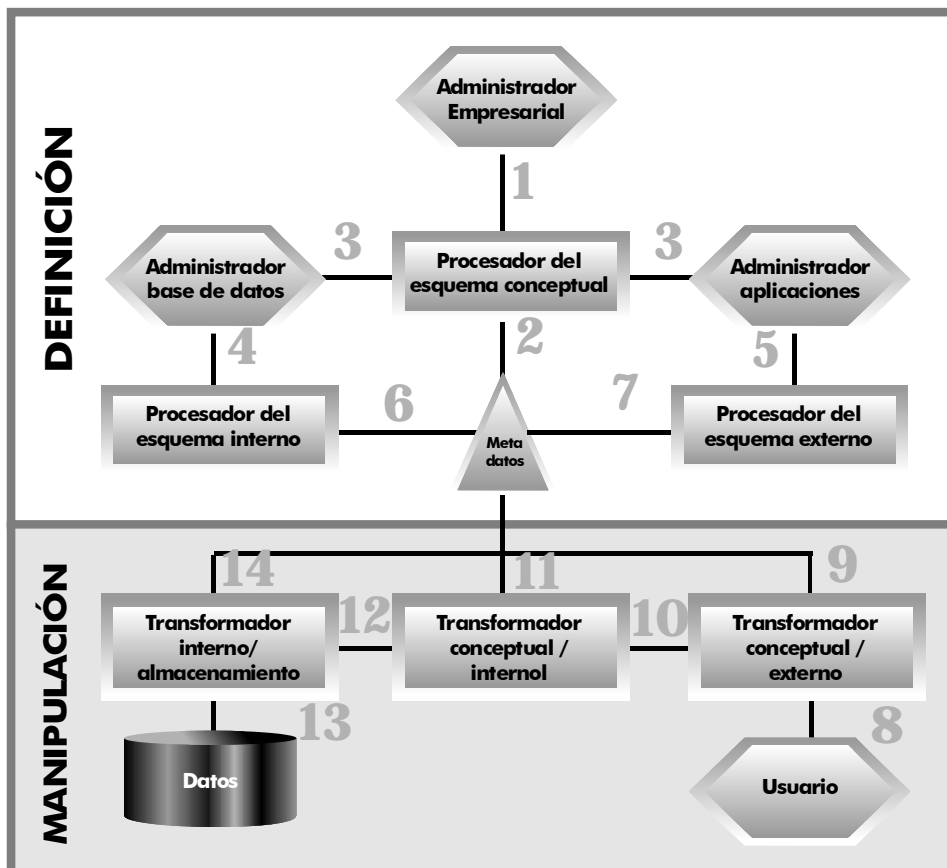


Ilustración 6, Arquitectura ANSI

El paso de un esquema a otro se realiza utilizando un interfaz o función de traducción. En su modelo, la ANSI no indica cómo se debe realizar esta función, sólo que debe existir.

La arquitectura completa (Ilustración 6) esta dividida en dos secciones, la zona de definición de datos y la de manipulación. Esa arquitectura muestra las funciones realizadas por humanos y las realizadas por programas.

En la fase de **definición**, una serie de interfaces permiten la creación de los **metadatos** que se convierten en el eje de esta arquitectura. La creación de la base de datos comienza con la elaboración del esquema conceptual realizándola el administrador de la empresa (actualmente es el diseñador, pero ANSI no lo llamó así). Ese esquema se procesa utilizando un procesador del esquema conceptual (normalmente una herramienta CASE, **interfaz 1** del dibujo anterior) que lo convierte en los metadatos (**interfaz 2**).

La **interfaz 3** permite mostrar los datos del esquema conceptual a los otros dos administradores: el administrador de la base de datos y el de aplicaciones (el desarrollador). Mediante esta información construyen los esquemas internos y externos mediante las **interfaces 4 y 5** respectivamente, los procesadores de estos esquemas almacenan la información correspondiente a estos esquemas en los metadatos (**interfaces 6 y 7**).

En la fase de **manipulación** el usuario puede realizar operaciones sobre la base de datos usando la **interfaz 8** (normalmente una aplicación) esta petición es transformada por el transformador externo/conceptual que obtiene el esquema correspondiente ayudándose también de los metadatos (**interfaz 9**). El resultado lo convierte otro transformador en el esquema interno (**interfaz 10**) usando también la información de los metadatos (**interfaz 11**). Finalmente del esquema interno se pasa a los datos usando el último transformador (**interfaz 12**) que también accede a los metadatos (**interfaz 13**) y de ahí se accede a los datos (**interfaz 14**). Para que los datos se devuelvan al usuario en formato adecuado para él se tiene que hacer el proceso contrario (observar dibujo).

(1.3.6) estructuras operacionales

Actualmente casi todos los sistemas gestores de base de datos poseen también la misma idea operacional en la que se entiende que la base de datos se almacena en un servidor y hay una serie de clientes que pueden acceder a los datos del mismo. Las posibilidades son:

- ◆ **Estructura Cliente-Servidor.** Estructura clásica, la base de datos y su SGBD están en un servidor al cual acceden los clientes. El cliente posee software que permite al usuario enviar instrucciones al SGBD en el servidor y recibir los resultados de estas instrucciones. Para ello el software cliente y el servidor deben utilizar software de comunicaciones en red.
- ◆ **Cliente multi-servidor.** Ocurre cuando los clientes acceden a datos situados en más de un servidor. También se conoce esta estructura como **base de datos distribuida**. El cliente no sabe si los datos están en uno o más servidores, ya que el resultado es el mismo independientemente de dónde se almacenan los datos. En esta estructura hay un servidor de aplicaciones que

es el que recibe las peticiones y el encargado de traducirlas a los distintos servidores de datos para obtener los resultados. Una posibilidad muy extendida hoy en día es la posibilidad **Cliente/Servidor Web/Servidor de datos**, el cliente se conecta a un servidor mediante un navegador web y desde las páginas de este ejecuta las consultas. El servidor web traduce esta consulta al servidor (o servidores) de datos.

(1.4) tipos de SGBD

(1.4.1) introducción

Como se ha visto en los apartados anteriores, resulta que cada SGBD puede utilizar un modelo diferente para los datos. Por lo que hay modelos conceptuales diferentes según que SGBD utilicemos.

No obstante existen modelos lógicos comunes, ya que hay SGBD de diferentes tipos. En la realidad el modelo ANSI se modifica para que existan dos modelos internos: el modelo lógico (referido a cualquier SGBD de ese tipo) y el modelo propiamente interno (aplicable sólo a un SGBD en particular). De hecho en la práctica al definir las bases de datos desde el mundo real hasta llegar a los datos físicos se pasa por los siguientes esquemas:

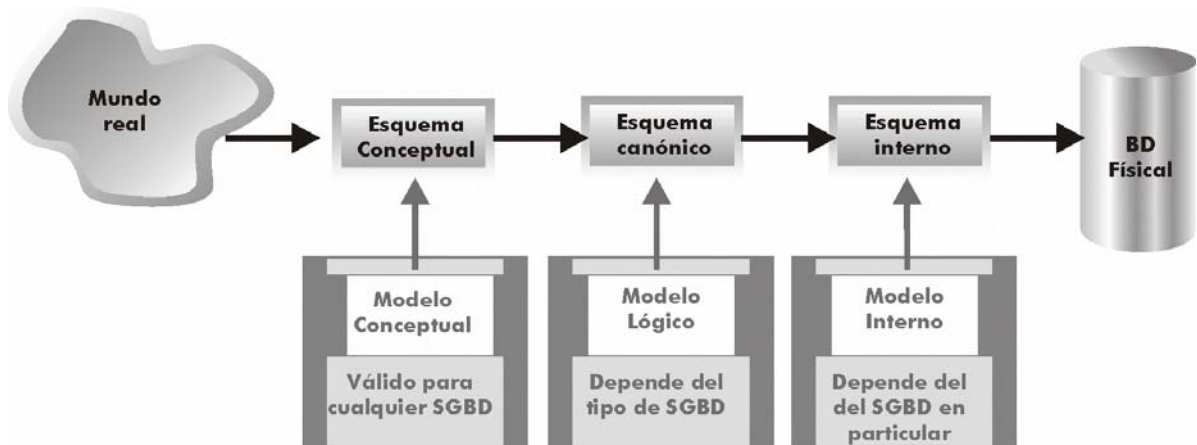


Ilustración 7, Modelos de datos utilizados en el desarrollo de una BD

Por lo tanto la diferencia entre los distintos SGBD está en que proporcionan diferentes modelos lógicos.

diferencias entre el modelo lógico y el conceptual

- ◆ El modelo conceptual es independiente del DBMS que se vaya a utilizar. El lógico depende de un **tipo** de SGBD en particular
- ◆ El modelo lógico es más cercano al ordenador
- ◆ El modelo conceptual es más cercano al usuario, el lógico es el encargado de establecer el paso entre el modelo informático y el modelo físico del sistema.

Sistemas Gestores de Bases de Datos

(Unidad 1) Sistemas Gestores de Bases de Datos

Algunos ejemplos de modelos conceptuales son:

- ◆ **Modelo E/R**
- ◆ **Modelo RM/T**
- ◆ **Modelos semánticos**

Ejemplos de modelos lógicos son:

- ◆ **Modelo relacional**
- ◆ **Modelo Codasyl**
- ◆ **Modelo Jerárquico**

A continuación se comentarán los modelos lógicos más importantes.

(1.4.2) modelo jerárquico

Era utilizado por los primeros SGBD, desde que IBM lo definió para su IMS (**Information Management System**, Sistema Administrador de Información) en 1970. Se le llama también modelo en árbol debido a que utiliza una estructura en árbol para organizar los datos.

La información se organiza con un jerarquía en la que la relación entre las entidades de este modelo siempre es del tipo **padre / hijo**. De esta forma hay una serie de nodos que contendrán atributos y que se relacionarán con nodos hijos de forma que puede haber más de un hijo para el mismo padre (pero un hijo sólo tiene un padre).

Los datos de este modelo se almacenan en estructuras lógicas llamadas **segmentos**. Los segmentos se relacionan entre sí utilizando **arcos**.

La forma visual de este modelo es de árbol invertido, en la parte superior están los padres y en la inferior los hijos.

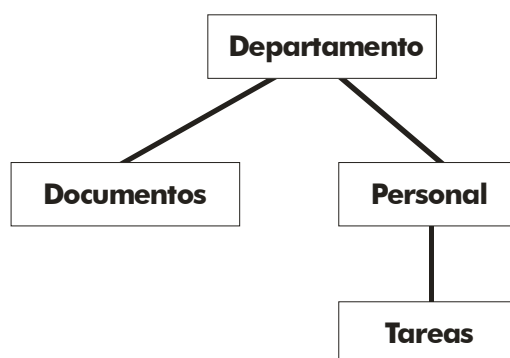


Ilustración 8, Ejemplo de esquema jerárquico

Este esquema está en absoluto desuso ya que no es válido para modelar la mayoría de problemas de bases de datos.

(1.4.3) modelo en red (CodasyI)

Es un modelo que ha tenido una gran aceptación (aunque apenas se utiliza actualmente). En especial se hizo popular la forma definida por CodasyI a principios de los 70 que se ha convertido en el modelo en red más utilizado.

El modelo en red organiza la información en **registros** (también llamados **nodos**) y **enlaces**. En los registros se almacenan los datos, mientras que los enlaces permiten relacionar estos datos. Las bases de datos en red son parecidas a las jerárquicas sólo que en ellas puede haber más de un padre.

En este modelo se pueden representar perfectamente cualquier tipo de relación entre los datos (aunque el CodasyI restringía un poco las relaciones posibles), pero hace muy complicado su manejo.

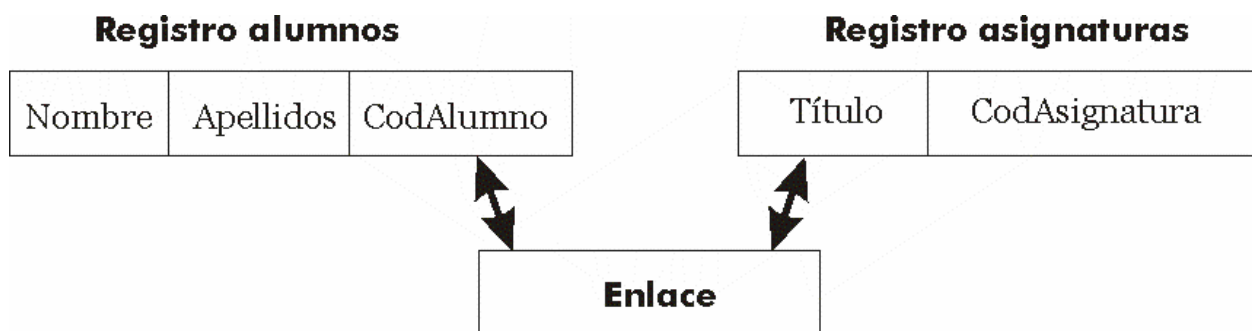


Ilustración 9, ejemplo de diagrama de estructura de datos CodasyI

(1.4.4) modelo relacional

En este modelo los datos se organizan en tablas cuyos datos se relacionan. Es el modelo más popular y se describe con más detalle en los temas siguientes.

(1.4.5) modelo de bases de datos orientadas a objetos

Desde la aparición de la programación orientada a objetos (**POO** u **OOO**) se empezó a pensar en bases de datos adaptadas a estos lenguajes. La programación orientada a objetos permite cohesionar datos y procedimientos, haciendo que se diseñen estructuras que poseen datos (**atributos**) en las que se definen los procedimientos (**operaciones**) que pueden realizar con los datos. En las bases orientadas a objetos se utiliza esta misma idea.

A través de este concepto se intenta que estas bases de datos consigan arreglar las limitaciones de las relacionales. Por ejemplo el problema de la herencia (el hecho de que no se puedan realizar relaciones de herencia entre las tablas), tipos definidos por el usuario, disparadores (triggers) almacenables en la base de datos, soporte multimedia...

Se supone que son las bases de datos de tercera generación (la primera fue las bases de datos en red y la segunda las relacionales), lo que significa que el futuro parece estar a favor de estas bases de datos. Pero siguen sin reemplazar a las relacionales, aunque son el tipo de base de datos que más está creciendo en los últimos años.

Su modelo conceptual se suele diseñar en **UML** y el lógico actualmente en **ODMG** (**Object Data Management Group**, grupo de administración de objetos de datos, organismo que intenta crear estándares para este modelo).

(1.4.6) bases de datos objeto relacionales

Tratan de ser un híbrido entre el modelo relacional y el orientado a objetos. El problema de las bases de datos orientadas a objetos es que requieren reinvertir capital y esfuerzos de nuevo para convertir las bases de datos relacionales en bases de datos orientadas a objetos. En las bases de datos objeto relacionales se intenta conseguir una compatibilidad relacional dando la posibilidad de integrar mejoras de la orientación a objetos.

Estas bases de datos se basan en el estándar **SQL 99**. En ese estándar se añade a las bases relacionales la posibilidad de almacenar procedimientos de usuario, triggers, tipos definidos por el usuario, consultas recursivas, bases de datos OLAP, tipos LOB,...

Las últimas versiones de la mayoría de las clásicas grandes bases de datos relacionales (**Oracle**, **SQL Server**, **Informix**, ...) son objeto relacionales.

(1.5) diseño conceptual de bases de datos. el modelo entidad - relación

(1.5.1) introducción

Ya hemos visto anteriormente que existen varios esquemas a realizar para poder representar en forma de base de datos informática un problema procedente del ordenador.

El primero de esos esquemas es el llamado **esquema conceptual**, que representa la información de forma absolutamente independiente al Sistema Gestor de Base de Datos. Los esquemas internos de las diferentes bases de datos no captan suficientemente bien la semántica del mundo real, de ahí que primero haya que pasar por uno o dos esquemas previos más cercanos al mundo real.

El hecho de saltarse el esquema conceptual conlleva un problema de pérdida con el problema real. El esquema conceptual debe reflejar todos los aspectos relevantes del mundo a real a modelar.

Peter P. Chen y el modelo entidad/relación

En 1976 y 1977 dos artículos de **Peter P. Chen** presentan un modelo para realizar esquemas que posean una visión unificada de los datos. Este modelo es el modelo entidad/interrelación (**entity/relationship** en inglés) que actualmente se conoce más con el nombre de entidad/relación (**Modelo E/R** o **ME/R**, en inglés **E/RM**).

Posteriormente otros autores han añadido mejoras a este modelo lo que ha producido una familia de modelos. La más aceptada actualmente es el modelo entidad/relación extendido (ERE) que complementa algunas carencias del modelo

original. No obstante las diversas variantes del modelo hacen que la representación de este modelo no sea muy estándar, aunque hay ideas muy comunes a todas las variantes.

Hay que insistir en que este modelo no tiene nada que ver con las bases de datos relacionales, los esquemas entidad/relación se pueden utilizar con cualquier SGBD ya que son conceptuales. Confunde el uso de la palabra **relación**, pero el concepto de relación en este esquema no tiene nada que ver con la idea de relación expuesta por **Codd** en su modelo relacional.

(1.5.2) componentes del modelo

entidad

Se trata de cualquier objeto u elemento (real o abstracto) acerca del cual se pueda almacenar información en la base de datos. Es decir cualquier elemento informativo que tenga importancia para una base de datos.

Ejemplos de entidades son Pedro, la factura número 32456, el coche matrícula 3452BCW, etc. Una entidad no es un propiedad concreta sino un objeto que puede poseer múltiples propiedades (atributos). Es decir “Sánchez” es el contenido del atributo **Primer Apellido** de la entidad que representa a la persona Pedro Sánchez Crespo con DNI 12766374,...

Una entidad es un objeto concreto, no un simple dato: el coche que tenemos en el garaje es una entidad, “Mercedes” sin embargo es la marca de ese coche, es decir es un atributo de esa entidad.

conjuntos de entidades

Las entidades que poseen las mismas propiedades forman conjuntos de entidades. Ejemplos de conjuntos de entidades son los conjuntos: personas, facturas, coches,...

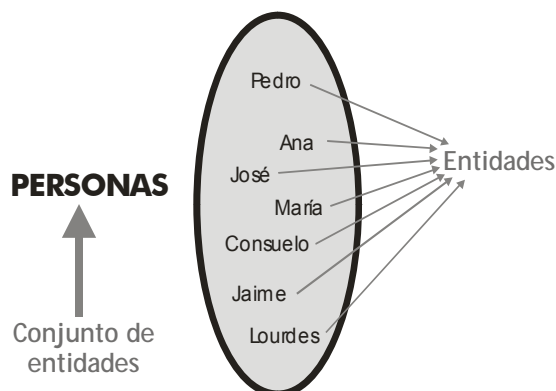


Ilustración 10, Ejemplos de entidad y conjunto de entidad

En la actualidad se suele llamar **entidad** a lo que anteriormente se ha definido como conjunto de entidades. De este modo hablaríamos de la entidad PERSONAS. Mientras que cada persona en concreto sería una **ocurrencia** o un **ejemplar** de la entidad **persona**.

Esa terminología es la que actualmente vamos a utilizar en este manual.

representación gráfica de las entidades

En el modelo entidad relación los conjuntos de entidades se representan con un rectángulo dentro del cual se escribe el nombre de la entidad:



Ilustración 11, Representación de la entidad persona

tipos de entidades

- ◆ **Regulares.** Son las entidades normales que tienen existencia por sí mismas sin depender de otras. Su representación gráfica es la indicada arriba
- ◆ **Débiles.** Su existencia depende de otras. Por ejemplo la entidad **tarea laboral** sólo podrá tener existencia si existe la entidad **trabajo**. Las entidades débiles se presentan de esta forma:



Ilustración 12, Entidad débil

(1.5.3) relaciones

qué es una relación

Representan **asociaciones** entre entidades. Es el elemento del modelo que permite relacionar en sí los datos del mismo. Por ejemplo, en el caso de que tengamos una entidad personas y otra entidad trabajos. Ambas se realizan ya que las personas trabajan y los trabajos son realizados por personas:

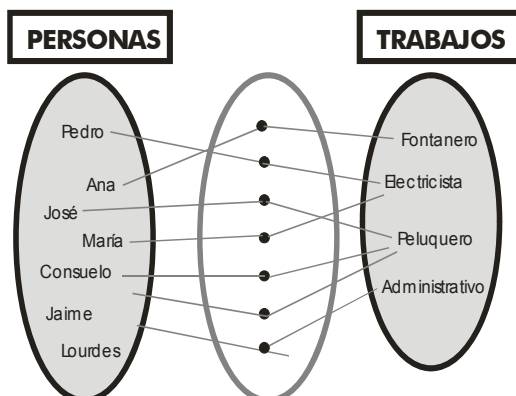


Ilustración 13, ejemplo de relación

En una relación (Chen llamaba conjunto de relaciones a lo que ahora se llama relación a secas) cada ejemplar (relación en la terminología de Chen) asocia un elemento de una entidad con otro de la otra entidad. **En una relación no pueden aparecer dos veces relacionados los mismos ejemplares.** Es decir en el ejemplo anterior, en la relación no puede aparecer dos veces el mismo trabajador asociado al mismo trabajo.

representación gráfica

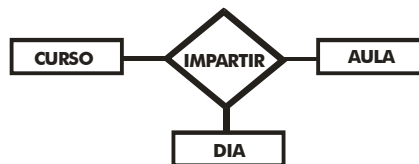
La representación gráfica de las entidades se realiza con un rombo al que se le unen líneas que se dirigen a las entidades, las relaciones tienen nombre (se suele usar un verbo). En el ejemplo anterior podría usarse como nombre de relación, trabajar:



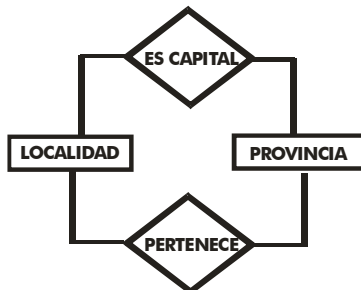
ejemplos de relaciones



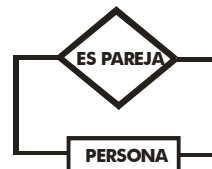
Relación binaria



Relación ternaria



Relación doble



Relación reflexiva

- ◆ **Relaciones Binarias.** Son las relaciones típicas. Se trata de relaciones que asocian dos entidades.
- ◆ **Relaciones Ternarias.** Relacionan tres entidades. A veces se pueden simplificar en relaciones binarias, pero no siempre es posible.
- ◆ **Relaciones n -arias.** Relacionan n entidades
- ◆ **Relaciones dobles.** Se llaman así a dos relaciones distintas que sirven para relacionar a las mismas relaciones. Son las más difíciles de manejar ya que al manipular las entidades hay que elegir muy bien la relacionan a utilizar para relacionar los datos.
- ◆ **Relación reflexiva.** Es una relación que sirve para relacionar ejemplares de la misma entidad (personas con personas, piezas con piezas, etc.)

cardinalidad

Indica el número de relaciones en las que una entidad puede aparecer. Se anota en términos de:

- ◆ **cardinalidad mínima.** Indica el número mínimo de asociaciones en las que aparecerá cada ejemplar de la entidad (el valor que se anota es de cero o uno, aunque tenga una cardinalidad mínima de más de uno, se indica sólo un uno)
- ◆ **cardinalidad máxima.** Indica el número máximo de relaciones en las que puede aparecer cada ejemplar de la entidad. Puede ser uno, otro valor concreto mayor que uno (tres por ejemplo) o muchos (se representa con *n*)

En los esquemas entidad / relación la cardinalidad se puede indicar de muchas formas. Quizá la más completa (y la que se utiliza en este documento es ésta) consiste en anotar en los extremos la cardinalidad máxima y mínima de cada entidad en la relación.

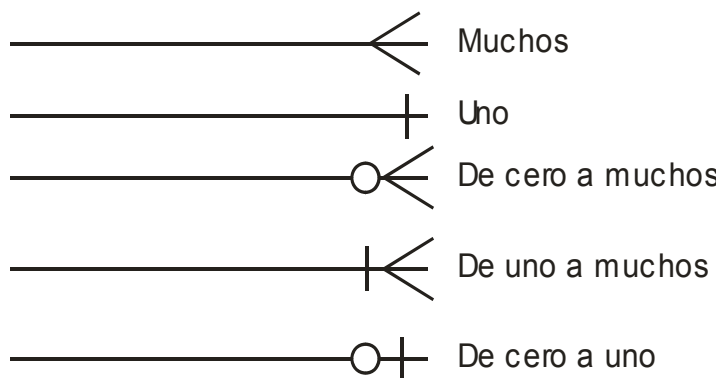
Ejemplo de uso de cardinalidad:



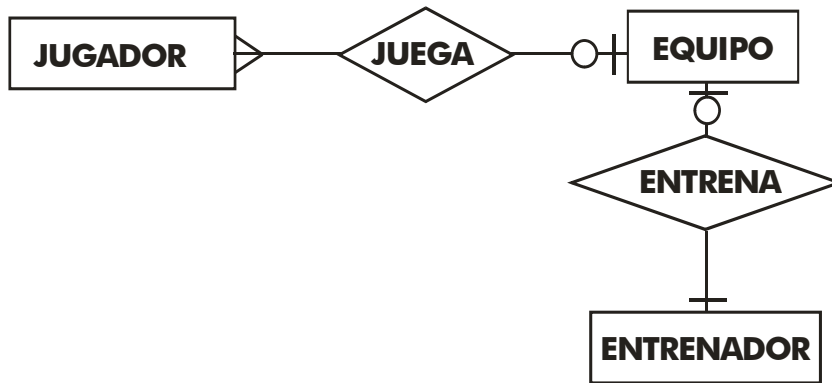
Ilustración 14, Cardinalidades.

En el ejemplo un jugador tiene una cardinalidad mínima de 0 (puede no estar en ningún equipo) y una máxima de 1 (como mucho está en un equipo, no puede estar en dos a la vez). Cada equipo tiene una cardinalidad mínima de uno (en realidad sería una cardinalidad mínima más alta, pero se anota un uno) y una máxima de *n* (en cada equipo hay muchos jugadores)

En la página siguiente se indican otras notaciones para las cardinalidades.

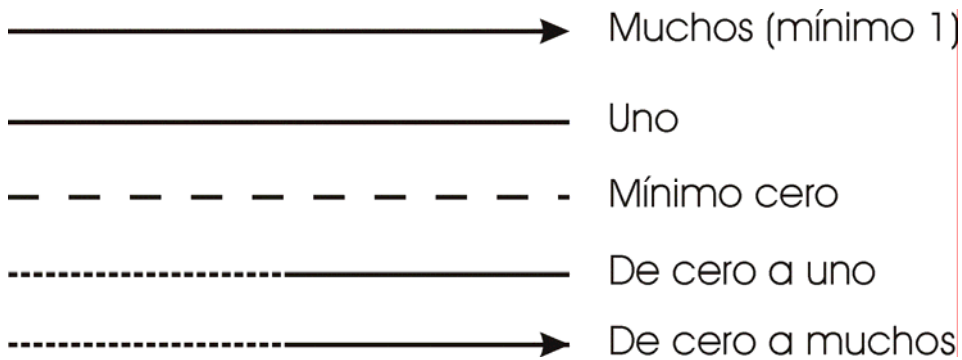


Ejemplo:



En el ejemplo, cada equipo cuenta con varios jugadores. Un jugador juega como mucho en un equipo y podría no jugar en ninguno. Cada entrenador entrena a un equipo (podría no entrenar a ninguno), el cual tiene un solo entrenador como mucho y como poco.

Otra notación es:



Y aún habría más pero nos quedaremos con la primera ya que es la más completa.

roles

A veces en las líneas de la relación se indican **roles**. Los roles representan el papel que juega una entidad en una determinada relación.

Ejemplo:

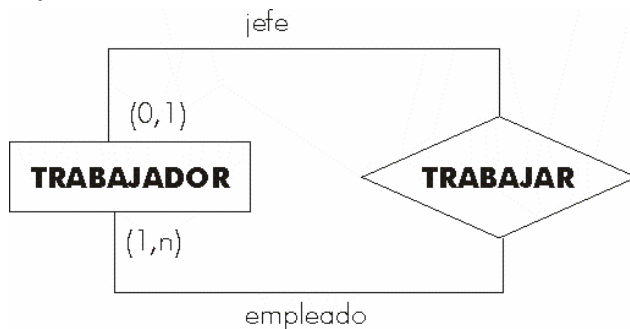


Ilustración 15, Ejemplo de rol. Un trabajador puede ser visto como jefe o como empleado según a qué lado de la relación esté

(1.5.4) atributos

Describen propiedades de las entidades y las relaciones. En este modelo se representan con un círculo, dentro del cual se coloca el nombre del atributo. Ejemplo:

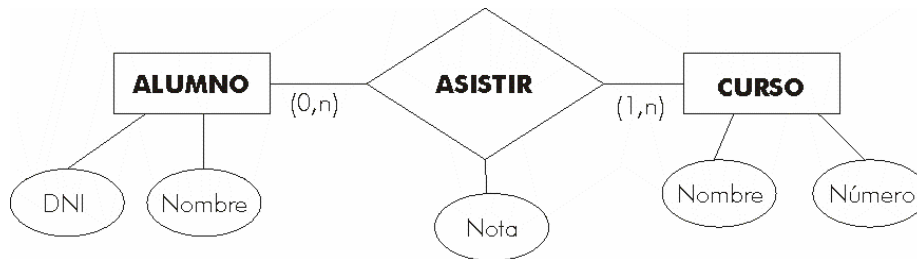
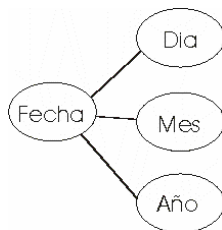


Ilustración 16, Atributos

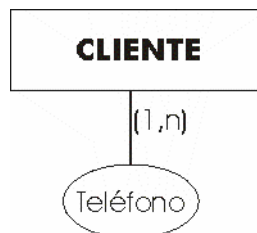
tipos de atributos

compuesto



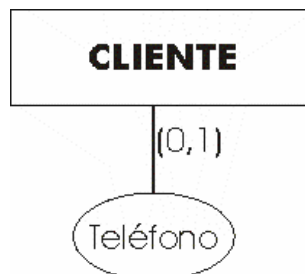
múltiples

Pueden tomar varios valores (varios teléfonos para el mismo cliente):



opcionales

Lo son si pueden tener valor nulo:



identificador o clave

Se trata de uno o más campos cuyos valores son únicos en cada ejemplar de una entidad. Se indican subrayando el nombre del identificador.

Para que un atributo sea considerado un buen identificador tiene que cumplir con los siguientes requisitos:

- (1) Deben distinguir a cada ejemplar teniendo en cuenta las entidades que utiliza el modelo. No tiene que ser un identificador absoluto.
- (2) Todos los ejemplares de una entidad deben tener el mismo identificador.
- (3) Cuando un atributo es importante aun cuando no tenga una entidad concreta asociada, entonces se trata de una entidad y no de un atributo

identificador alternativo

Se trata de uno o más campos cuyos valores son únicos para cada ejemplar de una entidad, pero que no son identificadores ya que existen identificadores mejores en la entidad. En este caso los candidatos es aconsejable marcarlos con un subrayado discontinuo (ejemplo de subrayado discontinuo)

(1.5.5) modelo entidad relación extendido

En el modelo entidad relación extendido aparecen nuevos tipos de relaciones. Son las **relaciones ISA (es un)** y las **entidades débiles**

relaciones is a o relaciones de herencia

Se utilizan para unificar entidades agrupándolas en una entidad más general (**generalización**) o bien para dividir una entidad general en entidades más específicas (**especificación**).

Se habla de generalización si inicialmente partimos de una serie de entidades que al estudiarlas en detalle descubrimos que todas ellas pertenecen al mismo conjunto. En la generalización las entidades son totalmente heterogéneas, es decir, los atributos son diferentes. La entidad general se llama **superentidad** las otras se denominan **subentidades**. La superentidad normalmente tiene una clave principal distinta de las subentidades.

La especialización ocurre cuando partimos de una entidad que podemos dividir en subentidades para detallar atributos que varían en las mismas. Comparten clave con la superentidad y los atributos de la superclase se heredan en las subclases.

En la práctica se manejan casi igual ambas; de hecho la representación es la misma:

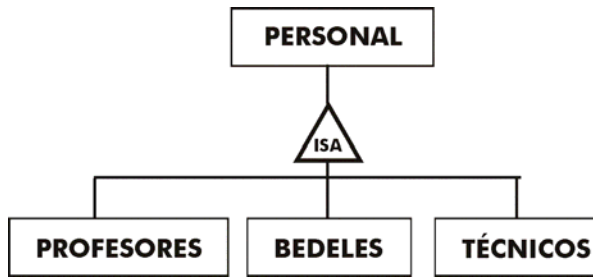


Ilustración 17, Relación ISA

La entidad general personal se ha dividido en tres pequeñas entidades. La cuestión de si es generalización o especialización no suele ser excesivamente importante salvo por el tema de la herencia de los atributos; hay que recordar que en la especialización, las subentidades heredan los atributos de la superentidad.

En el diseño la herencia de los atributos queda reflejado por las cardinalidades, si en la superentidad se indica una cardinalidad mínima de uno, se entiende entonces que se trata de de una especialización en las que las subentidades heredan los atributos de la superclase. Si la cardinalidad mínima es cero se entiende que las subclases no heredan los atributos.

Como se comentó antes la cuestión de si es una especialización o generalización se suele distinguir por las claves; si se comparte clave entre la superentidad y sus descendientes, se habla de especialización; de otro modo se habla de generalización (aunque esto es muy rebatible, en la práctica suele ser la única forma de distinguir ambos conceptos en el esquema).

De cualquier modo, la cuestión de si tenemos una generalización o una especialización no es tan importante como el hecho de no fallar con las cardinalidades, ya que al pasar el esquema al modelo relacional es lo que importa más.

La representación de relaciones ISA (independientemente de si es generalización o especialización) es esta:

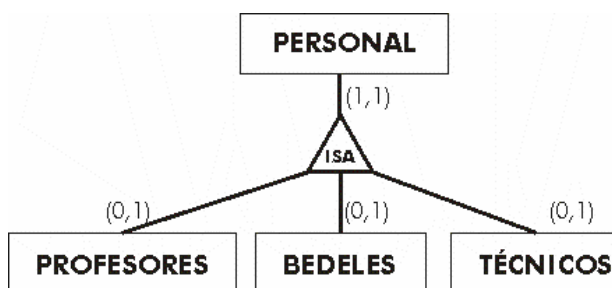


Ilustración 18, Relación ISA con cardinalidades

Con atributos el esquema sería:

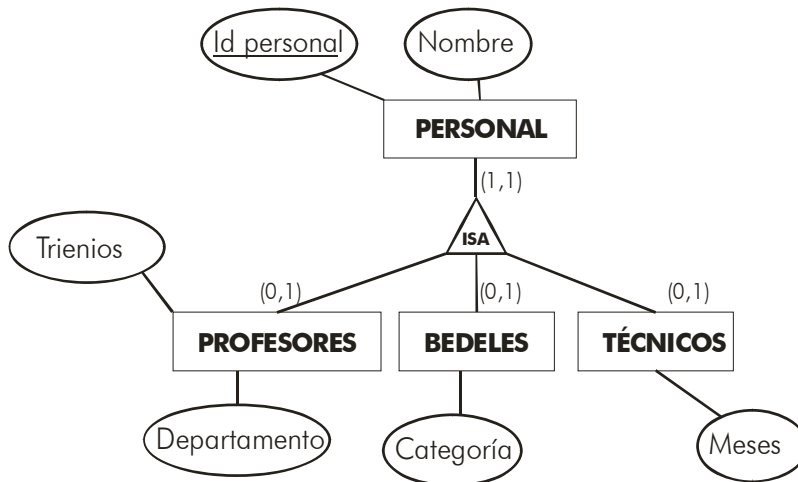


Ilustración 19, Especialización

En la especialización anterior (lo es porque la clave la tiene la superentidad) los profesores, bedeles y técnicos heredan el atributo **id personal** y el **nombre**, el resto son atributos propios sólo de cada entidad (**trienios** pertenece sólo a los profesores, en este ejemplo)

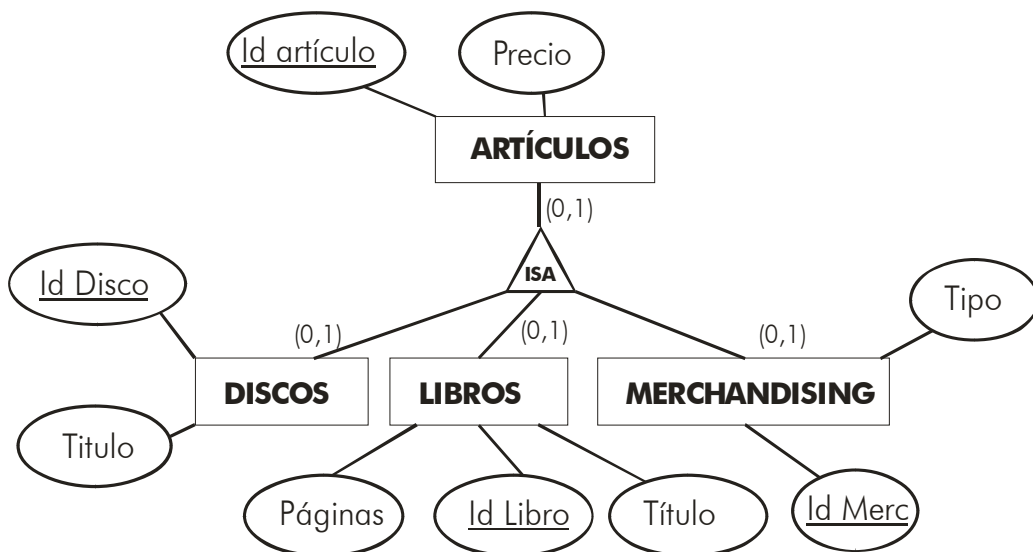


Ilustración 20, Generalización

En la ilustración anterior artículo es una generalización de los discos, libros y artículos de merchandising, se utiliza una clave distinta para esta entidad. Incluso en este caso podría haber discos o libros o merchandising que no están relacionados con los artículos (la cardinalidad de artículos es 0,1).

obligatoriedad

En las relaciones ISA (y también en otros tipos de relaciones) se puede indicar el hecho de que cada ejemplar obligatoriamente tiene que participar en una de entre

varias ramas de una relación. Este hecho se marca con un arco entre las distintas relaciones. En las relaciones ISA se usa mucho, por ejemplo:

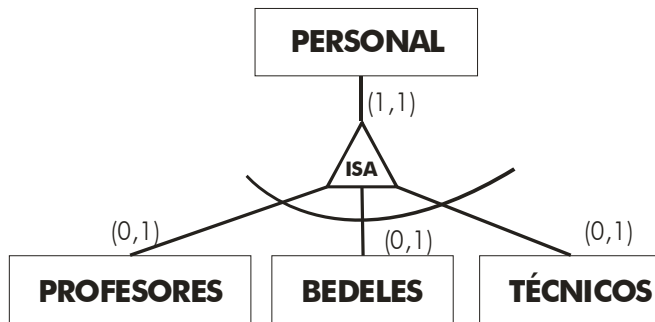


Ilustración 21, Relación ISA con obligatoriedad

En el ejemplo, el personal tiene que ser obligatoriamente un bedel, o un profesor o un técnico; una de las tres cosas (no puede haber personal que no sea una de estas tres cosas).

entidades débiles

Ya se ha comentado antes que una entidad débil es aquella cuya existencia depende de otra. Ahora vamos a clarificar más estas entidades. Efectivamente ocurren cuando hay una entidad más fuerte de la que dependen. Lógicamente tienen relación con esa entidad. En la forma clásica se representaría de esta forma:

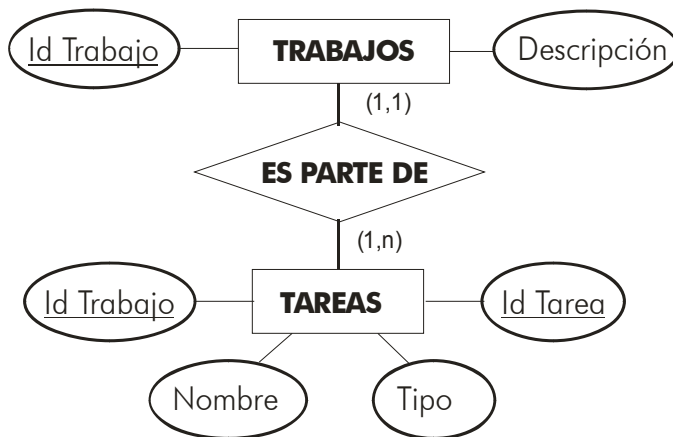


Ilustración 22, Relación candidata a entidad débil

En el diagrama la relación entre las tareas y los trabajos es 1 a n (cada trabajo se compone de n tareas). Una tarea obligatoriamente está asignada a un trabajo, es más no tiene sentido hablar de tareas sin hablar del trabajo del que forma parte.

Hay incluso (aunque no siempre) una **dependencia de identificación** ya que las tareas se identifican por un número de tarea y el número de trabajo al que se asignan. Esto es un síntoma definitivo de que se trata de una entidad débil.

Todas las entidades débiles tienen este tipo de relación 1 a n con respecto a la entidad fuerte de la que depende su existencia, por eso se representan de esta otra forma:

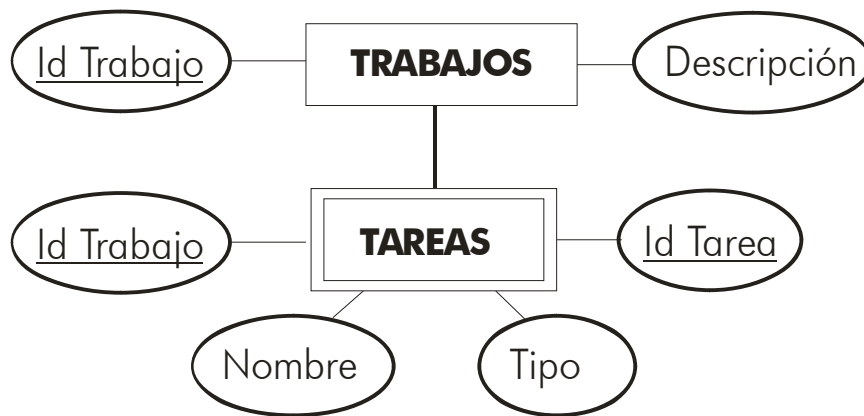


Ilustración 23, Entidad débil relacionada con su entidad fuerte

No hace falta dibujar el rombo de la relación ni la cardinalidad, se sobreentiende el tipo y cardinalidad (1 a n) que posee.

(Unidad 2)

Bases de Datos Relacionales

(2.1) el modelo relacional

(2.1.1) introducción

Edgar Frank Codd a finales definió las bases del modelo relacional a finales de los 60. En 1970 publica el documento *“A Relational Model of data for Large Shared Data Banks”* (*“Un modelo relacional de datos para grandes bancos de datos compartidos”*). Actualmente se considera que ese es uno de los documentos más influyentes de toda la historia de la informática.

Anteriormente el único modelo teórico estandarizado era el **CodasyI** que se utilizó masivamente en los años 70. Codd introduce el término relación (en inglés *relationship*, a veces traducido como interrelación) que es el que aglutina los datos de forma independiente a lo que será su almacenamiento físico.

Lo que Codd intenta precisamente es que este modelo oculte completamente conceptos y términos de la computadora en sí, es decir se abstrae más que los modelos anteriores.

Trabajaba para **IBM**, empresa que tardó un poco en implementar sus bases. De hecho fueron otras empresas (en especial **Oracle**) las que implementaron sus teorías. Pocos años después el modelo se empezó a utilizar cada vez más, hasta finalmente ser el modelo de bases de datos más popular. Hoy en día casi todas las bases de datos siguen este modelo.

(2.1.2) objetivos

Codd perseguía estos objetivos con su modelo:

- ◆ **Independencia física.** La forma de almacenar los datos, no debe influir en su manipulación lógica. Si el almacenamiento físico cambia, los usuarios no tienen ni siquiera porque enterarse, seguirán funcionando sus aplicaciones.

Sistemas Gestores de Bases de Datos

(Unidad 2) Bases de Datos Relacionales

- ◆ **Independencia lógica.** Las aplicaciones que utilizan la base de datos no deben ser modificadas por que se modifiquen elementos de la base de datos. Es decir, añadir, borrar y suprimir datos, no influye en las vistas de los usuarios.
- ◆ **Flexibilidad.** La base de datos ofrece fácilmente distintas vistas en función de los usuarios y aplicaciones.
- ◆ **Uniformidad.** Las estructuras lógicas siempre tienen una única forma conceptual (las tablas)
- ◆ **Sencillez.**

(2.1.3) historia del modelo relacional

Año	Hecho
1970	Codd publica las bases del modelo relacional
1971-72	Primeros desarrollos teóricos
1973-78	Primeros prototipos (System R de IBM)
1978	Aparece el lenguaje QBE (Query By Example) lenguaje de acceso relacional a los archivos VSAM de IBM
1979	Aparece Oracle, el primer SGBD relacional. Codd revisa su modelo relacional y lanza el modelo RM/T como un intento de subsanar sus deficiencias.
1980	Aparece Ingres, SGBD relacional basado en cálculo relacional. Utilizaba el lenguaje Quel que era el favorito para las universidades.
1981	Aparece SQL que se convertirá en el estándar en la creación de bases de datos relacionales
1982	Aparece DB2 , el sistema gestor de bases de datos relacionales de IBM
1986	ANSI normaliza el SQL (SQL/ANSI)
1987	ISO también normaliza SQL. Es el SQL ISO(9075)
1989	ISO revisa el estándar y publica el estándar SQL Addendum
1990	Versión dos del modelo relacional (RM/V2) realizada por Codd. Propuesta de Stonebraker para añadir al modelo relacional capacidades de orientación a objetos.
1992	ISO publica el estándar SQL 92 (todavía el más utilizado)
1995	Manifiesto de Darwen y Date en el que animan a reinterpretar el modelo relacional desde una perspectiva de objetos.
1999	ISO publica un nuevo estándar que incluye características más avanzadas. Se llama SQL 99

(2.1.4) las reglas de Codd

Codd escribió doce reglas (muy famosas) que debían cumplir todas las bases de datos relacionales. Se explican con detalle un poco más adelante (cuando se conozcan mejor los términos del tema)

(2.2) estructura de las bases de datos relacionales

(2.2.1) relación o tabla

Según el modelo relacional (desde que Codd lo enunció) el elemento fundamental es lo que se conoce como relación, aunque más habitualmente se le llama **tabla** (o también array o matriz). Codd definió las relaciones utilizando un lenguaje matemático, pero se pueden asociar a la idea de tabla (de filas y columnas) ya que es más fácil de entender.

Las relaciones constan de:

- ◆ **Atributos.** Referido a cada dato que se almacena en la relación (nombre, dni,...).
- ◆ **Tuplas.** Referido a cada elemento de la relación. Por ejemplo si una relación almacena personas, una tupla representaría a una persona en concreto.

Puesto que una relación se representa como una tabla; podemos entender que las columnas de la tabla son los atributos; y las filas, las tuplas.

atributo 1	atributo 2	atributo 3	atributo n	
valor 1,1	valor 1,2	valor 1,3	valor 1,n	← tupla 1
valor 2,1	valor 2,2	valor 2,3	valor 2,n	← tupla 2
.....
valor m,1	valor m,2	valor m,3	valor m,n	← tupla m

La tabla superior representa la estructura de una relación según el modelo de Codd.

(2.2.2) tupla

Cada una de las filas de la tabla. Se corresponde con la idea de **registro**. Representa por tanto cada elemento individual de esa relación. Tiene que cumplir que:

- ◆ Cada tupla se debe corresponder con un elemento del mundo real.
- ◆ No puede haber dos tuplas iguales (con todos los valores iguales).

(2.2.3) dominio

Un dominio contiene todos los posibles valores que puede tomar un determinado atributo. Dos atributos distintos pueden tener el mismo dominio.

Un dominio en realidad es un conjunto finito de valores del mismo tipo. A los dominios se les asigna un nombre y así podemos referirnos a ese nombre en más de un atributo.

La forma de indicar el contenido de un dominio se puede hacer utilizando dos posibles técnicas:

- ◆ **Intensión.** Se define el dominio indicando la definición exacta de sus posibles valores. Por intención se puede definir el dominio de edades de los trabajadores como: **números enteros entre el 16 y el 65** (un trabajador sólo podría tener una edad entre 16 y 65 años).
- ◆ **Extensión.** Se indican algunos valores y se sobreentiende el resto gracias a que se autodefinen con los anteriores. Por ejemplo el dominio localidad se podría definir por extensión así: **Palencia, Valladolid, Villamuriel de Cerrato,...**

Además pueden ser:

- ◆ **Generales.** Los valores están comprendidos entre un máximo y un mínimo
- ◆ **Restringidos.** Sólo pueden tomar un conjunto de valores

(2.2.4) grado

Indica el tamaño de una relación en base al número de columnas (atributos) de la misma. Lógicamente cuanto mayor es el grado de una relación, mayor es su complejidad al manejarla.

(2.2.5) cardinalidad

Número de tuplas de una relación, o número de filas de una tabla.

(2.2.6) sinónimos

Los términos vistos anteriormente tienen distintos sinónimos según la nomenclatura utilizada. A ese respecto se utilizan tres nomenclaturas:

Términos 1 (nomenclatura relacional)		Términos 2 (nomenclatura tabla)		Términos 3 (nomenclatura ficheros)
relación	=	tabla	=	fichero
tupla	=	fila	=	registro
atributo	=	columna	=	campo
grado	=	nº de columnas	=	nº de campos
cardinalidad	=	nº de filas	=	nº de registros

(2.2.7) definición formal de relación

Una relación está formada por estos elementos:

- ◆ **Nombre.** Identifica la relación.
- ◆ **Cabecera de relación.** Conjunto de todos los pares atributo-domino de la relación:
- ◆ $\{(A_i:D_i)\}_{i=1}^n$ donde n es el **grado**.
- ◆ **Cuerpo de la relación.** Representa el conjunto de **m** tuplas $\{t_1, t_2, \dots, t_n\}$ que forman la relación. Cada tupla es un conjunto de **n** pares atributo-valor $\{(A_i:V_{ij})\}$, donde V_{ij} es el valor j del dominio D_i asociado al atributo A_i .
- ◆ **Esquema de la relación.** Se forma con el nombre R y la cabecera. Es decir:
- ◆ $R\{(A_i:D_i)\}_{i=1}^n$
- ◆ **Estado de la relación.** Lo forman el esquema y el cuerpo.

Ejemplo:

Cliente

DNI	Nombre	Edad
12333944C	Ana	52
12374678G	Eva	27
28238232H	Martín	33

Esquema: Cliente(DNI:DNI, Nombre:Nombre, Edad:Edad)

Cuerpo: $\{(DNI: "12333944C", Nombre:"Ana", Edad:52), (DNI: "12374678G", Nombre:"Eva", Edad;52), (DNI: "28238232H", Nombre:"Martín",Edad:33)\}$

(2.2.8) propiedades de las tablas (o relaciones)

- ◆ Cada tabla tiene un nombre distinto
- ◆ Cada atributo de la tabla toma un solo valor en cada tupla
- ◆ Cada atributo tiene un nombre distinto en cada tabla (puede coincidir en tablas distintas)
- ◆ Cada tupla es única (no hay tuplas duplicadas)
- ◆ El orden de los atributos no importa
- ◆ El orden de las tuplas no importa

(2.2.9) tipos de tablas

- ◆ **Persistentes.** Sólo pueden ser borradas por los usuarios:
 - **Base.** Independientes, se crean indicando su estructura y sus ejemplares.
 - **Vistas.** Son tablas que sólo almacenan una definición de consulta, resultado de la cual se produce una tabla cuyos datos proceden de las bases o de otras vistas e instantáneas. Si los datos de las tablas base cambian, los de la vista que utiliza esos datos también cambia.
 - **Instantáneas.** Son vistas (creadas de la misma forma) que sí que almacenan los datos que muestra, además de la consulta que dio lugar a esa vista. Sólo modifican su resultado (actualizan los datos) siendo refrescadas por el sistema cada cierto tiempo.
- ◆ **Temporales.** Son tablas que se eliminan automáticamente por el sistema. Pueden ser de cualquiera de los tipos anteriores. Las utiliza el SGBD como almacén intermedio de datos (resultados de consultas, por ejemplo)

(2.2.10) claves

clave candidata

Conjunto de atributos que identifican unívocamente cada tupla de la relación. Es decir columnas cuyos valores no se repiten en ninguna otra tupla de esa tabla.

clave primaria

Clave candidata que se escoge como **identificador** de las tuplas. Se elige como primaria la candidata que identifique mejor a cada tupla en el contexto de la base de datos.

Por ejemplo un campo con el DNI sería clave candidata de una tabla de clientes, si esa relación tiene un campo de código de cliente, éste sería mejor candidato (y por lo tanto clave principal) porque es mejor identificador para ese contexto.

clave alternativa

Cualquier clave candidata que no sea primaria.

clave externa, ajena o secundaria

Son los datos de atributos de una tabla cuyos valores están relacionados con atributos de otra tabla. Por ejemplo en la tabla equipos tenemos estos datos:

Equipo	Nº Equipo
Real Madrid	1
F.C. Barcelona	2
Athletic Bilbao	3

En la tabla anterior la clave principal es el atributo **nº equipo**. En otra tabla tenemos:

Nº Jugador	Jugador	Nº Equipo
1	Karanka	3
2	Ronaldinho	2
3	Raul	1
4	Beckham	1

El atributo **Nº Equipo** sirve para relacionar el Jugador con el equipo al que pertenece. Ese campo en la tabla de jugadores es una clave secundaria.

(2.2.11) nulos

En los lenguajes de programación se utiliza el valor nulo para reflejar que un identificador no tiene ningún contenido. Al programar en esos lenguajes se trata de un valor que no permite utilizarse en operaciones aritméticas o lógicas.

En las bases de datos relacionales se utiliza con más posibilidades, aunque su significado no cambia: valor vacío, se utiliza para diversos fines.

En claves secundarias indican que el registro actual no está relacionado con ninguno. En otros atributos indica que no se puede rellenar ese valor por la razón que sea.

Es importante indicar que el texto vacío "", no es lo mismo que el nulo. Como tampoco el valor cero significa nulo.

Puesto que ese valor se utiliza continuamente, resulta imprescindible saber como actúa cuando se emplean operaciones lógicas sobre ese valor. Eso significa definir un tercer valor en la lógica booleana, además de los clásicos **verdadero** y **falso**. Un valor nulo no es ni verdadero ni falso (se suele interpretar como un **quizás**). El uso de operadores lógicos con el nulo da lugar a que:

- ◆ **verdadero Y (AND) nulo** da como resultado, nulo
- ◆ **falso Y (AND) nulo** da como resultado, falso
- ◆ **verdadero O (OR) nulo** da como resultado, verdadero
- ◆ **falso O nulo** da como resultado nulo
- ◆ **la negación de nulo**, da como resultado nulo

Se utiliza un operador en todas las bases relacionales llamado **es_nulo** (*is null*) que devuelve verdadero si el valor con el que se compara es nulo.

(2.3) restricciones

Se trata de unas condiciones de obligado cumplimiento por los datos de la base de datos. Las hay de varios tipos.

(2.3.1) inherentes

Son aquellas que no son determinadas por los usuarios, sino que son definidas por el hecho de que la base de datos sea relacional. Las más importantes son:

- ◆ No puede haber dos tuplas iguales
- ◆ El orden de las tuplas no es significativo
- ◆ El orden de los atributos no es significativo
- ◆ Cada atributo sólo puede tomar un valor en el dominio en el que está inscrito

(2.3.2) semánticas

El modelo relacional permite al usuario incorporar restricciones personales a los datos. Se comentan las diferentes reglas semánticas a continuación:

clave primaria (*primary key*)

Hace que los atributos marcados como clave primaria no puedan repetir valores. Además obliga a que esos atributos no puedan estar vacíos (nulos); es más si la clave primaria la forman varios atributos, ninguno de ellos podrá estar vacío.

unicidad (*unique*)

Impide que los valores de los atributos marcados de esa forma, puedan repetirse. Esta restricción debe indicarse en todas las claves alternativas.

Al marcar una clave primaria se añade automáticamente sobre los atributos que forman la clave un criterio de unicidad.

obligatoriedad (*not null*)

Prohíbe que el atributo marcado de esta forma no tenga ningún valor (es decir impide que pueda contener el valor nulo, *null*).

integridad referencial (*foreign key*)

Sirve para indicar una clave externa. Cuando esa clave se marca con integridad referencial, no se podrán introducir valores que no estén incluidos en los campos relaciones con esa clave.

Es decir si hay una tabla de alquileres en la que cada fila es un alquiler, existirá un atributo *cod_cliente* que indicará el código del cliente y que estará relacionado con una tabla de clientes, en la que dicho atributo es la clave principal. De hecho no se podrá incluir un código que no esté en la tabla clientes; eso es lo que prohíbe la integridad referencial.

Eso causa problemas en las operaciones de borrado y modificación de registros; ya que si se ejecutan esas operaciones sobre la tabla principal (si se modifica o borra un cliente) quedarán filas en la tabla secundaria con la clave externa haciendo referencia a un valor que ya no existe.

Esto último se puede manipular de estas formas:

- ◆ **Prohibiendo la operación (*no action*)**.
- ◆ **Transmitiendo la operación en cascada (*cascade*)**. Es decir si se modifica o borra un cliente; también se modificarán o barrarán los alquileres relacionados con él.
- ◆ **Colocando nulos (*set null*)** Las referencias al cliente en la tabla de alquileres se colocan como nulos (es decir, alquileres sin cliente).
- ◆ **Usando el valor por defecto (*default*)**. Se colocan un valor por defecto en las claves externas relacionadas.

regla de validación (*check*)

Condición que debe de cumplir un dato concreto para que sea actualizado. Puede afectar a una tabla o a varias. Por ejemplo restringir el campo sueldo para que siempre sea mayor de 1000, sería una regla de validación.

disparadores o triggers

Se trata de instrucciones creadas en un lenguaje estructurado incluido en la base de datos, que sirve para realizar una serie de acciones cuando ocurre un determinado evento a la base de datos (cuando se añade una tupla, cuando se borra un dato,...)

Permite realizar restricciones muy potentes, pero son las más difíciles de realizar. En el caso de Oracle se realizan mediante el lenguaje PL/SQL

(2.4) las 12 reglas de Codd

Preocupado por los productos que decían ser sistemas gestores de bases de datos relacionales (**RDBMS**) sin serlo, Codd publica las 12 reglas que debe cumplir todo DBMS para ser considerado relacional. Estas reglas en la práctica las cumplen pocos sistemas relacionales. Las reglas son:

- (1) Información.** Toda la información de la base de datos (**metadatos**) debe estar representada explícitamente en el esquema lógico. Es decir, **todos** los datos están en las tablas.
- (2) Acceso garantizado.** Todo **dato** es accesible sabiendo el valor de su clave y el nombre de la columna o atributo que contiene el dato.
- (3) Tratamiento sistemático de los valores nulos.** El DBMS debe permitir el tratamiento adecuado de estos valores. De ese modo el valor nulo se utiliza para representar la ausencia de información de un determinado registro en un atributo concreto.
- (4) Catálogo en línea basado en el modelo relacional.** Los metadatos deben de ser accesibles usando un esquema relacional. Es decir la forma de acceder a los metadatos es la misma que la de acceder a los datos.

- (5) **Sublenguaje de datos completo.** Al menos debe de existir un lenguaje que permita el manejo completo de la base de datos. Este lenguaje, por lo tanto, debe permitir realizar cualquier operación sobre la misma.
- (6) **Actualización de vistas.** El DBMS debe encargarse de que las vistas muestren la última información.
- (7) **Inserciones, modificaciones y eliminaciones de dato nivel.** Cualquier operación de modificación debe actuar sobre conjuntos de filas o registros, nunca deben actuar registro a registro.
- (8) **Independencia física.** Los datos deben de ser accesibles desde la lógica de la base de datos aún cuando se modifique el almacenamiento. La forma de acceder a los datos no varía porque el esquema físico de la base de datos, cambia.
- (9) **Independencia lógica.** Los programas no deben verse afectados por cambios en las tablas. Que las tablas cambien no implica que cambien los programas.
- (10) **Independencia de integridad.** Las reglas de integridad deben almacenarse en la base de datos (en el diccionario de datos), no en los programas de aplicación.
- (11) **Independencia de la distribución.** El sublenguaje de datos debe permitir que sus instrucciones funciones igualmente en una base de datos distribuida que en una que no lo es.
- (12) **No subversión.** Si el DBMS posee un lenguaje que permite el recorrido fila a fila, éste no puede utilizarse para incumplir o evitar las reglas relacionales establecidas por el sublenguaje de datos.

(2.5) álgebra relacional

(2.5.1) dinámica del modelo relacional

Lo visto anteriormente pertenece al campo de la **estática** de las bases de datos relacionales. La **dinámica** en cambio dicta las transformaciones que se pueden realizar sobre la base de datos. La estática se refiere a las tablas base, la dinámica a las vistas.

(2.5.2) lenguajes formales

Las bases de datos relacionales sólo contienen tablas, por lo que la dinámica manipula esas tablas. Para indicar la transformación se emplean lenguajes formales, estos lenguajes manipulan los datos sin tener en cuenta cómo se almacenan

Codd definió dos lenguajes formales

- ◆ **Álgebra relacional.** Operaciones que combinan tablas y que dan como resultado otra tabla.
- ◆ **Cálculo relacional.** Operaciones que a partir de una fórmula obtienen un conjunto de tuplas.

Se supone que ambos son equivalentes.

En este apartado se explica el álgebra relacional.

(2.5.3) operaciones básicas unitarias

renombrado

La operación de renombrado se utiliza exclusivamente para cambiar el nombre de una relación. Se utiliza de esta forma:

$$R \rightarrow R'$$

La relación R pasa a llamarse R'

selección (σ)

Es una operación que aplicada a una tabla obtiene un subconjunto de filas de esa tabla en la que sólo aparecen las filas que cumplan un determinado criterio.

Formalmente se define de esta forma:

Dado una relación **R** de cardinalidad **C** y grado **G**, una operación consiste en la creación de una relación R', cuyos elementos son las tuplas de R que cumplen una fórmula F. Esto se expresa de esta forma:

$$\sigma_F(R) = R'$$

F es una fórmula que puede contener diversos operadores:

- ◆ **Operadores de comparación.** <, >, ≤, ≥, ≠, =
- ◆ **Operadores booleanos.** ∧ (AND), ∨ (OR) y NO

Ejemplo:

$$\sigma_{\text{nombre}=\text{"Pepe"}}(\text{Cliente})$$

Eso obtiene una tabla cuyo contenido son las filas de la tabla **Cliente** cuya columna **nombre** valga **"Pepe"**

$$\sigma_{\text{nombre}=\text{"Pepe"} \wedge \text{edad}>25} (\text{Cliente})$$

Eso obtiene una tabla cuyo contenido son las filas de la tabla **Cliente** cuya columna **nombre** valga **"Pepe"** y la columna **edad** valga más que 25.

Las operaciones se pueden mezclar (ejemplo de selección y renombrado):

$$(\sigma_{\text{nombre}=\text{"Pepe"} \wedge \text{edad}>25} (\text{Cliente})) \rightarrow \text{Pepes}$$

proyección

Se trata de una operación que obtiene un subconjunto de una tabla basado sólo en algunas columnas de la tabla original.

Su definición formal es:

$$\Pi_{i,j,k,..} (R)$$

Si R es una relación de grado G, entonces la proyección $\Pi_{i,j,k,..}$ de R es un relación de grado G' con $G' < G$, formada por los atributos de nombre **i, j, k,...**

Ejemplo

$$\Pi_{\text{nombre}, \text{apellidos}} (\text{Cliente})$$

Obtiene una tabla con las columnas **nombre** y **apellidos** de la tabla **Cliente**

(2.5.4) operaciones básicas binarias

unión

Si R y S son dos relaciones del mismo grado y definidas sobre el mismo conjunto de atributos; $R \cup S$ es una relación del mismo grado que R y S y definida sobre el mismo conjunto de atributos, donde las tuplas de esa nueva relación son todas las de R y todas las de S.

Hay que tener en cuenta que las tuplas exactas no aparecen repetidas en la relación resultante.

Ejemplo:

R	A	B
	a	1
	a	3
	b	2
	c	1

S	A	B
	a	1
	b	2
	b	4

$R \cup S$	A	B
	a	1
	a	3
	b	2
	b	4
	c	1

diferencia

Si R y S son dos relaciones del mismo grado y definidas sobre el mismo conjunto de atributos; $R - S$ es una relación del mismo grado y atributos formada por todas las tuplas de R que no están presentes en S. Ejemplo (basado en las tablas anteriores):

$R - S$	A	B
	a	3
	c	1

intersección

Si R y S son dos relaciones del mismo grado y definidas sobre el mismo conjunto de atributos; $R \cap S$ es una relación del mismo grado y atributos formada por todas las tuplas de R que están también presentes en S. Ejemplo (basado en las tablas anteriores):

$R \cap S$	A	B
	a	1
	b	2

producto cartesiano

Si R es una relación de grado G1 y S es una relación de grado G2; $R \times S$ es una relación de grado G1+G2 cuyos G1 primeros componentes forman una tupla de R y los siguientes G2 forman una tupla de S.

Es decir el producto cartesiano es una relación que contiene todas las tuplas que resultan de combinar cada tupla de R con cada tupla de S.

Sistemas Gestores de Bases de Datos

(Unidad 2) Bases de Datos Relacionales

Ejemplo:

R	A	B
	a	1
	a	3

S	D	E	F
	a	3	1
	b	4	2
	c	5	4

R × S	A	B	D	E	F
	a	1	a	3	1
	a	1	b	4	2
	a	1	c	5	4
	a	3	a	3	1
	a	3	b	4	2
	a	3	c	5	4

combinación (join)

Se trata de un producto cartesiano en el que las tuplas resultantes cumplen una determinada condición.

$$R \bowtie_{\substack{a \Theta \\ b}} S$$

Donde Θ simboliza a cualquier operador de relación y **a** y **b** son atributos de R o de S.

Ejemplo:

$$\text{cliente} \bowtie_{\text{cliente.dni} = \text{alquiler.dni}} \text{alquiler}$$

Esto combinaría todos los clientes con todos los alquileres mostrando sólo las filas cuyo número de **dni** de la tabla de clientes coincida con el **dni** de la tabla de alquileres.

Es decir:

$$\text{cliente} \bowtie_{\text{cliente.dni} = \text{alquiler.dni}} \text{alquiler} = \sigma_{\text{cliente.dni} = \text{alquiler.dni}}(\text{cliente} \times \text{alquiler})$$

combinación natural (natural join)

Es una combinación que no indica condición alguna y que automáticamente obtiene las tuplas combinadas cuyos atributos comunes a ambas tablas sean del mismo valor. Es decir si la tabla **cliente** y la tabla **alquiler** tienen en común el atributo **dni**, entonces:

$$\text{cliente} \bowtie \text{alquiler}$$

Obtiene los datos combinados de cliente y alquiler cuyos dnis coincidan. La definición de $R \bowtie S$ es:

$$R \bowtie S = \sigma_{R.A1=S.A1 \wedge R.A2=S.A2 \wedge \dots \wedge R.An=S.An} (R \times S)$$

En ese esquema los atributos A1, A2,...An son atributos comunes a ambas relaciones.

división

Si R y S son relaciones de grado G1 y G2 respectivamente y A es el conjunto de atributos comunes a ambas relaciones; R : S obtiene una relación de grado G1-G2 en la que las tuplas resultantes son las tuplas formadas por los atributos distintos de A que poseen todos los valores posibles de A en la tabla S.

Es decir se obtienen las tuplas cuyos contenidos en los atributos comunes poseen todas las combinaciones almacenadas en S.

Ejemplo:

R	D	E	F
	a	34	uno
	a	34	dos
	c	23	uno
	d	11	dos

S	F
	uno
	dos

R : S	D	E
	a	34

(2.6) cálculo relacional

Utiliza expresiones matemáticas para generar fórmulas que han de cumplir los elementos de la relación; de modo que se obtiene una relación formada por las tuplas que cumplen esa fórmula.

La dificultad está en saber escribir la fórmula.

(2.6.1) cálculo relacional orientado a tuplas

Se basa en responder a la pregunta **¿Qué tuplas se encuentran en el resultado?**

- ◆ Esta pregunta se resuelve con una fórmula cuyo formato es

$$\{t|P(t)\}$$

Se lee: el conjunto de las tuplas **t** tales que cumplen el predicado **P**. Ese predicado es una expresión que puede contener una o más **variables de tupla** (variables que representan tuplas arbitrarias de una relación)

Hay dos tipos de variable de tupla:

- ◆ **ligadas**. Se cuantifican mediante los operadores \forall y \exists . Ejemplo:

$\{t \mid \exists s (s \in \text{cliente} \wedge t[\text{dni}] = s[\text{dni}] \wedge t[\text{edad}] > 25)\}$

Obtiene clientes con más de 25 años.

- ◆ **libres.** Son las que no utilizan los operadores anteriores (\forall y \exists .)

Las fórmulas pueden contener:

- ◆ **Valores constantes**
- ◆ **Variables de tupla.** Se denotan con una letra minúscula (normalmente t, u, v, \dots)
- ◆ **Expresiones como $t[i]$,** que significan la i -ésima componente de t .
- ◆ **Expresiones como $t[N]$,** donde N es el nombre de un atributo de t .

La construcción sigue estas reglas:

- (1) Se llaman átomos de fórmula a:
 - **$R(s)$,** donde R es el nombre de una relación y s una variable de tupla. Significa que la variable de tupla s , pertenece a R
 - **$S[i] \Theta u[j]$,** donde Θ representa un operador de relación ($>, <, >=, <=, \dots$). El atributo i -ésimo de S estará relacionado con el j -ésimo componente de u
- (2) Las expresiones de la fórmula se construyen:
 - Sabiendo que los átomos de fórmula son expresiones válidas
 - Si F_1 y F_2 son fórmulas, las expresiones $F_1 \wedge F_2, F_1 \vee F_2, F_1 \Leftarrow F_2$ y $\neg F_1$, son válidas
 - Si F es una fórmula se puede indicar $\forall s F(s)$, que significa para todas las tuplas de s se cumple F .
 - Si F es una fórmula se puede indicar $\exists s F(s)$, que significa que existe una tupla de s que cumple la fórmula F
 - Se admiten paréntesis en las fórmulas

(2.6.2) cálculo relacional orientado a dominios

Utilizan una sintaxis semejante, en esta forma:

$\{V_1, V_2, V_3, \dots, V_n \mid P(V_1, V_2, V_3, \dots, V_n)\}$

Donde V_1, V_2, \dots son variables de dominio de cada atributo y $P(V_1, V_2, \dots)$ es una fórmula sobre esas variables. Con esas variables se pueden utilizar los mismos operadores indicados anteriormente. De hecho la diferencia está en la aplicación a dominios y no a tuplas, lo que varía las expresiones.

Gracias este cálculo se generan expresiones en cálculo relacional semejantes a las del álgebra relacional.

(2.7) paso de entidad/relación al modelo relacional

(2.7.1) transformación de las entidades fuertes

En principio las entidades fuertes del modelo Entidad Relación son transformados al modelo relacional siguiendo estas instrucciones:

- ◆ **Entidades.** Las entidades pasan a ser tablas
- ◆ **Atributos.** Los atributos pasan a ser columnas o atributos de la tabla.
- ◆ **Identificadores principales.** Pasan a ser claves primarias
- ◆ **Identificadores candidatos.** Pasan a ser claves candidatas.

Esto hace que la transformación se produzca según este ejemplo:

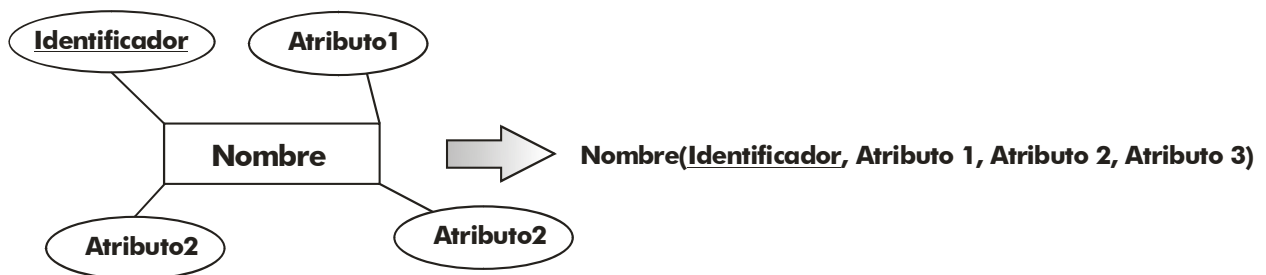


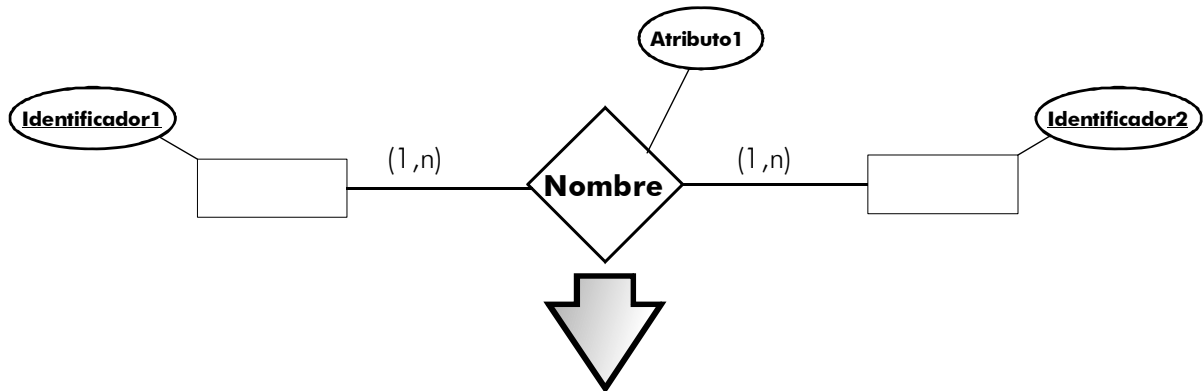
Ilustración 24, Transformación de una entidad fuerte al esquema relacional

(2.7.2) transformación de relaciones

La idea inicial es transformar a cada relación en una tabla en el modelo relacional. Pero hay que distinguir según el tipo de relación.

relaciones varios a varios

En las relaciones varios a varios (n a n en la cardinalidad mayor, la cardinalidad menor no importa salvo para determinar los índices), la relación se transforma en una tabla cuyos atributos son: los atributos de la relación y las claves de las entidades relacionadas (que pasarán a ser claves externas). La clave de la tabla la forman todas las claves externas:

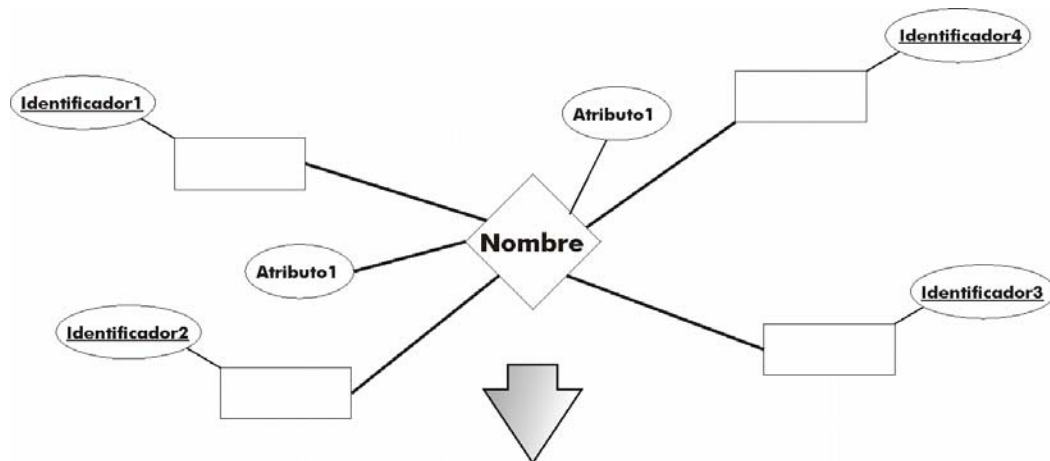


Nombre(Identificador1, Identificador2, Atributo1)

Ilustración 25, Transformación de una relación varios a varios

relaciones de orden n

Las relaciones ternarias, cuaternarias y n -arias que unen más de dos relaciones se transforman en una tabla que contiene los atributos de la relación más los identificadores de las entidades relacionadas. La clave la forman todas las claves externas:



Nombre(Identificador1, Identificador2, Identificador3, Identificador4, Atributo1, Atributo2)

Ilustración 26, Transformación en el modelo relacional de una entidad n -aria

relaciones uno a varios

Las relaciones binarios de tipo uno a varios no requieren ser transformadas en una tabla en el modelo relacional. En su lugar la tabla del lado **varios** (tabla relacionada) incluye como clave externa¹ el identificador de la entidad del lado **uno** (tabla principal):

¹ Clave externa, clave ajena, clave foránea, clave secundaria y *foreign key* son sinónimos

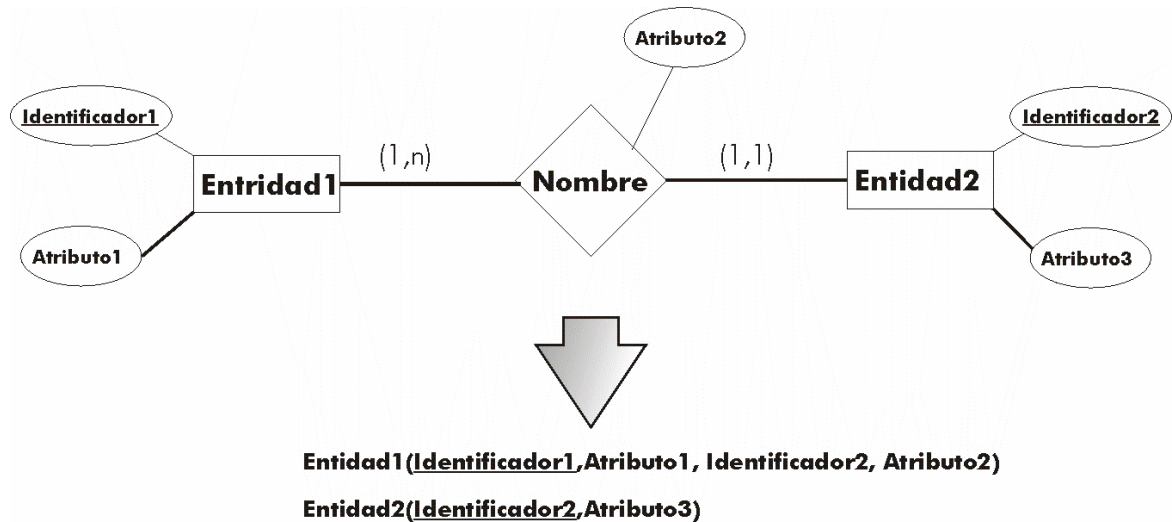


Ilustración 27, Transformación de una relación uno a varios

Así en el dibujo, el **identificador2** en la tabla **Entidad1** pasa a ser una clave externa. En el caso de que el número mínimo de la relación sea de **cero** (puede haber ejemplares de la entidad uno sin relacionar), se deberá permitir valores nulos en la clave externa **identificador2**. En otro caso no se podrán permitir (ya que siempre habrá un valor relacionado).

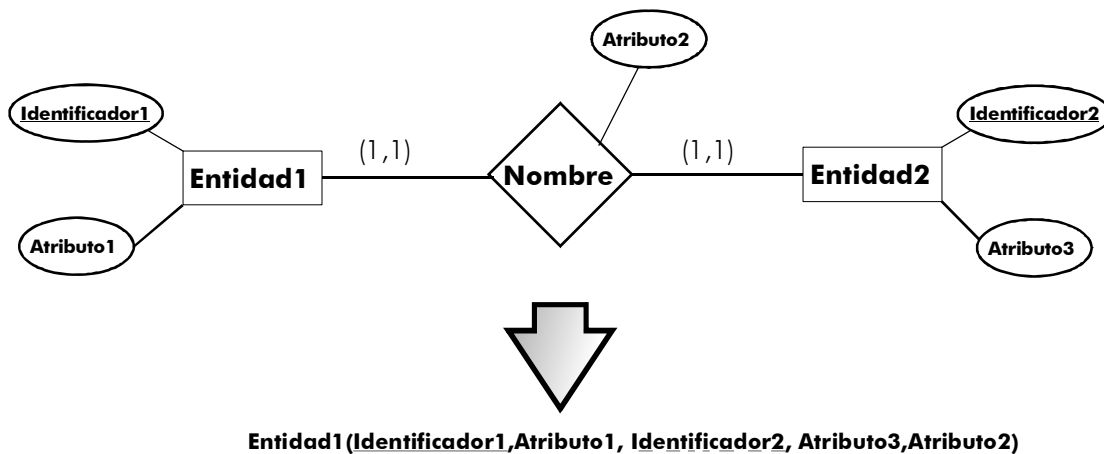


Ilustración 28, Posible solución a la cardinalidad 1 a 1

relaciones 1 a 1

En el caso de las relaciones entre dos entidades con todas las cardinalidades a 1; las posibilidades son varias:

- ◆ Colocar la clave de una de las entidades como clave externa de la otra tabla (da igual cual), teniendo en cuenta que dicha clave será **clave alternativa** además de ser clave secundaria.
- ◆ Generar una única tabla con todos los atributos de ambas entidades colocando como **clave principal** cualquiera de las claves de las dos

entidades. La otra clave será marca como **clave alternativa**. El nombre de la tabla sería el de la entidad más importante en el modelo conceptual

relaciones 0 a 1

Se trata de relaciones entre dos entidades con cardinalidad máxima de 1 en ambas direcciones, pero en una de ellas la cardinalidad mínima es 0. En este caso las soluciones difieren respecto a la anterior solución. No conviene generar una única tabla ya que habría numerosos valores nulos en la tabla (ya que los registros de una entidad no tienen relación todos los de la segunda).

La solución sería generar dos tablas, una para cada entidad. En la tabla con cardinalidad 0, se coloca como clave secundaria, la clave principal de la otra (dicha clave sería clave alternativa de esa tabla):

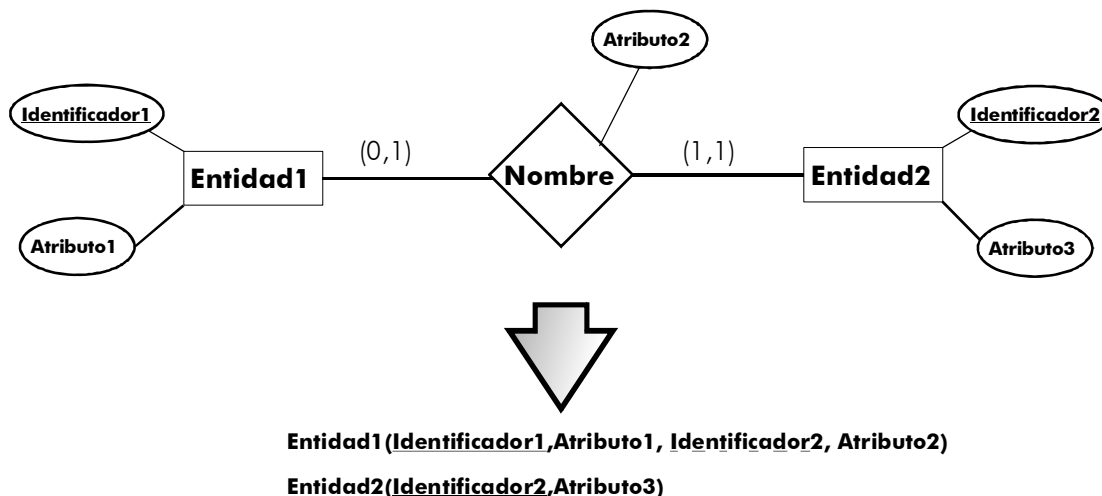


Ilustración 29, Solución a la relación 0 a 1

En el caso de que en ambos extremos nos encontremos con relaciones 0 a 1, entonces la solución es la misma, pero la clave que se copia en la tabla para ser clave secundaria, debe de ser tomada de la entidad que se relacione más con la otra (la que esté más cerca de tener la cardinalidad 1 a 1 en el otro extremo). Dicha clave secundaria, en este caso, no será clave alternativa (pero sí tendría restricción de unicidad).

relaciones recursivas

Las relaciones recursivas se tratan de la misma forma que las otras, sólo que un mismo atributo puede figurar dos veces en una tabla como resultado de la transformación:

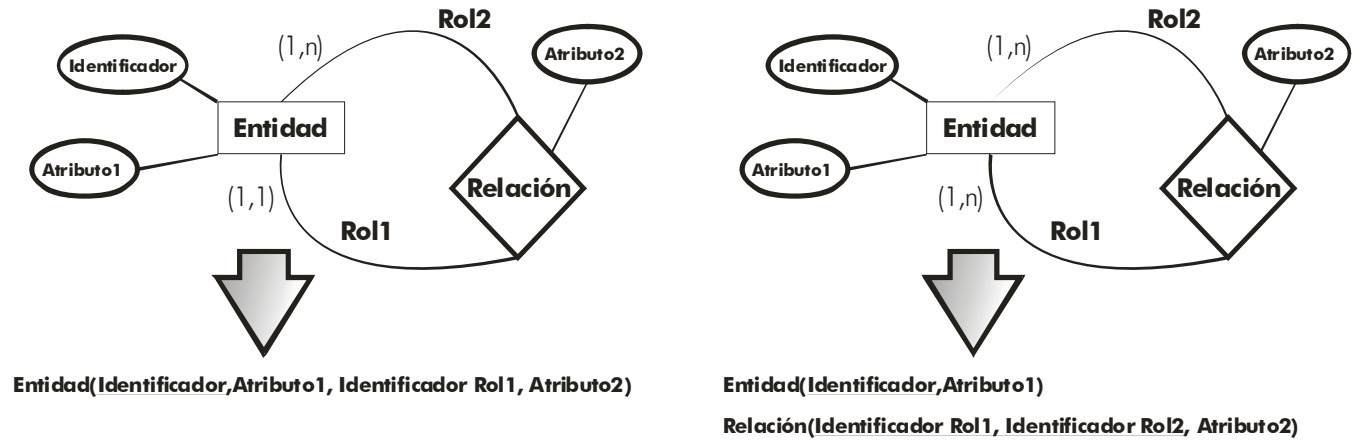


Ilustración 30, Transformación de relaciones recursivas en el modelo relacional

(2.7.3) entidades débiles

Toda entidad débil incorpora una relación implícita con una entidad fuerte. Esta relación no necesita incorporarse como tabla en el modelo relacional. Sí se necesita incorporar la clave de la entidad fuerte como clave externa en la entidad débil. Es más, en muchos casos esa clave externa, forma parte de la clave principal de la tabla que representa a la entidad débil. El proceso es:

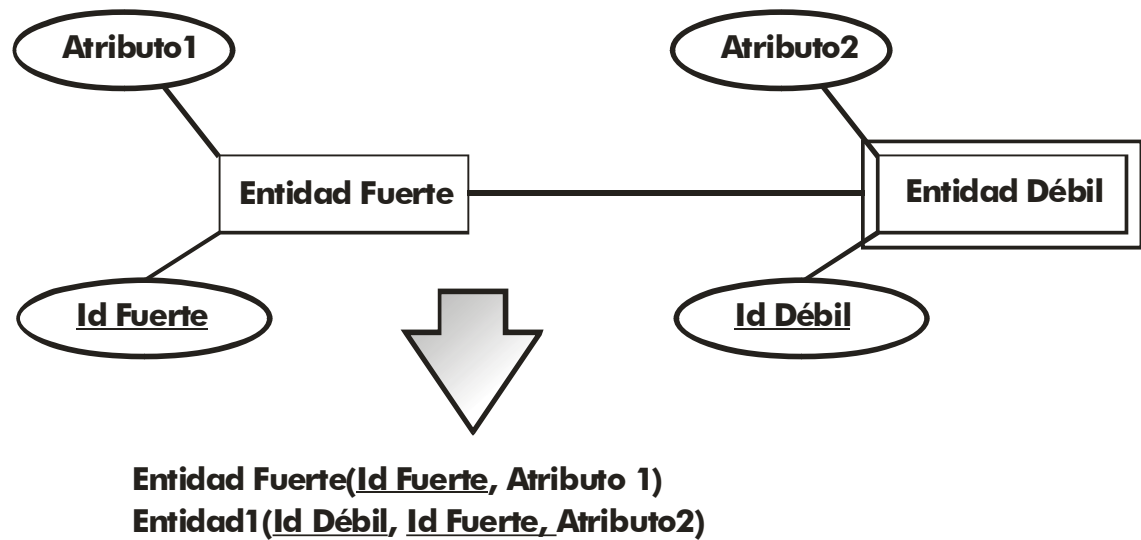


Ilustración 31, transformación de una entidad débil en el modelo relacional

En ocasiones el identificador de la entidad débil es suficiente para identificar los ejemplares de dicha entidad, entonces ese identificador quedaría como clave principal, pero el identificador de la entidad fuerte seguiría figurando como clave externa en la entidad débil.

(2.7.4) generalizaciones y especificaciones

Las generalizaciones y/o especificaciones se convierten al modelo relacional de esta forma (superentidad es la entidad más general, subentidades son las entidades más específicas):

- (1) Las subentidades pasan a ser tablas.
- (2) Si la clave de la superentidad es distinta de las subentidades, entonces se coloca el identificador de la superentidad en cada subentidad como clave externa o secundaria. Si la clave es la misma, entonces no es necesario.

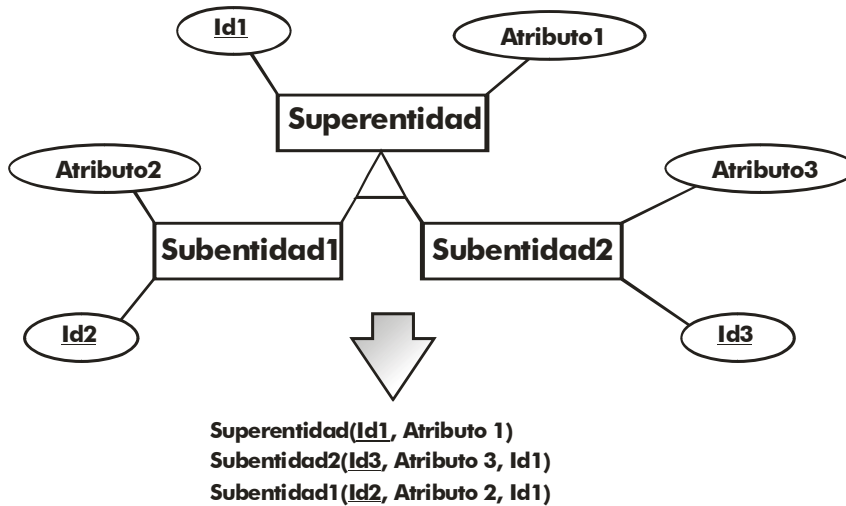


Ilustración 32, Proceso de transformación de relaciones ISA con clave propia

- (3) Si la clave es la misma, entonces todas las entidades tendrán el mismo atributo como clave principal:

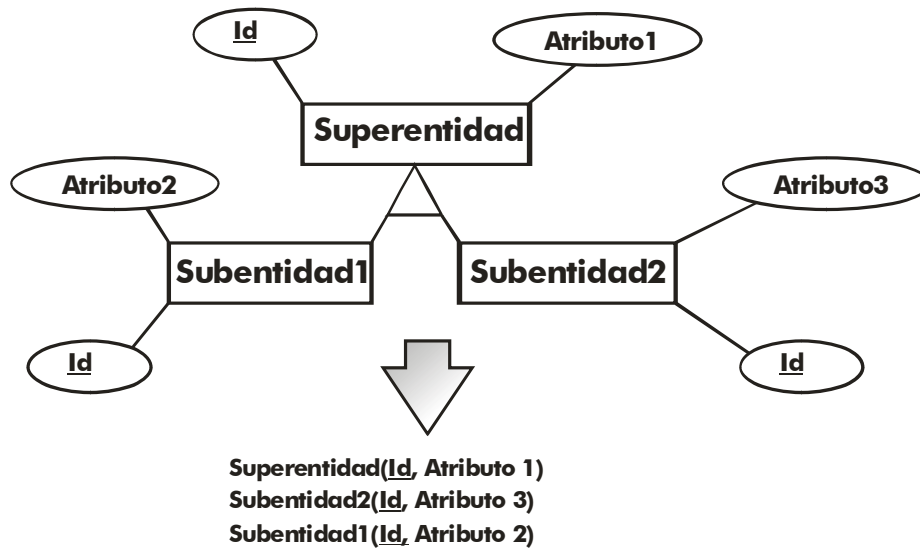


Ilustración 33, Proceso de transformación de relaciones ISA en el modelo relacional si tienen la misma clave

- (4) La superentidad debe generar una tabla sólo en el caso de que haya posibilidad de que exista un ejemplar de dicha entidad que no sea ejemplar de las subentidades (un trabajador que no sea ni bedel, ni administrativo, ni ninguna categoría). Esto se detecta en el caso de que la superentidad posea una cardinalidad mínima de 0 (debería marcarse en el esquema entidad/relación de forma clara). De otro modo basta con generar las tablas de las subentidades e incluir los atributos de la entidad superior:

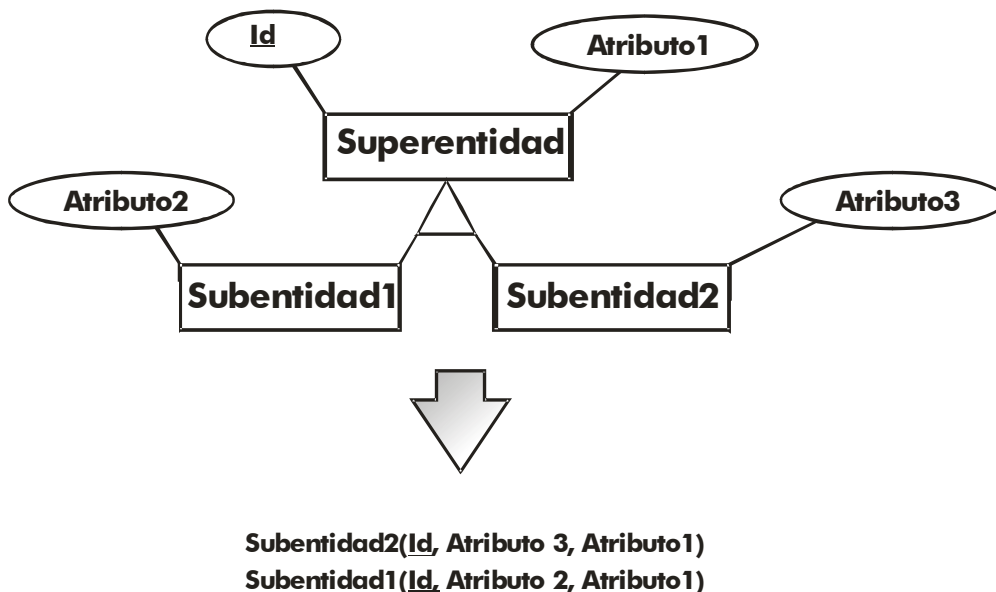


Ilustración 34, Paso de relaciones ISA al modelo relacional cuando toda superentidad figura como subentidad. En ese caso se heredan los atributos de la superentidad

También se suele evitar este hecho de heredar los atributos en el caso de que haya subentidades que no están asociadas a ninguna superentidad. En este caso se generaría una tabla para la superentidad y no se heredan los atributos, sólo se copia la clave de la superentidad en las subentidades.

(2.7.5) notas finales

El modelo conceptual entidad/relación es el verdadero mapa de la base de datos. Hay aspectos que no se reflejan al instante, por ejemplo el hecho de si la cardinalidad mínima es 0 o uno, o la obligatoriedad en una relación,... Son aspectos a tener en cuenta en el siguiente modelo (en el interno) al crear por ejemplo índices y restricciones. Por ello ese modelo es la referencia obligada de los profesionales de la base de datos (en especial de los administradores) y su contenido no debe dejar de tenerse en cuenta aunque ya tengamos el esquema relacional.

(2.8) representación de esquemas de bases de datos relacionales

En el tema 3, ya vimos como eran los esquemas relacionales. Ejemplo:

PIEZA(Tipo, Modelo, Nombre, Apellido1, Apellido2)

EMPRESA(CIF, Cod_Empresa, Nombre, Dirección)

SUMINISTROS(Tipo,Modelo, Cod_Empresa, Precio)

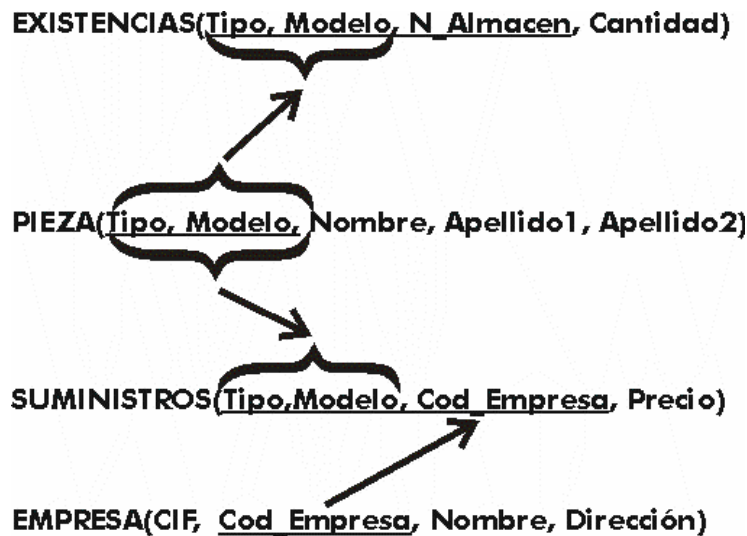
EXISTENCIAS(Tipo, Modelo, N_Almacen, Cantidad)

En ese tipo de esquemas es difícil ver las relaciones en los datos, algo que sí se ve muy bien en los esquemas entidad relación. Por ello se suelen complementar los esquemas clásicos con líneas y diagramas que representan esa información.

(2.8.1) Grafos relacionales

Es un esquema relacional en el que hay líneas que enlazan las claves principales con las claves secundarias para representar mejor las relaciones. A veces se representa en forma de nodos de grafos y otras se complementa el clásico.

Ejemplo:

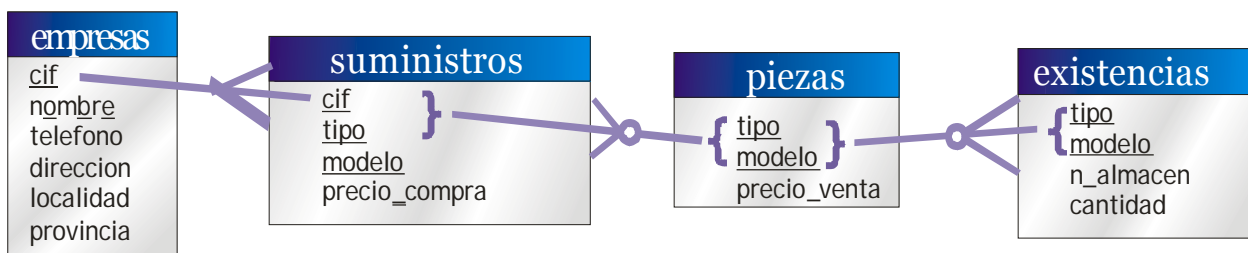


(2.8.2) Esquemas relacionales derivados del modelo entidad/relación

Hay quien los llama esquemas entidad/relación relacionales. De hecho es una mezcla entre los esquemas relacionales y los entidad/relación. Hoy en día se utiliza mucho, en especial por las herramientas CASE de creación de diseños de bases de datos.

Las tablas se representan en forma de rectángulo que contiene una fila por cada atributo y una fila inicial para la cabecera en la que aparece el nombre de la tabla. Después aparecen líneas que muestran la relación entre las claves y su cardinalidad.

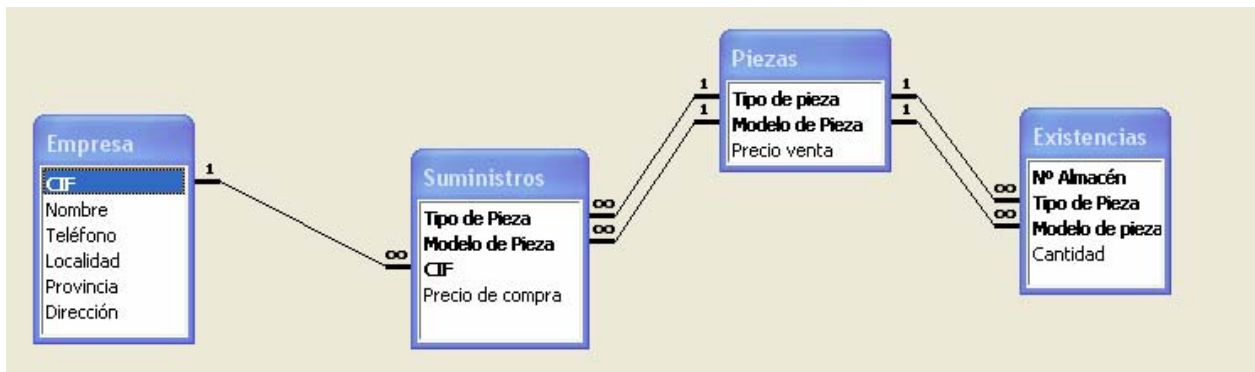
Uno de los más utilizados actualmente es éste:



Las cardinalidades se pueden mostrar en otros formatos, pero siempre se mostrarán en este tipo de esquemas.

Se ha hecho muy popular la forma de presentar esquemas relacionales del programa Microsoft Access.

Ejemplo:



Es una forma muy clara de representar relaciones y cardinalidades.

(2.9) normalización

(2.9.1) problemas del esquema relacional

Una vez obtenido el esquema relacional resultante del esquema entidad/relación que representa la base de datos, normalmente tendremos una buena base de datos. Pero otras veces, debido a fallos en el diseño o a problemas indetectables, tendremos un esquema que puede producir una base de datos que incorpore estos problemas:

- ◆ **Redundancia.** Se llama así a los datos que se repiten continua e innecesariamente por las tablas de las bases de datos. Cuando es excesiva es evidente que el diseño hay que revisarlo, es el primer síntoma de problemas y se detecta fácilmente.
- ◆ **Ambigüedades.** Datos que no clarifican suficientemente el registro al que representan. Los datos de cada registro podrían referirse a más de un registro o incluso puede ser imposible saber a qué ejemplar exactamente se están refiriendo. Es un problema muy grave y difícil de detectar.
- ◆ **Pérdida de restricciones de integridad.** Normalmente debido a **dependencias funcionales**. Más adelante se explica este problema. Se arreglan fácilmente siguiendo una serie de pasos concretos.
- ◆ **Anomalías en operaciones de modificación de datos.** El hecho de que al insertar un solo elemento haya que repetir tuplas en una tabla para variar unos pocos datos. O que eliminar un elemento suponga eliminar varias tuplas necesariamente (por ejemplo que eliminar un cliente suponga borrar seis o siete filas de la tabla de clientes, sería un error muy grave y por lo tanto un diseño terrible).

El principio fundamental reside en que las tablas deben referirse a objetos o situaciones muy concretas, relacionados exactamente con elementos

reconocibles por el sistema de información de forma inequívoca. Cada fila de una tabla representa inequívocamente un elemento reconocible en el sistema. Lo que ocurre es que conceptualmente es difícil agrupar esos elementos correctamente.

En cualquier caso la mayor parte de problemas se agravan si no se sigue un modelo conceptual y se decide crear directamente el esquema relacional. En ese caso el diseño tiene una garantía casi asegurada de funcionar mal.

Cuando aparecen los problemas enumerados entonces se les puede resolver usando reglas de normalización. Estas reglas suelen forzar la división de una tabla en dos o más tablas para arreglar ese problema.

(2.9.2) formas normales

Las formas normales se corresponde a una teoría de normalización iniciada por el propio **Codd** y continuada por otros autores (entre los que destacan **Boyce** y **Fagin**). Codd definió en 1970 la primera forma normal, desde ese momento aparecieron la segunda, tercera, la Boyce-Codd, la cuarta y la quinta forma normal.

Una tabla puede encontrarse en primera forma normal y no en segunda forma normal, pero no al contrario. Es decir los números altos de formas normales son más restrictivos (la quinta forma normal cumple todas las anteriores).

La teoría de formas normales es una teoría absolutamente matemática, pero en el presente manual se describen de forma más intuitiva.

Hay que tener en cuenta que muchos diseñadores opinan que basta con llegar a la forma Boyce-Codd, ya que la cuarta, y sobre todo la quinta, forma normal es polémica. Hay quien opina que hay bases de datos peores en quinta forma normal que en tercera. En cualquier caso debería ser obligatorio para cualquier diseñador llegar hasta la forma normal de Boyce-Codd.

(2.9.3) primera forma normal (1FN)

Es una forma normal inherente al esquema relacional. Es decir toda tabla realmente relacional la cumple.

Se dice que una tabla se encuentra en primera forma normal si impide que un atributo de una tupla pueda tomar más de un valor. La tabla:

TRABAJADOR		
DNI	Nombre	Departamento
12121212A	Andrés	Mantenimiento
12345345G	Andrea	Dirección Gestión

Visualmente es un tabla, pero no una tabla relacional (lo que en terminología de bases de datos relacionales se llama **relación**). No cumple la primera forma normal.

Sería primera forma normal si los datos fueran:

TRABAJADOR		
DNI	Nombre	Departamento
12121212A	Andrés	Mantenimiento
12345345G	Andrea	Dirección
12345345G	Andrea	Gestión

Esa tabla sí esta en primera forma normal.

(2.9.4) dependencias funcionales

dependencia funcional

Se dice que un conjunto de atributos (**Y**) depende funcionalmente de otro conjunto de atributos (**X**) si para cada valor de **X** hay un único valor posible para **Y**. Simbólicamente se denota por $X \rightarrow Y$.

Por ejemplo el nombre de una persona depende funcionalmente del DNI, para un DNI concreto sólo hay un nombre posible. En la tabla ejemplo anterior, el departamento no tiene dependencia funcional, ya que para un mismo DNI puede haber más de un departamento posible. Pero el nombre sí que depende del DNI.

Al conjunto **X** del que depende funcionalmente el conjunto **Y** se le llama **determinante**. Al conjunto **Y** se le llama **implicado**.

dependencia funcional completa

Un conjunto de atributos (**Y**) tiene una dependencia funcional completa sobre otro conjunto de atributos (**X**) si **Y** tiene dependencia funcional de **X** y además no se puede obtener de **X** un conjunto de atributos más pequeño que consiga una dependencia funcional de **Y** (es decir, no hay en **X** un determinante formado por atributos más pequeños).

Por ejemplo en una tabla de clientes, el conjunto de atributos formado por el **nombre** y el **dni** producen una dependencia funcional sobre el atributo **apellidos**. Pero no es plena ya que el **dni** individualmente, también produce una dependencia funcional sobre **apellidos**. El dni sí produce una dependencia funcional completa sobre el campo apellidos.

Una dependencia funcional completa se denota como $X \Rightarrow Y$

dependencia funcional elemental

Se produce cuando **X** e **Y** forman una dependencia funcional completa y además **Y** es un único atributo.

dependencia funcional transitiva

Es más compleja de explicar, pero tiene también utilidad. Se produce cuando tenemos tres conjuntos de atributos **X**, **Y** y **Z**. **Y** depende funcionalmente de **X** ($X \rightarrow Y$), **Z** depende funcionalmente de **Y** ($Y \rightarrow Z$). Además **X** no depende

funcionalmente de Y ($Y \dashv \rightarrow X$). Entonces ocurre que X produce una dependencia funcional transitiva sobre Z . Esto se denota como:

$(X \dashv \rightarrow Z)$

Por ejemplo si X es el atributo **Número de Clase** de un instituto, e Y es el atributo **Código Tutor**. Entonces $X \rightarrow Y$ (el tutor depende funcionalmente del número de clase). Si Z representa el **Código del departamento**, entonces $Y \rightarrow Z$ (el código del departamento depende funcionalmente del código tutor, cada tutor sólo puede estar en un departamento). Como ocurre que $Y \dashv \rightarrow X$ (el código de la clase no depende funcionalmente del código tutor, un código tutor se puede corresponder con varios códigos de clase). Entonces $X \dashv \rightarrow Z$ (el código del departamento depende transitivamente del código de la clase).

(2.9.5) segunda forma normal (2FN)

Ocurre si una tabla está en primera forma normal y además cada atributo que no sea clave, depende de forma funcional completa respecto de cualquiera de las claves. Toda la clave principal debe hacer dependientes al resto de atributos, si hay atributos que depende sólo de parte de la clave, entonces esa parte de la clave y esos atributos formarán otra tabla.

Ejemplo:

ALUMNOS				
DNI	Cod Curso	Nombre	Apellido1	Nota
12121219A	34	Pedro	Valiente	9
12121219A	25	Pedro	Valiente	8
3457775G	34	Ana	Fernández	6
5674378J	25	Sara	Crespo	7
5674378J	34	Sara	Crespo	6

Sistemas Gestores de Bases de Datos

(Unidad 2) Bases de Datos Relacionales

Suponiendo que el DNI y el código de curso formen una clave principal para esta tabla, sólo la nota tiene dependencia funcional completa. El nombre y los apellidos dependen de forma completa del DNI. La tabla no es 2FN, para arreglarlo:

ALUMNOS		
DNI	Nombre	Apellido1
12121219A	Pedro	Valiente
3457775G	Ana	Fernández
5674378J	Sara	Crespo

ASISTENCIA		
DNI	Cod Curso	Nota
12121219A	34	9
12121219A	25	8
3457775G	34	6
5674378J	25	7
5674378J	34	6

(2.9.6) tercera forma normal (3FN)

Ocurre cuando una tabla está en 2FN y además ningún atributo que no sea clave depende transitivamente de las claves de la tabla. Es decir no ocurre cuando algún atributo depende funcionalmente de atributos que no son clave.

Ejemplo:

ALUMNOS				
DNI	Nombre	Apellido1	Cod Provincia	Provincia
12121349A	Salvador	Velasco	34	Palencia
12121219A	Pedro	Valiente	34	Palencia
3457775G	Ana	Fernández	47	Valladolid
5674378J	Sara	Crespo	47	Valladolid
3456858S	Marina	Serrat	08	Barcelona

La Provincia depende funcionalmente del código de provincia, lo que hace que no esté en 3FN. El arreglo sería:

ALUMNOS			
DNI	Nombre	Apellido1	Cod Provincia
12121349A	Salvador	Velasco	34
12121219A	Pedro	Valiente	34
3457775G	Ana	Fernández	47
5674378J	Sara	Crespo	47

3456858S	Marina	Serrat	08
----------	--------	--------	----

PROVINCIA	
Cod Provincia	Provincia
34	Palencia
47	Valladolid
08	Barcelona

(2.9.7) forma normal de Boyce-Codd (FNBC o BCFN)

Ocurre si una tabla está en tercera forma normal y además todo determinante es una clave candidata. Ejemplo:

TUTORÍAS		
DNI	Asignatura	Tutor
12121219A	Lenguaje	Eva
12121219A	Matemáticas	Andrés
3457775G	Lenguaje	Eva
5674378J	Matemáticas	Guillermo
5674378J	Lenguaje	Julia
5634823H	Matemáticas	Guillermo

Esa tabla está en tercera forma normal (no hay dependencias transitivas), pero no en forma de Boyce - Codd, ya que **(DNI, Asignatura) → Tutor** y **Tutor → Asignatura**. En este caso la redundancia ocurre por mala selección de clave. La redundancia de la asignatura es completamente evitable. La solución sería:

TUTORÍAS	
DNI	Tutor
12121219A	Eva
12121219A	Andrés
3457775G	Eva
5674378J	Guillermo
5674378J	Julia
5634823H	Guillermo

ASIGNATURASTUTOR	
Asignatura	Tutor
Lenguaje	Eva
Matemáticas	Andrés

Matemáticas	Guillermo
Lenguaje	Julia

En las formas de Boyce-Codd hay que tener cuidado al descomponer ya que se podría perder información por una mala descomposición

(2.9.8) cuarta forma normal (4FN). dependencias multivaluadas

dependencia multivaluada

Para el resto de formas normales (las diseñadas por Fagin, mucho más complejas), es importante definir este tipo de dependencia, que es distinta de las funcionales. Si las funcionales eran la base de la segunda y tercera forma normal (y de la de Boyce-Codd), éstas son la base de la cuarta forma normal.

Una dependencia multivaluada de X sobre Z (es decir $X \twoheadrightarrow Z$), siendo X y Z atributos de la misma tabla, ocurre cuando Y tiene un conjunto de valores bien definidos sobre cualquier valor de X. Es decir, dado X sabremos los posibles valores que puede tomar Y.

Se refiere a posibles valores (en plural) y se trata de que los valores de ese atributo siempre son los mismos según el valor de un atributo y no del otro.

Ejemplo:

Nº Curso	Profesor	Material
17	Eva	1
17	Eva	2
17	Julia	1
17	Julia	2
25	Eva	1
25	Eva	2
25	Eva	3

La tabla cursos, profesores y materiales del curso. La tabla está en FNBC ya que no hay dependencias transitivas y todos los atributos son clave sin dependencia funcional hacia ellos. Sin embargo hay redundancia. Los materiales se van a repetir para cualquier profesor dando cualquier curso, ya que los profesores van a utilizar todos los materiales del curso (de no ser así no habría ninguna redundancia).

Los materiales del curso dependen del curso y no del profesor en una dependencia multivaluada (no hay dependencia funcional ya que los posibles valores son varios). Para el par N° de curso y profesor podemos saber los materiales; pero lo sabemos por el curso y no por el profesor.

cuarta forma normal

Ocurre esta forma normal cuando una tabla está en forma normal de Boyce Codd y toda dependencia multivaluada es una dependencia funcional. Para la tabla anterior la solución serían dos tablas:

Nº Curso	Material
17	1
17	2
25	1
25	2
25	3

Nº Curso	Profesor
17	Eva
17	Julia
25	Eva

Un teorema de Fagin indica cuando hay tres pares de conjuntos de atributos X , Y y Z si ocurre $X \twoheadrightarrow Y$ y $X \twoheadrightarrow Z$ (Y y Z tienen dependencia multivaluada sobre X), entonces las tablas X, Y y X, Z reproducen sin perder información lo que poseía la tabla original. Este teorema marca la forma de dividir las tablas hacia una 4FN

(2.9.9) quinta forma normal (5FN)

dependencias de JOIN o de reunión

Una proyección de una tabla es la tabla resultante de tomar un subconjunto de los atributos de una tabla (se trata de la operación proyección, Π , del álgebra relacional).

Se dice que se tiene una tabla con dependencia de tipo **JOIN** si se puede obtener esa tabla como resultado de combinar (mediante la operación JOIN del álgebra relacional) varias proyecciones de la misma.

quinta forma normal

Ocurre cuando está en 4FN y además no hay proyecciones que combinadas formen la tabla original, o si las hay son consecuencia de aplicar la clave principal. Es la más compleja y polémica de todas. Polémica pues no está claro en muchas ocasiones que sea una solución mejor sacar las proyecciones de la tabla. Fue definida también por Fagin.

En definitiva una tabla está en 5FN si está en 4FN y no hay restricciones impuestas por el creador de la base de datos.

Es raro encontrarse este tipo de problemas cuando la normalización llega a 4FN. Se deben a restricciones muy concretas. Ejemplo:

Sistemas Gestores de Bases de Datos

(Unidad 2) Bases de Datos Relacionales

Proveedor	Material	Proyecto
1	1	2
1	2	1
2	1	1
1	1	1

Indican códigos de material suministrado por un proveedor y utilizado en un determinado proyecto.

Si ocurre una restricción especial como por ejemplo: Cuando un proveedor nos ha suministrado alguna vez un determinado material, si ese material aparece en otro proyecto, haremos que el proveedor nos suministre también ese material para ese proyecto.

Eso ocurre en los datos como el proveedor número 1 nos suministró el material número 1 para el proyecto 2 y en el proyecto 1 utilizamos el material 1, aparecerá la tupla proveedor 1, material 1 y proyecto 1.

La dependencia que produce esta restricción es lejana y se la llama **de reunión**. Para esa restricción esta división en tablas sería válida:

Proveedor	Material
1	1
1	2
2	1

Material	Proyecto
1	2
2	1
1	1

Esa descomposición no pierde valores en este caso, sabiendo que si el proveedor nos suministra un material podremos relacionarle con todos los proyectos que utilizan ese material.

Resumiendo, una tabla no está en quinta forma normal si hay una descomposición de esa tabla que muestre la misma información que la original.

Normalmente se crean tablas en quinta forma normal cuando en la misma tabla hay muchos atributos y es casi inmanejable o cuando hay muchos registros y pocos atributos. En el caso de que haya muchos atributos se divide la tabla en dos donde la clave es la misma en ambas tablas.

(Unidad 3)

Lenguaje SQL

notas previas

versión de SQL

Aunque estos apuntes sirven como guía de uso de SQL, en estos apuntes sobre todo se escenifica el SQL del programa comercial Oracle. Normalmente se indican las diferencias entre Oracle y SQL estándar, pero todos los ejemplos han sido pensados para utilizarse en Oracle.

La razón de utilizar Oracle como base de trabajo se debe a su respeto por SQL estándar (hasta cierto punto) y por ser el SGBD de referencia en el mundo desde hace ya muchos años.

formato de las instrucciones en los apuntes

En este manual en muchos apartados se indica sintaxis de comandos. Esta sintaxis sirve para aprender a utilizar el comando, e indica la forma de escribir dicho comando en el programa utilizado para escribir SQL.

En el presente manual la sintaxis de los comandos se escribe en párrafos sombreados de azul celeste con el reborde en azul oscuro. Ejemplo:

```
SELECT * | {[DISTINCT] columna | expresión [alias], ...}  
FROM tabla;
```

Otras veces se describen códigos de ejemplo de un comando. Los ejemplos se escriben también con fondo celeste, pero sin el reborde. Ejemplo:

```
SELECT nombre FROM cliente;
```

Los ejemplos sirven para escenificar una instrucción concreta, la sintaxis se utiliza para indicar la forma de utilizar un comando. Para indicar la sintaxis de un comando se usan símbolos especiales. Los símbolos que utiliza este libro (de acuerdo con la sintaxis que se utiliza normalmente en cualquier documentación de este tipo) son:

Sistemas Gestores de Bases de Datos

(Unidad 3) Lenguaje SQL

- ◆ **PALABRA** Cuando en la sintaxis se utiliza una palabra en negrita, significa que es un comando que hay que escribir literalmente.
- ◆ *texto* El texto que aparece en cursiva sirve para indicar que no hay que escribirlo literalmente, sino que se refiere a un tipo de elemento que se puede utilizar en el comando. Ejemplo:

```
SELECT columna FROM tabla;
```

El texto *columna* hay que cambiarlo por un nombre concreto de columna (nombre, apellidos,...) , al igual que *tabla* se refiere a un nombre de tabla concreto.

- ◆ **[] (corchetes)**. Los corchetes sirven para encerrar texto que no es obligatorio en el comando, es decir para indicar una parte opcional.
- ◆ **| (barra vertical)**. Este símbolo (|) , la barra vertical, indica opción, es decir que se puede elegir entre varias opciones
- ◆ **... (puntos suspensivos)** Indica que se puede repetir el texto anterior en el comando continuamente (significaría, y *así sucesivamente*)
- ◆ **{ (llaves)** Las llaves sirven para indicar opciones mutuamente exclusivas pero obligatorias. Es decir, opciones de las que sólo se puede elegir una opción, pero de las que es obligado elegir una. Ejemplo:

```
SELECT { * | columna | expresión }  
FROM tabla;
```

El ejemplo anterior indicaría que se debe elegir obligatoriamente el asterisco o un nombre de columna o una expresión. Si las llaves del ejemplo fueran corchetes, entonces indicarían que incluso podría no aparecer ninguna opción.

(3.2) introducción

(3.2.1) objetivos

SQL es el lenguaje fundamental de los SGBD relacionales. Se trata de uno de los lenguajes más utilizados de la historia de la informática.

SQL pretende ser un lenguaje que simula su escritura en lenguaje normal. De ahí que se le considere un lenguaje de cuarta generación. Consta de palabras especiales y de expresiones.

Se trata de un lenguaje que intenta agrupar todas las funciones que se le pueden pedir a una base de datos, por lo que es el lenguaje utilizado tanto por administradores como por programadores o incluso usuarios avanzados.

Es un lenguaje de tipo declarativo, se especifica **qué** es lo que desea realizar, no el **cómo** queremos realizarlo (el **cómo** es la pregunta de los lenguajes imperativos, como **C**, **C++** o **Java**).

(3.2.2) modos de utilización

ejecución directa. SQL interactivo

Las instrucciones SQL se introducen a través de una herramienta que las traduce inmediatamente a la base de datos, por lo que se ejecutan al instante.

ejecución incrustada o embebida

Las instrucciones SQL se colocan como parte del código de otro lenguaje anfitrión (C, Java, Pascal, Visual Basic,...). Estas instrucciones están separadas del resto del código de forma conveniente. Al compilar el código se utiliza un precompilador de la propia base de datos para traducir el SQL.

ejecución dinámica

Se trata de SQL incrustado en módulos especiales que pueden ser invocados una y otra vez desde distintas aplicaciones.

(3.3) historia del lenguaje SQL

El nacimiento del lenguaje SQL data de 1970 cuando **E. F. Codd** publica su libro: "**Un modelo de datos relacional para grandes bancos de datos compartidos**". Ese libro dictaría las directrices de las bases de datos relacionales. Apenas dos años después **IBM** (para quien trabajaba Codd) utiliza las directrices de Codd para crear el **Standard English Query Language (Lenguaje Estándar Inglés para Consultas)** al que se le llamó **SEQUEL**. Más adelante se le asignaron las siglas **SQL (Standard Query Language, lenguaje estándar de consulta)** aunque en inglés se siguen pronunciando **secuel**. En español se le llama **esecuele**.

En 1979 Oracle presenta la primera implementación comercial del lenguaje. Poco después se convertía en un estándar en el mundo de las bases de datos avalado por los organismos **ISO** y **ANSI**. En el año 1986 se toma como lenguaje estándar por ANSI de los SGBD relacionales. Un año después lo adopta ISO, lo que convierte a SQL en estándar mundial.

En 1989 aparece el estándar ISO (y ANSI) llamado **SQL89** o **SQL1**. En 1992 aparece la nueva versión estándar de SQL (a día de hoy sigue siendo la más conocida) llamada **SQL92**. En 1999 se aprueba un nuevo SQL estándar que incorpora mejoras que incluyen triggers, procedimientos, funciones,... y otras características de las bases de datos objeto-relacionales; dicho estándar se conoce como **SQL99**. El último estándar es el del año 2003 (**SQL2003**) que añade secuencias y sobre todo el soporte para **XML** (un nuevo estándar en formatos de datos).

(3.4) elementos del lenguaje SQL

(3.4.1) código SQL

El código SQL consta de los siguientes elementos:

- ◆ **Comandos.** Las distintas instrucciones que se pueden realizar desde SQL
 - **SELECT.** Se trata del comando que permite realizar consultas sobre los datos de la base de datos. Obtiene datos de la base de datos. A ésta parte del lenguaje se la conoce como **DQL** (**Data Query Language**, Lenguaje de consulta de datos); pero es parte del DML del lenguaje.
 - **DML, Data Manipulation Language** (Lenguaje de manipulación de datos). Modifica filas (registros) de la base de datos. Lo forman las instrucciones **INSERT, UPDATE, MERGE y DELETE.**
 - **DDL, Data Definition Language** (Lenguaje de definición de datos). Permiten modificar la estructura de las tablas de la base de datos. Lo forman las instrucciones **CREATE, ALTER, DROP, RENAME y TRUNCATE.**
 - **DCL, Data Control Language** (Lenguaje de control de datos). Administran los derechos y restricciones de los usuarios. Lo forman las instrucciones **GRANT y REVOKE.**
 - **Instrucciones de control de transacciones.** Administran las modificaciones creadas por las instrucciones DML. Lo forman las instrucciones **ROLLBACK, COMMIT y SAVEPOINT.** Se las considera parte del DCL.
- ◆ **Cláusulas.** Son palabras especiales que permiten modificar el funcionamiento de un comando (**WHERE, ORDER BY,...**)
- ◆ **Operadores.** Permiten crear expresiones complejas. Pueden ser aritméticos (+, -, *, /, ...) lógicos (>, <, !=, <>, AND, OR, ...)
- ◆ **Funciones.** Para conseguir valores complejos (SUM(), DATE(), ...)
- ◆ **Constantes.** Valores literales para las consultas, números, textos, caracteres, ...
- ◆ **Datos.** Obtenidos de la propia base de datos

(3.4.2) normas de escritura

- ◆ En SQL no se distingue entre mayúsculas y minúsculas. Da lo mismo como se escriba.
- ◆ El final de una instrucción lo determina el signo del punto y coma

- ◆ Los comandos SQL (SELECT, INSERT,...) pueden ser partidos por espacios o saltos de línea antes de finalizar la instrucción
- ◆ Se pueden tabular líneas para facilitar la lectura si fuera necesario
- ◆ Los comentarios en el código SQL comienzan por */** y terminan por **/* (excepto en algunos SGBD)

(3.4.3) proceso de las instrucciones SQL

El proceso de una instrucción SQL es el siguiente:

- (1) Se analiza la instrucción. Para comprobar la sintaxis de la misma
- (2) Si es correcta se valora si los metadatos de la misma son correctos. Se comprueba esto en el diccionario de datos.
- (3) Si es correcta, se optimiza, a fin de consumir los mínimos recursos posibles.
- (4) Se ejecuta la sentencia y se muestra el resultado al emisor de la misma.

(3.5)

tipos de datos

A la hora de crear tablas, hay que indicar el tipo de datos de cada campo. Para ello describimos la siguiente tabla:

Descripción	Tipos ANSI SQL	SQL Server	Oracle SQL	Tipo MySQL
Texto de anchura fija	CHARACTER(<i>n</i>) CHAR(<i>n</i>)	CHAR(<i>n</i>)	CHAR(<i>n</i>)	CHAR(<i>n</i>)
Texto de anchura variable	CHARACTER VARYING(<i>n</i>) CHAR VARYING(<i>n</i>)	VARCHAR(<i>n</i>)	VARCHAR2(<i>n</i>)	VARCHAR(<i>n</i>)
Texto de anchura fija para caracteres nacionales	NATIONAL CHARACTER(<i>n</i>) NATIONAL CHAR(<i>n</i>) NCHAR(<i>n</i>)	NCHAR(<i>n</i>)	NCHAR(<i>n</i>)	
Texto de anchura variable para caracteres	NATIONAL CHARACTER VARYING(<i>n</i>) NATIONAL CHAR	NVARCHAR(<i>n</i>)	NVARCHAR2(<i>n</i>)	

Descripción	Tipos ANSI SQL	SQL Server	Oracle SQL	Tipo MySQL
nacionales	VARYING(<i>n</i>)			
	NCHAR VARYING(<i>n</i>)			
Enteros	INTEGER	INT	NUMBER(38)	INT
	INT	INT		
	SMALLINT	SMALL INT		SMALL INT TINY INT
Decimal de coma variable	FLOAT(<i>b</i>)	FLOAT	NUMBER	FLOAT(<i>m,d</i>)
	DOUBLE			DOUBLE(<i>m,d</i>)
	DOUBLE PRECISION			
	REAL			
Decimal de coma fija	NUMERIC(<i>m,d</i>)	NUMERIC(<i>m,d</i>)	NUMBER(<i>m,d</i>)	
	DECIMAL(<i>m,d</i>)	DECIMAL(<i>m,d</i>)		DECIMAL(<i>m,d</i>)
Fechas	DATE		DATE	DATE
Fecha y hora	TIMESTAMP	TIMESTAMP	TIMESTAMP	TIMESTAMP
		DATETIME		DATETIME
		SMALLDATETIME		TIME
Intervalos	INTERVAL		INTERVAL	YEAR
Lógicos	BIT	BINARY		BIT BOOL
Texto gran longitud	CLOB	TEXT y NTEXT	LONG (en desuso) y CLOB	TEXT, MEDIUM TEXT y LONG TEXT
Binario de gran longitud	BLOB	IMAGE	RAW, LONG RAW	BLOB, MEDIUM BLOB
			BLOB	BLOB y LONG BLOB

Durante el resto del manual se hará referencia sólo a los tipos Oracle.

(3.5.1) textos

Para los textos disponemos de los siguientes tipos (Oracle):

- ◆ **VARCHAR2** . Para textos de longitud variable. Su tamaño depende de la base de datos (en Oracle es de 4000)
- ◆ **CHAR**. Para textos de longitud fija (en Oracle hasta 2000 caracteres).
- ◆ **NCHAR**. Para el almacenamiento de caracteres nacionales de texto fijo
- ◆ **NVARCHAR**. Para el almacenamiento de caracteres nacionales de longitud variable.

En todos estos tipos se indican los tamaños entre paréntesis tras el nombre del tipo. Conviene poner suficiente espacio para almacenar los valores. En el caso de los VARCHAR, no se malgasta espacio por poner más espacio del deseado ya que si el texto es más pequeño que el tamaño indicado, el resto del espacio se ocupa.

(3.5.2) números

El tipo **NUMBER** es un formato versátil que permite representar todo tipo de números. Su rango recoge números de entre 10^{-130} y $9,9999999999 * 10^{128}$. Fuera de estos rangos Oracle devuelve un error.

Los números decimales (números de coma fija) se indican con **NUMBER(p,s)**, donde *p* es la precisión máxima y *s* es la escala (número de decimales a la derecha de la coma). Por ejemplo, NUMBER (8,3) indica que se representan números de ocho cifras de precisión y tres decimales. Los decimales en Oracle se presenta con el **punto y no con la coma**.

Para números enteros se indica **NUMBER(p)** donde *p* es el número de dígitos. Eso es equivalente a NUMBER(*p*,0).

Para números de coma flotante (equivalentes a los **float** o **double** de muchos lenguajes de programación) simplemente se indica el texto **NUMBER** sin precisión ni escala.

precisión y escala

La cuestión de la precisión y la escala es compleja. Para entenderla mejor, se muestran estos ejemplos:

Formato	Número escrito por el usuario	Se almacena como...
NUMBER	345255.345	345255.345
NUMBER(9)	345255.345	345255
NUMBER(9,2)	345255.345	345255.36
NUMBER(7)	345255.345	Da error de precisión
NUMBER(7,-2)	345255.345	345300
NUMBER(7,2)	345255.345	Da error de precisión

En definitiva, la precisión debe incluir todos los dígitos del número (puede llegar hasta 38 dígitos). La escala sólo indica los decimales que se respetarán del número, pero si es negativa indica ceros a la izquierda del decimal.

(3.5.3) fechas y horas

DATE

El tipo **DATE** permite almacenar fechas. Las fechas se pueden escribir en formato día, mes y año entre comillas. El separador puede ser una barra de dividir, un guión y casi cualquier símbolo.

Para almacenar la fecha actual la mayoría de bases de datos proporcionan funciones (como SYSDATE en Oracle) que devuelven ese valor.

TIMESTAMP

Es una extensión del anterior, almacena valores de día, mes y año, junto con hora, minuto y segundos (incluso con decimales). Con lo que representa un instante concreto en el tiempo. Un ejemplo de **TIMESTAMP** sería '2/2/2004 18:34:23,34521'. En este caso si el formato de fecha y hora del sistema está pensado para el idioma español, el separador decimal será la coma (y no el punto).

intervalos

Sirven para almacenar intervalos de tiempo (no fechas, sino una suma de elementos de tiempo). En el caso de Oracle son:

INTERVAL YEAR TO MONTH

Este tipo de datos almacena años y meses. Tras la palabra **YEAR** se puede indicar la precisión de los años (cifras del año), por defecto es de dos. Ejemplo:

```
CREATE TABLE tiempos (meses INTERVAL YEAR(3) TO MONTH);  
INSERT INTO tiempos VALUES('3-2');
```

En el ejemplo se inserta un registro que representa 3 años y dos meses.

INTERVAL DAY TO SECOND

Representa intervalos de tiempo que expresan días, horas, minutos y segundos. Se puede indicar la precisión tras el texto **DAY** y el número de decimales de los segundos tras el texto **SECOND**. Ejemplo:

```
CREATE TABLE tiempos (dias INTERVAL DAY(3) TO SECOND(0));  
INSERT INTO tiempos VALUES('2 7:12:23');
```

(3.5.4) Datos grandes

Son tipos pensados para almacenar datos de tamaño muy grande. No pueden poseer índices ni ser parte de claves.

LOB

Son varios tipos de datos que permiten almacenar valores muy grandes. Incluye a **BLOB** (datos binarios grandes) y **CLOB** (textos de tamaño grande).

LONG

Se trata de un tipo de datos que actualmente se mantiene por compatibilidad. Se recomienda encarecidamente utilizar en su lugar el tipo **CLOB**.

RAW

Sirve para almacenar valores binarios de hasta 2000 bytes (se puede especificar el tamaño máximo entre paréntesis). El valor LONG RAW almacena hasta 2GB. Está en desuso y se aconseja utilizar BLOB.

(3.5.5) ROWID

Valor hexadecimal que representa la dirección única de una fila en su tabla.

(3.6) consultas de datos con SQL. DQL

(3.6.1) capacidades

DQL es la abreviatura del **Data Query Language** (lenguaje de consulta de datos) de SQL. El único comando que pertenece a este lenguaje es el versátil comando **SELECT**. Este comando permite:

- ◆ Obtener datos de ciertas columnas de una tabla (**proyección**)
- ◆ Obtener registros (filas) de una tabla de acuerdo con ciertos criterios (**selección**)
- ◆ Mezclar datos de tablas diferentes (**asociación, join**)
- ◆ Realizar cálculos sobre los datos
- ◆ Agrupar datos

(3.6.2) sintaxis sencilla del comando SELECT

```
SELECT * | {[DISTINCT] columna | expresión [[AS] alias],  
...}  
FROM tabla;
```

Donde:

- ◆ *. El asterisco significa que se seleccionan todas las columnas
- ◆ **DISTINCT**. Hace que no se muestren los valores duplicados.
- ◆ **columna**. Es el nombre de una columna de la tabla que se desea mostrar

Sistemas Gestores de Bases de Datos

(Unidad 3) Lenguaje SQL

- ◆ **expresión.** Una expresión válida SQL
- ◆ **alias.** Es un nombre que se le da a la cabecera de la columna en el resultado de esta instrucción.

Ejemplos:

```
/* Selección de todos los registros de la tabla clientes
*/
SELECT * FROM Clientes;
/* Selección de algunos campos*/
SELECT nombre, apellido1, apellido2 FROM Clientes;
```

relación con el álgebra relacional

El comando SELECT permite implementar de forma exacta todas las consultas del álgebra relación. En concreto la proyección se implementa así. Ejemplo:

$$\Pi_{\text{nombre, apellidos}}(\text{Cliente})$$

La proyección anterior en SQL se escribe como:

```
SELECT nombre, apellidos FROM Cliente;
```

El renombrado es lo que permite realizar la cláusula AS, sólo que en SQL se utiliza exclusivamente para abreviar las referencias a una tabla. Ejemplo:

```
SELECT c.nombre, c.apellidos, a.fecha
FROM Clientes AS c, ALQUILERES AS a
WHERE c.dni=a.dni;
```

La palabra AS se puede obviar:

```
SELECT c.nombre, c.apellidos, a.fecha
FROM Clientes c, ALQUILERES a
WHERE c.dni=a.dni;
```

(3.7) cálculos

(3.7.1) aritméticos

Los operadores + (suma), - (resta), * (multiplicación) y / (división), se pueden utilizar para hacer cálculos en las consultas. Cuando se utilizan como expresión en una consulta SELECT, no modifican los datos originales sino que como resultado de la vista generada por SELECT, aparece un nueva columna. Ejemplo:

```
SELECT nombre, precio, precio*1.16 FROM articulos
```

Esa consulta obtiene tres columnas. La tercera tendrá como nombre la expresión utilizada, para poner un alias basta utilizar dicho alias tras la expresión:

```
SELECT nombre, precio, precio*1.16 AS precio_con_iva
FROM articulos;
```

La prioridad de esos operadores es la normal: tienen más prioridad la multiplicación y división, después la suma y la resta. En caso de igualdad de prioridad, se realiza primero la operación que esté más a la izquierda. Como es lógico se puede evitar cumplir esa prioridad usando paréntesis; el interior de los paréntesis es lo que se ejecuta primero.

Cuando una expresión aritmética se calcula sobre valores NULL, el resultado de la expresión es siempre NULL.

(3.7.2) concatenación de textos

Todas las bases de datos incluyen algún operador para encadenar textos. En SQLSERVER es el signo & en Oracle son los signos || Ejemplo (Oracle):

```
SELECT tipo, modelo, tipo || '-' || modelo "Clave Pieza"
FROM piezas;
```

El resultado puede ser:

TIPO	MODELO	Clave Pieza
AR	6	AR-6
AR	7	AR-7
AR	8	AR-8
AR	9	AR-9
AR	12	AR-12
AR	15	AR-15
AR	20	AR-20
AR	21	AR-21
BI	10	BI-10
BI	20	BI-20
BI	38	BI-38
BI	57	BI-57

(3.8) condiciones

Se pueden realizar consultas que restrinjan los datos de salida de las tablas. Para ello se utiliza la cláusula **WHERE**. Esta cláusula permite colocar una condición que han de cumplir todos los registros, los que no la cumplan no aparecen en el resultado.

Ejemplo:

```
SELECT Tipo, Modelo FROM Pieza WHERE Precio>3;
```

(3.8.1) relación con el álgebra relacional

La cláusula WHERE es la encargada de implementar la operación de selección del álgebra relacional. Ejemplo:

$\sigma_{\text{nombre}=\text{"Pepe"} \wedge \text{edad}>25}$ (Cliente)

Se corresponde con el SQL:

```
SELECT * FROM Clientes WHERE nombre='Pepe' AND edad>25
```

(3.8.2) operadores de comparación

Se pueden utilizar en la cláusula WHERE, son:

Operador	Significado
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
=	Igual
<>	Distinto
!=	Distinto

Se pueden utilizar tanto para comparar números como para comparar textos y fechas. En el caso de los textos, las comparaciones se hacen en orden alfabético. Sólo que es un orden alfabético estricto. Es decir el orden de los caracteres en la tabla de códigos.

En muchas bases de datos hay problemas con la Ñ y otros símbolos nacionales. No obstante es un problema que tiende a arreglarse en la mayoría de aplicaciones.

(3.8.3) valores lógicos

Son:

Operador	Significado
AND	Devuelve verdadero si las expresiones a su izquierda y derecha son ambas verdaderas
OR	Devuelve verdadero si cualquiera de las dos expresiones a izquierda y derecha del OR, son verdaderas

Operador	Significado
NOT	Invierte la lógica de la expresión que está a su derecha. Si era verdadera, mediante NOT pasa a ser falso.

Ejemplos:

```
/* Obtiene a las personas de entre 25 y 50 años*/  
SELECT nombre,apellidos FROM personas  
WHERE edad>=25 AND edad<=50;  
/*Obtiene a la gente de más de 60 años o de menos de 20*/  
SELECT nombre,apellidos FROM personas  
WHERE edad>60 OR edad<20;
```

(3.8.4) BETWEEN

El operador **BETWEEN** nos permite obtener datos que se encuentren en un rango. Uso:

```
SELECT tipo,modelo,precio FROM piezas  
WHERE precio BETWEEN 3 AND 8;
```

Saca piezas cuyos precios estén entre 3 y 8 (ambos incluidos).

(3.8.5) IN

Permite obtener registros cuyos valores estén en una lista:

```
SELECT tipo,modelo,precio FROM piezas  
WHERE precio IN (3,5, 8);
```

Obtiene piezas cuyos precios sea 3, 5 u 8, sólo uno de esos tres.

(3.8.6) LIKE

Se usa sobre todo con textos, permite obtener registros cuyo valor en un campo cumpla una condición textual. LIKE utiliza una cadena que puede contener estos símbolos:

%	Una serie cualquiera de caracteres
_	Un carácter cualquiera

Ejemplos:

```
/* Selecciona nombres que empiecen por S */  
SELECT nombre FROM personas WHERE nombre LIKE 'A%';  
/*Selecciona las personas cuyo apellido sea Sanchez,  
Senchez, Stnchez,...*/  
SELECT apellido1 FROM Personas WHERE apellido1 LIKE  
'S_nchez';
```

(3.8.7) IS NULL

Devuelve verdadero si una expresión contiene un nulo:

```
SELECT nombre,apellidos FROM personas  
WHERE telefono IS NULL
```

Esa instrucción selecciona a la gente que no tiene teléfono. Se puede usar la expresión IS NOT NULL que devuelve verdadero cuando una determinada columna no es nula

(3.8.8) Precedencia de operadores

A veces las expresiones que se producen en los SELECT son muy extensas y es difícil saber que parte de la expresión se evalúa primero, por ello se indica la siguiente tabla de precedencia (tomada de Oracle):

Orden de precedencia	Operador
1	*(Multiplicar) / (dividir)
2	+ (Suma) - (Resta)
3	(Concatenación)
4	Comparaciones (>, <, !=, ...)
5	IS [NOT] NULL, [NOT]LIKE, IN
6	NOT
7	AND
8	OR

(3.9) ordenación

El orden inicial de los registros obtenidos por un SELECT no guarda más que una relación respecto al orden en el que fueron introducidos. Para ordenar en base a criterios más interesantes, se utiliza la cláusula ORDER BY.

En esa cláusula se coloca una lista de campos que indica la forma de ordenar. Se ordena primero por el primer campo de la lista, si hay coincidencias por el segundo, si ahí también las hay por el tercero, y así sucesivamente.

Se puede colocar las palabras **ASC** O **DESC** (por defecto se toma **ASC**). Esas palabras significan en ascendente (de la A a la Z, de los números pequeños a los

grandes) o en descendente (de la Z a la a, de los números grandes a los pequeños) respectivamente.

Sintaxis completa de SELECT:

```
SELECT expresiones
FROM tabla
[WHERE condición]
ORDER BY listaDeCamposOAlias;
```

(3.10) subconsultas

Se trata de una técnica que permite utilizar el resultado de una tabla SELECT en otra consulta SELECT. Permite solucionar problemas en los que el mismo dato aparece dos veces.

La sintaxis es:

```
SELECT listaExpresiones
FROM tabla
WHERE expresión operador
      (SELECT listaExpresiones
      FROM tabla);
```

Se puede colocar el SELECT dentro de las cláusulas WHERE, HAVING o FROM. El operador puede ser >, <, >=, <=, !=, = o IN.

Ejemplo:

```
SELECT nombre_empleado, paga
FROM empleados
WHERE paga <
      (SELECT paga FROM empleados
      WHERE nombre_empleado='Martina')
;
```

Lógicamente el resultado de la subconsulta debe incluir el campo que estamos analizando. Se pueden realizar esas subconsultas las veces que haga falta:

```
SELECT nombre_empleado, paga
FROM empleados
WHERE paga <
      (SELECT paga FROM empleados
       WHERE nombre_empleado='Martina')
AND paga >
      (SELECT paga FROM empleados WHERE
nombre_empleado='Luis');
```

La última consulta obtiene los empleados cuyas pagas estén entre lo que gana Luís y lo que gana Martina.

Una subconsulta que utilice los valores >,<,>=,... tiene que devolver un único valor, de otro modo ocurre un error. Pero a veces se utilizan consultas del tipo: **mostrar el sueldo y nombre de los empleados cuyo sueldo supera al de cualquier empleado del departamento de ventas.**

La subconsulta necesaria para ese resultado mostraría los sueldos del departamento de ventas. Pero no podremos utilizar un operador de comparación directamente ya que compararíamos un valor con muchos valores. La solución a esto es utilizar instrucciones especiales entre el operador y la consulta.

Esas instrucciones son:

Instrucción	Significado
ANY	Compara con cualquier registro de la consulta. La instrucción es válida si hay un registro en la subconsulta que permite que la comparación sea cierta
ALL	Compara con todos los registros de la consulta. La instrucción resulta cierta si es cierta toda comparación con los registros de la subconsulta
IN	No usa comparador, ya que sirve para comprobar si un valor se encuentra en el resultado de la subconsulta
NOT IN	Comprueba si un valor no se encuentra en una subconsulta

Ejemplo:

```
SELECT nombre, sueldo
FROM empleados
WHERE sueldo >= ALL (SELECT sueldo FROM empleados)
```

Esa consulta obtiene el empleado que más cobra. Otro ejemplo:

```
SELECT nombre FROM empleados  
WHERE dni IN (SELECT dni FROM directivos)
```

En ese caso se obtienen los nombres de los empleados cuyos **dni** están en la tabla de directivos.

Si se necesita comprobar dos columnas en una consulta IN, se hace:

```
SELECT nombre FROM empleados  
WHERE (cod1,cod2) IN (SELECT cod1,cod2 FROM directivos)
```

(3.11) obtener datos de múltiples tablas

Es más que habitual necesitar en una consulta datos que se encuentran distribuidos en varias tablas. Las bases de datos relacionales se basan en que los datos se distribuyen en tablas que se pueden relacionar mediante un campo. Ese campo es el que permite integrar los datos de las tablas.

Por ejemplo si disponemos de una tabla de empleados cuya clave es el **dni** y otra tabla de tareas que se refiere a tareas realizadas por los empleados, es seguro (si el diseño está bien hecho) que en la tabla de tareas aparecerá el dni del empleado para saber qué empleado realizó la tarea.

(3.11.1) producto cruzado o cartesiano de tablas

En el ejemplo anterior si quiere obtener una lista de los datos de las tareas y los empleados, se podría hacer de esta forma:

```
SELECT cod_tarea, descripcion_tarea, dni_empleado,  
nombre_empleado  
FROM tareas,empleados;
```

La sintaxis es correcta ya que, efectivamente, en el apartado **FROM** se pueden indicar varias tareas separadas por comas. Pero eso produce un producto cruzado, aparecerán todos los registros de las tareas relacionados con todos los registros de empleados.

El producto cartesiano a veces es útil para realizar consultas complejas, pero en el caso normal no lo es. necesitamos discriminar ese producto para que sólo aparezcan los registros de las tareas relacionadas con sus empleados correspondientes. A eso se le llama asociar (**join**) tablas

(3.11.2) asociando tablas

La forma de realizar correctamente la consulta anterior (asociado las tareas con los empleados que la realizaron sería:

Sistemas Gestores de Bases de Datos

(Unidad 3) Lenguaje SQL

```
SELECT cod_tarea, descripcion_tarea, dni_empleado,
nombre_empleado
FROM tareas,empleados
WHERE tareas.dni_empleado = empleados.dni;
```

Nótese que se utiliza la notación **tabla.columna** para evitar la ambigüedad, ya que el mismo nombre de campo se puede repetir en ambas tablas. Para evitar repetir continuamente el nombre de la tabla, se puede utilizar un alias de tabla:

```
SELECT a.cod_tarea, a.descripcion_tarea, b.dni_empleado,
b.nombre_empleado
FROM tareas a,empleados b
WHERE a.dni_empleado = b.dni;
```

Al apartado WHERE se le pueden añadir condiciones encadenándolas con el operador AND. Ejemplo:

```
SELECT a.cod_tarea, a.descripcion_tarea
FROM tareas a,empleados b
WHERE a.dni_empleado = b.dni AND
b.nombre_empleado='Javier';
```

Finalmente indicar que se pueden enlazar más de dos tablas a través de sus campos relacionados. Ejemplo:

```
SELECT a.cod_tarea, a.descripcion_tarea,
b.nombre_empleado, c.nombre_utensilio
FROM tareas a,empleados b, utensilios_utilizados c
WHERE a.dni_empleado = b.dni AND a.cod_tarea=c.cod_tarea;
```

(3.11.3) relaciones sin igualdad

A las relaciones descritas anteriormente se las llama relaciones en igualdad (**equijoins**), ya que las tablas se relacionan a través de campos que contienen valores iguales en dos tablas.

Sin embargo no siempre las tablas tienen ese tipo de relación, por ejemplo:

EMPLEADOS		
Empleado	Sueldo	
Antonio	18000	
Marta	21000	
Sonia	15000	

CATEGORIAS		
categoría	Sueldo mínimo	Sueldo máximo
D	6000	11999
C	12000	17999
B	18000	20999
A	20999	80000

En el ejemplo anterior podríamos averiguar la categoría a la que pertenece cada empleado, pero estas tablas poseen una relación que ya no es de igualdad.

La forma sería:

```
SELECT a.empleado, a.sueldo, b.categoria
FROM empleados a, categorias b
WHERE a.sueldo between b.sueldo_minimo and
b.sueldo_maximo;
```

(3.11.4) sintaxis SQL 1999

En la versión SQL de 1999 se ideó una nueva sintaxis para consultar varias tablas. La razón fue separar las condiciones de asociación respecto de las condiciones de selección de registros. Oracle incorpora totalmente esta normativa.

La sintaxis completa es:

```
SELECT tabla1.columna1, tabla1.columna2,...
       tabla2.columna1, tabla2.columna2,... FROM tabla1
[CROSS JOIN tabla2]|
[NATURAL JOIN tabla2]|
[JOIN tabla2 USING(columna)]|
[JOIN tabla2 ON (tabla1.columa=tabla2.columa)]|
```

```
[ LEFT | RIGHT | FULL OUTER JOIN tabla2 ON  
( tabla1.columna=tabla2.columna ) ]
```

Se describen sus posibilidades en los siguientes apartados.

CROSS JOIN

Utilizando la opción CROSS JOIN se realiza un producto cruzado entre las tablas indicadas. Esta orden es equivalente al producto del álgebra relacional. Ejemplo:

- ♦ **álgebra relacional:** $R \times S$
- ♦ **SQL:** SELECT * FROM R CROSS JOIN S;

NATURAL JOIN

Establece una relación de igualdad entre las tablas a través de los campos que tengan el mismo nombre en ambas tablas:

```
SELECT * FROM piezas  
NATURAL JOIN existencias;
```

En ese ejemplo se obtienen los registros de piezas relacionados en existencias a través de los campos que tengan el mismo nombre en ambas tablas.

Equivalencia con el álgebra relacional:

- ♦ **álgebra relacional:** $R \bowtie S$
- ♦ **SQL:** SELECT * FROM R NATURAL JOIN S;

JOIN USING

Permite establecer relaciones indicando qué campo (o campos) común a las dos tablas hay que utilizar:

```
SELECT * FROM piezas  
JOIN existencias USING(tipo,modelo);
```

Equivalencia con el álgebra relacional:

- ♦ **álgebra relacional:**
cliente \bowtie alquiler
dni
- ♦ **SQL:** SELECT * FROM clientes JOIN alquiler USING(dni);

JOIN ON

Permite establecer relaciones cuya condición se establece manualmente, lo que permite realizar asociaciones más complejas o bien asociaciones cuyos campos en las tablas no tienen el mismo nombre:

```
SELECT * FROM piezas  
JOIN existencias ON(piezas.tipo=existencias.tipo AND  
piezas.modelo=existencias.modelo);
```

Equivalencia con el álgebra relacional:

◆ **álgebra relacional:**

localidades ∞ provincias

provincias.id_capital=localidades.id_localidad

◆ **SQL.** SELECT * FROM localidades JOIN provincias
USING(provincias.id_capital=localidades.id_capital);

relaciones externas

La última posibilidad es obtener relaciones laterales o externas (**outer join**). Para ello se utiliza la sintaxis:

```
SELECT * FROM piezas  
LEFT OUTER JOIN existencias  
ON(piezas.tipo=existencias.tipo AND  
piezas.modelo=existencias.modelo);
```

En esta consulta además de las relacionadas, aparecen los datos de los registros de la tabla piezas que no están en existencias. Si el **LEFT** lo cambiamos por un **RIGHT**, aparecerán las existencias no presentes en la tabla piezas (además de las relacionadas en ambas tablas).

La condición **FULL OUTER JOIN** produciría un resultado en el que aparecen los registros no relacionados de ambas tablas.

(Unidad 4)

Funciones

(4.1) funciones

Todos los SGBD implementan funciones para facilitar la creación de consultas complejas. Esas funciones dependen del SGBD que utilizemos, las que aquí se comentan son algunas de las que se utilizan con Oracle.

Todas las funciones reciben datos para poder operar (parámetros) y devuelven un resultado (que depende de los parámetros enviados a la función. Los argumentos se pasan entre paréntesis:

```
nombreFunción(parámetro1[, parámetro2,...])
```

Si una función no precisa parámetros (como **SYSDATE**) no hace falta colocar los paréntesis.

En realidad hay dos tipos de funciones:

- ◆ Funciones que operan con datos de la misma fila
- ◆ Funciones que operan con datos de varias filas diferentes (funciones de agrupación).

Sólo veremos las de primeras (más adelante se comentan las de varias filas).

Nota: tabla DUAL (Oracle)

Oracle proporciona una tabla llamada dual con la que se permiten hacer pruebas. Esa tabla tiene un solo campo (llamado **DUMMY**) y una sola fila de modo que es posible hacer pruebas. Por ejemplo la consulta:

```
SELECT SQRT(5) FROM DUAL;
```

Muestra una tabla con el contenido de ese cálculo (la raíz cuadrada de 5). DUAL es una tabla interesante para hacer pruebas.

(4.2) funciones numéricas

(4.2.1) redondeos

Función	Descripción
ROUND (<i>n,decimales</i>)	Redondea el número al siguiente número con el número de decimales indicado más cercano. ROUND(8.239,2) devuelve 8.3
TRUNC (<i>n,decimales</i>)	Los decimales del número se cortan para que sólo aparezca el número de decimales indicado

(4.2.2) matemáticas

Función	Descripción
MOD (<i>n1,n2</i>)	Devuelve el resto resultado de dividir <i>n1</i> entre <i>n2</i>
POWER (<i>valor,exponente</i>)	Eleva el valor al exponente indicado
SQRT (<i>n</i>)	Calcula la raíz cuadrada de <i>n</i>
SIGN (<i>n</i>)	Devuelve 1 si <i>n</i> es positivo, cero si vale cero y -1 si es negativo
ABS (<i>n</i>)	Calcula el valor absoluto de <i>n</i>
EXP (<i>n</i>)	Calcula <i>eⁿ</i> , es decir el exponente en base <i>e</i> del número <i>n</i>
LN (<i>n</i>)	Logaritmo neperiano de <i>n</i>
LOG (<i>n</i>)	Logaritmo en base 10 de <i>n</i>
SIN (<i>n</i>)	Calcula el seno de <i>n</i> (<i>n</i> tiene que estar en radianes)
COS (<i>n</i>)	Calcula el coseno de <i>n</i> (<i>n</i> tiene que estar en radianes)
TAN (<i>n</i>)	Calcula la tangente de <i>n</i> (<i>n</i> tiene que estar en radianes)
ACOS (<i>n</i>)	Devuelve en radianes el arcocoseno de <i>n</i>
ASIN (<i>n</i>)	Devuelve en radianes el arcoseno de <i>n</i>
ATAN (<i>n</i>)	Devuelve en radianes el arcotangente de <i>n</i>
SINH (<i>n</i>)	Devuelve el seno hiperbólico de <i>n</i>
COSH (<i>n</i>)	Devuelve el coseno hiperbólico de <i>n</i>
TANH (<i>n</i>)	Devuelve la tangente hiperbólica de <i>n</i>

(4.3) funciones de caracteres

(4.3.1) conversión del texto a mayúsculas y minúsculas

Son:

Función	Descripción
LOWER (<i>texto</i>)	Convierte el texto a minúsculas (funciona con los caracteres españoles)
UPPER (<i>texto</i>)	Convierte el texto a mayúsculas
INITCAP (<i>texto</i>)	Coloca la primera letra de cada palabra en mayúsculas

(4.3.2) funciones de transformación

Función	Descripción
RTRIM (<i>texti</i>)	Elimina los espacios a la derecha del texto
LTRIM (<i>texto</i>)	Elimina los espacios a la izquierda que posea el texto
TRIM (<i>texto</i>)	Elimina los espacios en blanco a la izquierda y la derecha del texto y los espacios dobles del interior.
TRIM (<i>caracteres FROM texto</i>)	Elimina del texto los caracteres indicados. Por ejemplo TRIM('h' FROM nombre) elimina las haches de la columna <i>nombre</i> que estén a la izquierda y a la derecha
SUBSTR (<i>texto,n[,m]</i>)	Obtiene los <i>m</i> siguientes caracteres del texto a partir de la posición <i>n</i> (si <i>m</i> no se indica se cogen desde <i>n</i> hasta el final).
LENGTH (<i>texto</i>)	Obtiene el tamaño del texto
INSTR (<i>texto, textoBuscado [,posInicial [, nAparición]]</i>)	Obtiene la posición en la que se encuentra el texto buscado en el texto inicial. Se puede empezar a buscar a partir de una posición inicial concreta e incluso indicar el número de aparición del texto buscado. Ejemplo, si buscamos la letra <i>a</i> y ponemos 2 en <i>nAparición</i> , devuelve la posición de la segunda letra <i>a</i> del texto). Si no lo encuentra devuelve 0
REPLACE (<i>texto, textoABuscar, [textoReemplazo]</i>)	Buscar el texto a buscar en un determinado texto y lo cambia por el indicado como texto de reemplazo. Si no se indica texto de reemplazo, entonces esta función elimina el texto a buscar
LPAD (<i>texto, anchuraMáxima, [caracterDeRelleno]</i>) RPAD (<i>texto, anchuraMáxima, [caracterDeRelleno]</i>)	Rellena el texto a la izquierda (LPAD) o a la derecha (RPAD) con el carácter indicado para ocupar la anchura indicada. Si el texto es más grande que la anchura indicada, el texto se recorta. Si no se indica carácter de relleno se rellenará el espacio marcado con espacios en blanco.

	Ejemplo: <i>LPAD('Hola',10,'-')</i> da como resultado <i>-----Hola</i>
--	---

(4.4) funciones de trabajo con nulos

Permiten definir valores a utilizar en el caso de que las expresiones tomen el valor nulo.

Función	Descripción
<i>NVL(valor,sustituto)</i>	Si el <i>valor</i> es NULL, devuelve el valor <i>sustituto</i> ; de otro modo, devuelve valor
<i>NVL2(valor,sustituto1,sustituto2)</i>	Variante de la anterior, devuelve el valor <i>sustituto1</i> si <i>valor</i> no es nulo. Si <i>valor</i> es nulo devuelve el <i>sustituto2</i>

(4.5) funciones de fecha y manejo de fechas

Las fechas se utilizan muchísimo en todas las bases de datos. Oracle proporciona dos tipos de datos para manejar fechas, los tipos **DATE** y **TIMESTAMP**. En el primer caso se almacena una fecha concreta (que incluso puede contener la hora), en el segundo caso se almacena un instante de tiempo más concreto que puede incluir incluso fracciones de segundo.

Hay que tener en cuenta que a los valores de tipo fecha se les pueden sumar números y se entendería que esta suma es de días. Si tiene decimales entonces se suman días, horas, minutos y segundos. La diferencia entre dos fechas también obtiene un número de días.

(4.5.1) intervalos

Los intervalos son datos relacionados con las fechas en sí, pero que no son fechas. Hay dos tipos de intervalos el **INTERVAL DAY TO SECOND** que sirve para representar días, horas, minutos y segundos; y el **INTERVAL YEAR TO MONTH** que representa años y meses.

Para los intervalos de año a mes los valores se pueden indicar de estas formas:

```
/* 123 años y seis meses */  
INTERVAL '123-6' YEAR(4) TO MONTH  
/* 123 años */  
INTERVAL '123' YEAR(4) TO MONTH  
/* 6 meses */  
INTERVAL '6' MONTH(3) TO MONTH
```

La precisión en el caso de indicar tanto años como meses, se indica sólo en el año. En intervalos de días a segundos los intervalos se pueden indicar como:

```

/* 4 días 10 horas 12 minutos y 7 con 352 segundos */
INTERVAL '4 10:12:7,352' DAY TO SECOND(3)
/* 4 días 10 horas 12 minutos */
INTERVAL '4 10:12' DAY TO MINUTE
/* 4 días 10 horas */
INTERVAL '4 10' DAY TO HOUR
/* 4 días*/
INTERVAL '4' DAY
/*10 horas*/
INTERVAL '10' HOUR
/*25 horas*/
INTERVAL '253' HOUR
/*12 minutos*/
INTERVAL '12' MINUTE
/*30 segundos */
INTERVAL '30' SECOND
/*8 horas y 50 minutos */
INTERVAL ('8:50') HOUR TO MINUTE;
/*7 minutos 6 segundos*/
INTERVAL ('7:06') MINUTE TO SECOND;
/*8 horas 7 minutos 6 segundos*/
INTERVAL ('8:07:06') HOUR TO SECOND;

```

Esos intervalos se pueden sumar a valores de tipo DATE o TIMESTAMP

(4.5.2) obtener la fecha y hora actual

Función	Descripción
SYSDATE	Obtiene la fecha y hora actuales
SYSTIMESTAMP	Obtiene la fecha y hora actuales en formato TIMESTAMP

(4.5.3) calcular fechas

Función	Descripción
ADDMONTHS(<i>fecha</i>,<i>n</i>)	Añade a la fecha el número de meses indicado por <i>n</i>
MONTHS_BETWEEN(<i>fecha1</i>, <i>fecha2</i>)	Obtiene la diferencia en meses entre las dos fechas (puede ser decimal)

Función	Descripción
NEXT_DAY (<i>fecha</i> , <i>día</i>)	Indica cual es el día que corresponde a añadir a la fecha el día indicado. El día puede ser el texto ' Lunes ', ' Martes ', ' Miércoles ',... (si la configuración está en español) o el número de día de la semana (1=lunes, 2=martes,...)
LAST_DAY (<i>fecha</i>)	Obtiene el último día del mes al que pertenece la fecha. Devuelve un valor DATE
EXTRACT (<i>valor</i> FROM <i>fecha</i>)	Extrae un valor de una fecha concreta. El valor puede ser day (día), month (mes), year (año), etc.
GREATEST (<i>fecha1</i> , <i>fecha2</i> ,...)	Devuelve la fecha más moderna la lista
LEAST (<i>fecha1</i> , <i>fecha2</i> ,...)	Devuelve la fecha más antigua la lista
ROUND (<i>fecha</i> [, <i>formato</i>])	Redondea la fecha al valor de aplicar el formato a la fecha. El formato puede ser: <ul style="list-style-type: none"> ● 'YEAR' Hace que la fecha refleje el año completo ● 'MONTH' Hace que la fecha refleje el mes completo más cercano a la fecha ● 'HH24' Redondea la hora a las 00:00 más cercanas ● 'DAY' Redondea al día más cercano
TRUNC (<i>fecha</i> [<i>formato</i>])	Igual que el anterior pero trunca la fecha en lugar de redondearla.

(4.6) funciones de conversión

Oracle es capaz de convertir datos automáticamente a fin de que la expresión final tenga sentido. En ese sentido son fáciles las conversiones de texto a número y viceversa. Ejemplo:

```
SELECT 5+'3' FROM DUAL /*El resultado es 8 */
SELECT 5 || '3' FROM DUAL /* El resultado es 53 */
```

También ocurre eso con la conversión de textos a fechas. De hecho es forma habitual de asignar fechas.

Pero en diversas ocasiones querremos realizar conversiones explícitas.

(4.6.1) TO_CHAR

Obtiene un texto a partir de un número o una fecha. En especial se utiliza con fechas (ya que de número a texto se suele utilizar de forma implícita).

(4.7) fechas

En el caso de las fechas se indica el formato de conversión, que es una cadena que puede incluir estos símbolos (en una cadena de texto):

YY	Año en formato de dos cifras
YYYY	Año en formato de cuatro cifras
MM	Mes en formato de dos cifras
MON	Las tres primeras letras del mes
MONTH	Nombre completo del mes
DY	Día de la semana en tres letras
DAY	Día completo de la semana
DD	Día en formato de dos cifras
D	Día de la semana (del 1 al 7)
Q	Semestre
WW	Semana del año
D	Día de la semana (del 1 al 7)
DDD	Día del año
AM	Indicador AM
PM	Indicador PM
HH12	Hora de 1 a 12
HH24	Hora de 0 a 23
MI	Minutos (0 a 59)
SS	Segundos (0 a 59)
SSSS	Segundos desde medianoche
/ . , ; ' "	Posición de los separadores, donde se pongan estos símbolos aparecerán en el resultado

Ejemplos:

```
SELECT TO_CHAR(SYSDATE, 'DD/MONTH/YYYY, DAY HH:MI:SS')
FROM DUAL
/* Sale : 16/AGOSTO /2004, LUNES 08:35:15, por ejemplo
```

(4.8) números

Para convertir números a textos se usa esta función cuando se desean características especiales. En ese caso en el formato se pueden utilizar estos símbolos:

Símbolo	Significado
9	Posición del número

Símbolo	Significado
0	Posición del número (muestra ceros)
\$	Formato dólar
L	Símbolo local de la moneda
S	Hace que aparezca el símbolo del signo
D	Posición del símbolo decimal (en español, la coma)
G	Posición del separador de grupo (en español el punto)

(4.8.1) TO_NUMBER

Convierte textos en números. Se indica el formato de la conversión (utilizando los mismos símbolos que los comentarios anteriormente).

(4.8.2) TO_DATE

Convierte textos en fechas. Como segundo parámetro se utilizan los códigos de formato de fechas comentados anteriormente.

(4.9) función DECODE

Función que permite realizar condiciones en una consulta. Se evalúa una expresión y se colocan a continuación pares valor, resultado de forma que si se la expresión equivale al valor, se obtiene el resultado indicado. Se puede indicar un último parámetro con el resultado a efectuar en caso de no encontrar ninguno de los valores indicados.

Sintaxis:

```
DECODE(expresión, valor1, resultado1  
      [, valor2, resultado2, ...]  
      [, valorPordefecto])
```

Ejemplo:

```
SELECT  
DECODE(cotizacion, 1, salario*0.85,  
      2, salario * 0.93,  
      3, salario * 0.96,  
      salario)  
FROM empleados;
```

En el ejemplo dependiendo de la cotización se muestra rebajado el salario: un 85% si la cotización es uno, un 93 si es dos y un 96 si es tres. Si la cotización no es ni uno ni dos ni tres, sencillamente se muestra el salario sin más.

(Unidad 5)

Cláusulas avanzadas de selección

(5.1) agrupaciones

Es muy común utilizar consultas en las que se desee agrupar los datos a fin de realizar cálculos en vertical, es decir calculados a partir de datos de distintos registros.

Para ello se utiliza la cláusula GROUP BY que permite indicar en base a qué registros se realiza la agrupación. Con GROUP BY la instrucción SELECT queda de esta forma:

```
SELECT listaDeExpresiones
FROM listaDeTablas
[JOIN tablasRelacionadasYCondicionesDeRelación]
[WHERE condiciones]
[GROUP BY grupos]
[HAVING condiciones de grupo]
[ORDER BY columnas];
```

En el apartado GROUP BY, se indican las columnas por las que se agrupa. La función de este apartado es crear un único registro por cada valor distinto en las columnas del grupo. Si por ejemplo agrupamos en base a las columnas **tipo** y **modelo** en una tabla de **existencias**, se creará un único registro por cada tipo y modelo distintos:

```
SELECT tipo,modelo
FROM existencias
GROUP BY tipo,modelo;
```

Si la tabla de existencias sin agrupar es:

Sistemas Gestores de Bases de Datos
(Unidad 5) Cláusulas avanzadas de selección

TI	MODELO	N_ALMACEN	CANTIDAD
AR	6	1	2500
AR	6	2	5600
AR	6	3	2430
AR	9	1	250
AR	9	2	4000
AR	9	3	678
AR	15	1	5667
AR	20	3	43
BI	10	2	340
BI	10	3	23
BI	38	1	1100
BI	38	2	540
BI	38	3	

La consulta anterior creará esta salida:

TI	MODELO
AR	6
AR	9
AR	15
AR	20
BI	10
BI	38

Es decir es un resumen de los datos anteriores. Los datos *n_almacen* y *cantidad* no están disponibles directamente ya que son distintos en los registros del mismo grupo. Sólo se pueden utilizar desde funciones (como se verá ahora). Es decir esta consulta es errónea:

```
SELECT tipo,modelo, cantidad
FROM existencias
GROUP BY tipo,modelo;

SELECT tipo,modelo, cantidad
      *

ERROR en línea 1:
ORA-00979: no es una expresión GROUP BY
```

(5.1.1) funciones de cálculo con grupos

Lo interesante de la creación de grupos es las posibilidades de cálculo que ofrece. Para ello se utilizan funciones que permiten trabajar con los registros de un grupo son:

Función	Significado
COUNT(*)	Cuenta los elementos de un grupo. Se utiliza el asterisco para no tener que indicar un nombre de columna concreto, el resultado es el mismo para cualquier columna
SUM(<i>expresión</i>)	Suma los valores de la expresión
AVG(<i>expresión</i>)	Calcula la media aritmética sobre la expresión indicada
MIN(<i>expresión</i>)	Mínimo valor que toma la expresión indicada
MAX(<i>expresión</i>)	Máximo valor que toma la expresión indicada
STDDEV(<i>expresión</i>)	Calcula la desviación estándar
VARIANCE(<i>expresión</i>)	Calcula la varianza

Todos esos valores se calculan para cada elemento del grupo, así la expresión:

```
SELECT tipo,modelo, cantidad, SUM(Cantidad)
FROM existencias
GROUP BY tipo,modelo;
```

Obtiene este resultado:

TI	MODELO	SUM(CANTIDAD)
AR	6	10530
AR	9	4928
AR	15	5667
AR	20	43
BI	10	363
BI	38	1740

Se suman las cantidades para cada grupo

(5.1.2) condiciones HAVING

A veces se desea restringir el resultado de una expresión agrupada, por ejemplo con:

```
SELECT tipo,modelo, cantidad, SUM(Cantidad)
FROM existencias
WHERE SUM(Cantidad)>500
GROUP BY tipo,modelo;
```

Pero Oracle devolvería este error:

```
WHERE SUM(Cantidad)>500
      *
ERROR en línea 3:
ORA-00934: función de grupo no permitida aquí
```

La razón es que Oracle calcula primero el WHERE y luego los grupos; por lo que esa condición no la puede realizar al no estar establecidos los grupos.

Por ello se utiliza la cláusula HAVING, que se ejecuta una vez realizados los grupos. Se usaría de esta forma:

```
SELECT tipo,modelo, cantidad, SUM(Cantidad)
FROM existencias
GROUP BY tipo,modelo
HAVING SUM(Cantidad)>500;
```

Eso no implica que no se pueda usar WHERE. Ésta expresión sí es válida:

```
SELECT tipo,modelo, cantidad, SUM(Cantidad)
FROM existencias
WHERE tipo!='AR'
GROUP BY tipo,modelo
HAVING SUM(Cantidad)>500;
```

En definitiva, el orden de ejecución de la consulta marca lo que se puede utilizar con WHERE y lo que se puede utilizar con HAVING:

Para evitar problemas estos podrían ser los pasos en la ejecución de una instrucción de agrupación por parte del gestor de bases de datos:

- (1) Seleccionar las filas deseadas utilizando **WHERE**. Esta cláusula eliminará columnas en base a la condición indicada
- (2) Se establecen los grupos indicados en la cláusula **GROUP BY**
- (3) Se calculan los valores de las funciones de totales (**COUNT, SUM, AVG,...**)
- (4) Se filtran los registros que cumplen la cláusula **HAVING**
- (5) El resultado se ordena en base al apartado **ORDER BY**.

(5.2) combinaciones especiales

(5.2.1) uniones

La palabra **UNION** permite añadir el resultado de un **SELECT** a otro **SELECT**. Para ello ambas instrucciones tienen que utilizar el mismo número y tipo de columnas. Ejemplo:

```
SELECT nombre FROM provincias
UNION
SELECT nombre FROM comunidades
```

El resultado es una tabla que contendrá nombres de provincia y de comunidades. Es decir, **UNION** crea una sola tabla con registros que estén presentes en cualquiera de las consultas. Si están repetidas sólo aparecen una vez, para mostrar los duplicados se utiliza **UNION ALL** en lugar de la palabra **UNION**.

Es muy importante señalar que tanto ésta cláusula como el resto de combinaciones especiales, requieren en los dos **SELECT** que unen el mismo tipo de columnas (y en el mismo orden).

(5.2.2) intersecciones

De la misma forma, la palabra **INTERSECT** permite unir dos consultas **SELECT** de modo que el resultado serán las filas que estén presentes en ambas consultas.

Ejemplo; tipos y modelos de piezas que se encuentren sólo en los almacenes 1 y 2:

```
SELECT tipo,modelo FROM existencias
WHERE n_almacen=1
INTERSECT
SELECT tipo,modelo FROM existencias
WHERE n_almacen=2
```

(5.2.3) diferencia

Con **MINUS** también se combinan dos consultas **SELECT** de forma que aparecerán los registros del primer **SELECT** que no estén presentes en el segundo.

Ejemplo; tipos y modelos de piezas que se encuentren el almacén 1 y no en el 2

```
SELECT tipo,modelo FROM existencias
WHERE n_almacen=1
MINUS
SELECT tipo,modelo FROM existencias
WHERE n_almacen=2
```

Se podrían hacer varias combinaciones anidadas (una unión cuyo resultado se intersectará con otro SELECT por ejemplo), en ese caso es conveniente utilizar paréntesis para indicar qué combinación se hace primero:

```
( SELECT....
....
UNION
SELECT....
...
)
MINUS
SELECT.... /* Primero se hace la unión y luego la
diferencia*/
```

(5.3) vistas

(5.3.1) introducción

Una vista no es más que una consulta almacenada a fin de utilizarla tantas veces como se desee. Una vista no contiene datos sino la instrucción **SELECT** necesaria para crear la vista, eso asegura que los datos sean coherentes al utilizar los datos almacenados en las tablas. Por todo ello, las vistas gastan muy poco espacio de disco.

Las vistas se emplean para:

- ◆ Realizar consultas complejas más fácilmente, ya que permiten dividir la consulta en varias partes
- ◆ Proporcionar tablas con datos completos
- ◆ Utilizar visiones especiales de los datos
- ◆ Ser utilizadas como tablas que resumen todos los datos
- ◆ Ser utilizadas como cursores de datos en los lenguajes procedimentales (como PL/SQL)

Hay dos tipos de vistas:

- ◆ **Simples.** Las forman una sola tabla y no contienen funciones de agrupación. Su ventaja es que permiten siempre realizar operaciones DML sobre ellas.
- ◆ **Complejas.** Obtienen datos de varias tablas, pueden utilizar funciones de agrupación. No siempre permiten operaciones DML.

(5.3.2) creación de vistas

Sintaxis:

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW vista  
    [(alias[, alias2...])]  
AS consultaSELECT  
[WITH CHECK OPTION [CONSTRAINT restricción]]  
[WITH READ ONLY [CONSTRAINT restricción]]
```

- ◆ **OR REPLACE.** Si la vista ya existía, la cambia por la actual
- ◆ **FORCE.** Crea la vista aunque los datos de la consulta SELECT no existan
- ◆ **vista.** Nombre que se le da a la vista
- ◆ **alias.** Lista de alias que se establecen para las columnas devueltas por la consulta SELECT en la que se basa esta vista. El número de alias debe coincidir con el número de columnas devueltas por SELECT.
- ◆ **WITH CHECK OPTION.** Hace que sólo las filas que se muestran en la vista puedan ser añadidas (**INSERT**) o modificadas (**UPDATE**). La **restricción** que sigue a esta sección es el nombre que se le da a esta restricción de tipo **CHECK OPTION**.
- ◆ **WITH READ ONLY.** Hace que la vista sea de sólo lectura. Permite grabar un nombre para esta restricción.

Lo bueno de las vistas es que tras su creación se utilizan como si fueran una tabla.

Ejemplo:

```
CREATE VIEW resumen
/* alias */
(id_localidad, localidad, poblacion, n_provincia,
provincia,
superficie, capital_provincia,
id_comunidad, comunidad, capital_comunidad)

AS
( SELECT l.id_localidad, l.nombre, l.poblacion,
        n_provincia, p.nombre, p.superficie, l2.nombre,
        id_comunidad, c.nombre, l3.nombre
  FROM localidades l
  JOIN provincias p USING (n_provincia)
  JOIN comunidades c USING (id_comunidad)
  JOIN localidades l2 ON (p.id_capital=l2.id_localidad)
  JOIN localidades l3 ON (c.id_capital=l3.id_localidad)
)

SELECT DISTINCT (comunidad, capital_comunidad) FROM
resumen;
```

La creación de la vista del ejemplo es compleja ya que hay relaciones complicadas, pero una vez creada la vista, se le pueden hacer consultas como si se tratara de una tabla normal. Incluso se puede utilizar el comando **DESCRIBE** sobre la vista para mostrar la estructura de los campos que forman la vista.

(5.3.3) mostrar la lista de vistas

La vista del diccionario de datos de Oracle **USER_VIEWS** permite mostrar una lista de todas las vistas que posee el usuario actual. Es decir, para saber qué vistas hay disponibles se usa:

```
SELECT * FROM USER_VIEWS;
```

La columna **TEXT** de esa vista contiene la sentencia SQL que se utilizó para crear la vista (sentencia que es ejecutada cada vez que se invoca a la vista).

(5.3.4) borrar vistas

Se utiliza el comando DROP VIEW:

```
DROP VIEW nombreDeVista;
```

(Unidad 6)

DML

(6.1) introducción

Es una de las partes fundamentales del lenguaje SQL. El DML (*Data Manipulation Language*) lo forman las instrucciones capaces de modificar los datos de las tablas. Al conjunto de instrucciones DML que se ejecutan consecutivamente, se las llama **transacciones** y se pueden anular todas ellas o aceptar, ya que una instrucción DML no es realmente efectuada hasta que no se acepta (**commit**).

En todas estas consultas, el único dato devuelto por Oracle es el número de registros que se han modificado.

(6.2) inserción de datos

La adición de datos a una tabla se realiza mediante la instrucción **INSERT**. Su sintaxis fundamental es:

```
INSERT INTO tabla [(listaDeCampos)]  
VALUES (valor1 [,valor2 ...])
```

La **tabla** representa la tabla a la que queremos añadir el registro y los valores que siguen a VALUES son los valores que damos a los distintos campos del registro. Si no se especifica la lista de campos, la lista de valores debe seguir el orden de las columnas según fueron creados (es el orden de columnas según las devuelve el comando **DESCRIBE**).

La lista de campos a rellenar se indica si no queremos rellenar todos los campos. Los campos no rellenados explícitamente con la orden INSERT, se rellenan con su valor por defecto (**DEFAULT**) o bien con NULL si no se indicó valor alguno. Si algún campo tiene restricción de obligatoriedad (**NOT NULL**), ocurrirá un error si no rellenamos el campo con algún valor.

Por ejemplo, supongamos que tenemos una tabla de clientes cuyos campos son: dni, nombre, apellido1, apellido2, localidad y dirección; supongamos que ese es el orden de creación de los campos de esa tabla y que la localidad tiene como valor por defecto **Palencia** y la dirección no tiene valor por defecto. En ese caso estas dos instrucciones son equivalentes:

Sistemas Gestores de Bases de Datos

(Unidad 5) Cláusulas avanzadas de selección

```
INSERT INTO clientes
VALUES('11111111','Pedro','Gutiérrez',
'Crespo',DEFAULT,NULL);

INSERT INTO clientes(dni,nombre,apellido1,apellido2)
VALUES('11111111','Pedro','Gutiérrez','Crespo')
```

Son equivalentes puesto que en la segunda instrucción los campos no indicados se rellenan con su valor por defecto y la dirección no tiene valor por defecto. La palabra DEFAULT fuerza a utilizar ese valor por defecto.

El uso de los distintos tipos de datos debe de cumplir los requisitos ya comentados en temas anteriores.

(6.2.1) relleno de registros a partir de filas de una consulta

Hay un tipo de consulta, llamada de adición de datos, que permite rellenar datos de una tabla copiando el resultado de una consulta.

Ese relleno se basa en una consulta SELECT que poseerá los datos a añadir. Lógicamente el orden de esos campos debe de coincidir con la lista de campos indicada en la instrucción INDEX. Sintaxis:

```
INSERT INTO tabla (campo1, campo2,...)
SELECT campoCompatibleCampo1, campoCompatibleCampo2,...
FROM tabla(s)
[...otras cláusulas del SELECT...]
```

Ejemplo:

```
INSERT INTO clientes2004 (dni, nombre, localidad,
direccion)
SELECT dni, nombre, localidad, direccion
FROM clientes
WHERE problemas=0;
```

(6.3) actualización de registros

La modificación de los datos de los registros lo implementa la instrucción UPDATE. Sintaxis:

```
UPDATE tabla
SET columna1=valor1 [,columna2=valor2...]
[WHERE condición]
```

Se modifican las columnas indicadas en el apartado SET con los valores indicados. La cláusula WHERE permite especificar qué registros serán modificados.

Ejemplos:

```
UPDATE clientes SET provincia='Ourense'  
WHERE provincia='Ourense';  
  
UPDATE productos SET precio=precio*1.16;
```

El primer dato actualiza la provincia de los clientes de Ourense para que aparezca como Ourense. El segundo UPDATE incrementa los precios en un 16%. La expresión para el valor puede ser todo lo compleja que se desee:

```
UPDATE partidos SET fecha= NEXT_DAY(SYSDATE, 'Martes')  
WHERE fecha=SYSDATE;
```

Incluso se pueden utilizar subconsultas:

```
UPDATE empleados  
SET puesto_trabajo=(SELECT puesto_trabajo  
FROM empleados  
WHERE id_empleado=12)  
WHERE seccion=23;
```

Esta consulta coloca a todos los empleados de la sección 23 el mismo puesto de trabajo que el empleado número 12. Este tipo de actualizaciones sólo son válidas si el **subselect** devuelve un único valor, que además debe de ser compatible con la columna que se actualiza.

Hay que tener en cuenta que las actualizaciones no pueden saltarse las reglas de integridad que posean las tablas.

(6.4) borrado de registros

Se realiza mediante la instrucción DELETE:

```
DELETE [FROM] tabla  
[WHERE condición]
```

Es más sencilla que las anteriores, elimina los registros de la tabla que cumplan la condición indicada. Ejemplos:

```
DELETE FROM empleados  
WHERE seccion=23;
```

```
DELETE FROM empleados
WHERE id_empleado IN (SELECT id_empleado FROM
errores_graves);
```

Hay que tener en cuenta que el borrado de un registro no puede provocar fallos de integridad y que la opción de **integridad ON DELETE CASCADE** hace que no sólo se borren los registros indicados en el SELECT, sino todos los relacionados.

(6.5) transacciones

Como se ha comentado anteriormente, una transacción está formada por una serie de instrucciones DML. Una transacción comienza con la primera instrucción DML que se ejecute y finaliza con alguna de estas circunstancias:

- ◆ Una operación **COMMIT** o **ROLLBACK**
- ◆ Una instrucción DDL (como **ALTER TABLE** por ejemplo)
- ◆ Una instrucción DCL (como **GRANT**)
- ◆ El usuario abandona la sesión
- ◆ Caída del sistema

Hay que tener en cuenta que cualquier instrucción DDL o DCL da lugar a un COMMIT implícito, es decir todas las instrucciones DML ejecutadas hasta ese instante pasan a ser definitivas.

(6.5.2) COMMIT

La instrucción COMMIT hace que los cambios realizados por la transacción sean definitivos, irrevocables. Sólo se debe utilizar si estamos de acuerdo con los cambios, conviene asegurarse mucho antes de realizar el COMMIT ya que las instrucciones ejecutadas pueden afectar a miles de registros.

Además el cierre correcto de la sesión da lugar a un COMMIT, aunque siempre conviene ejecutar explícitamente esta instrucción a fin de asegurarnos de lo que hacemos.

(6.5.3) ROLLBACK

Esta instrucción regresa a la instrucción anterior al inicio de la transacción, normalmente el último COMMIT, la última instrucción DDL o DCL o al inicio de sesión. Anula definitivamente los cambios, por lo que conviene también asegurarse de esta operación.

Un abandono de sesión incorrecto o un problema de comunicación o de caída del sistema dan lugar a un ROLLBACK implícito.

(6.5.4) AUTOCOMMIT

No es una instrucción SQL. Se trata de una capacidad añadida que proporciona **SQL*Plus** o **iSQL*Plus** en Oracle (otros gestores de bases de datos también tienen ésta característica).

Por defecto es una opción deshabilitada, se puede comprobar haciendo uso de la instrucción **SHOW AUTOCOMMIT**. Ésta opción mostraría la situación actual del AUTOCOMMIT. En estado **OFF**, está desactivado; lo que significa que las transacciones se validen explícitamente (con COMMIT o ROLLBACK). En ON, se ejecuta un **COMMIT** implícito (e invisible) cada vez que se ejecuta una instrucción DML. Es decir, cada **UPDATE**, **DELETE** o **INSERT** es definitivo.

Para cambiar el estado del AUTOCOMMIT, se usa el comando **SET AUTOCOMMIT**, seguido de ON u OFF.

(6.5.5) estado de los datos durante la transacción

Si se inicia una transacción usando comandos DML hay que tener en cuenta que:

- ◆ Se puede volver a la instrucción anterior a la transacción cuando se desee
- ◆ Las instrucciones de consulta SELECT realizadas por el usuario que inició la transacción muestran los datos ya modificados por las instrucciones DML
- ◆ El resto de usuarios ven los datos tal cual estaban antes de la transacción, de hecho los registros afectados por la transacción aparecen bloqueados hasta que la transacción finalice. Esos usuarios no podrán modificar los valores de dichos registros.
- ◆ Tras la transacción todos los usuarios ven los datos tal cual quedan tras el fin de transacción. Los bloqueos son liberados y los puntos de ruptura borrados.

(Unidad 7)

DDL

(7.1) introducción

El DDL es la parte del lenguaje que realiza la función de definición de datos. Fundamentalmente se encarga de la creación de esquemas, tablas y vistas. Por ahora veremos como crear tablas.

Cada usuario de una base de datos posee un **esquema**. El esquema suele tener el mismo nombre que el usuario y sirve para almacenar los objetos de esquema, es decir los objetos que posee el usuario.

Esos objetos pueden ser: tablas, vistas, índices y otros objetos relacionados con la definición de la base de datos. Los objetos son manipulados y creados por los usuarios. En principio sólo los administradores y los usuarios propietarios pueden acceder a cada objeto, salvo que se modifiquen los privilegios del objeto para permitir el acceso a otros usuarios.

Hay que tener en cuenta que **ninguna instrucción DDL puede ser anulada por una instrucción ROLLBACK** (la instrucción ROLLBACK está relacionada con el uso de transacciones que se comentarán más adelante) por lo que hay que tener mucha precaución a la hora de utilizarlas. Es decir, las instrucciones DDL son definitivas.

(7.2) creación de bases de datos

Esta es una tarea administrativa que se comentará más profundamente en otros temas. Por ahora sólo se comenta de forma simple. Crear la base de datos implica indicar los archivos y ubicaciones que se utilizarán para la misma, además de otras indicaciones técnicas y administrativas que no se comentarán en este tema.

Lógicamente sólo es posible crear una base de datos si se tienen privilegios DBA (**DataBase Administrator**) o **SYSDBA**.

El comando SQL de creación de una base de datos es **CREATE DATABASE**. Este comando crea una base de datos con el nombre que se indique. Ejemplo:

```
CREATE DATABASE prueba;
```

Pero normalmente se indican más parámetros. Ejemplo:

```
CREATE DATABASE prueba
  LOGFILE prueba.log
  MAXLOGFILES 25
  MAXINSTANCES 10
  ARCHIVELOG
  CHARACTER SET WIN1214
  NATIONAL CHARACTER SET UTF8
  DATAFILE prueba1.dbf AUTOEXTEND ON MAXSIZE 500MB;
```

(7.3) objetos de la base de datos

Según los estándares actuales, una base de datos es un conjunto de objetos pensados para gestionar datos. Estos objetos están contenidos en **esquemas**, los esquemas suelen estar asociados al perfil de un usuario en particular.

En el estándar SQL existe el concepto de **catálogo** que sirve para almacenar esquemas. Así el nombre completo de un objeto vendría dado por:

```
catálogo.esquema.objeto
```

Si no se indica el catálogo se toma el catálogo por defecto. Si no se indica el esquema se entiende que el objeto está en el esquema actual. En Oracle, cuando se crea un usuario, se crea un esquema cuyo nombre es el del usuario.

(7.4) creación de tablas

(7.4.1) nombre de las tablas

Deben cumplir las siguientes reglas (reglas de Oracle, en otros SGBD podrían cambiar):

- Deben comenzar con una letra
- No deben tener más de 30 caracteres
- Sólo se permiten utilizar letras del alfabeto (inglés), números o el signo de subrayado (también el signo \$ y #, pero esos se utilizan de manera especial por lo que no son recomendados)
- No puede haber dos tablas con el mismo nombre para el mismo esquema (pueden coincidir los nombres si están en distintos esquemas)
- No puede coincidir con el nombre de una palabra reservada SQL (por ejemplo no se puede llamar SELECT a una tabla)
- En el caso de que el nombre tenga espacios en blanco o caracteres nacionales (permitido sólo en algunas bases de datos), entonces se suele entrecomillar (en Oracle con comillas dobles, en SQLServer con comillas simples). En el estándar SQL 99 (respetado por Oracle) se pueden utilizar comillas dobles al poner el nombre de la

tabla a fin de hacerla sensible a las mayúsculas (se diferenciará entre **“FACTURAS”** y **“Facturas”**)

(7.4.2) orden CREATE TABLE

Es la orden SQL que permite crear una tabla. Por defecto será almacenada en el espacio y esquema del usuario que crea la tabla. Sintaxis:

```
CREATE TABLE [esquema.] nombreDeTabla  
(nombreDeLaColumna1 tipoDeDatos [, ...]);
```

Ejemplo:

```
CREATE TABLE proveedores (nombre VARCHAR2(25));
```

Crea una tabla con un solo campo de tipo **VARCHAR2**.

Sólo se podrá crear la tabla si el usuario posee los permisos necesarios para ello. Si la tabla pertenece a otro esquema (suponiendo que el usuario tenga permiso para grabar tablas en ese otro esquema), se antepone al nombre de la tabla, el nombre del esquema:

```
CREATE TABLE otroUsuario.proveedores (nombre  
VARCHAR2(25));
```

(7.5) consultar las tablas del usuario

(7.5.1) consultar el diccionario de datos

Todas las bases de datos poseen un diccionario de datos que contiene la lista de tablas. En cada sistema de base de datos se accede a esos datos de una forma. En el caso de **Oracle**, hay una tabla en el diccionario de datos que se llama **USER_TABLES** y que contiene una lista de las tablas del usuario actual. Así para sacar una lista de las tablas, se usa:

```
SELECT * FROM USER_TABLES;
```

Hay diversas columnas que muestran datos sobre cada tabla, entre ellas la columna **TABLES_NAME** muestra el nombre de cada tabla del usuario; el resto muestra información sobre el almacenamiento de la tabla. La tabla **ALL_TABLES** mostrará una lista de todas las tablas accesibles por el usuario. Finalmente **DBA_TABLES** es una tabla que contiene todas las tablas del sistema; esto es accesible sólo por el usuario administrador (**DBA**).

(7.5.2) orden DESCRIBE

El comando DESCRIBE, permite obtener la estructura de una tabla. Ejemplo:

```
DESCRIBE existencias;
```

Y aparecerán los campos de la tabla proveedores. Esta instrucción no es parte del SQL estándar, pero casi es considerada así ya que casi todos los SGBD la utilizan.

El resultado de la orden anterior (en Oracle) sería:

Nombre	¿Nulo?	Tipo
N_ALMACEN	NOT NULL	NUMBER(2)
TIPO	NOT NULL	VARCHAR2(2)
MODELO	NOT NULL	NUMBER(2)
CANTIDAD		NUMBER(7)

(7.6) borrar tablas

La orden **DROP TABLE** seguida del nombre de una tabla, permite eliminar la tabla en cuestión.

Al borrar una tabla:

- Desaparecen todos los datos
- Cualquier vista y sinónimo referente a la tabla seguirá existiendo, pero ya no funcionará (conviene eliminarlos)
- Las transacciones pendientes son aceptadas (**COMMIT**), en aquellas bases de datos que tengan la posibilidad de utilizar transacciones.
- Lógicamente, sólo se pueden eliminar las tablas sobre las que tenemos permiso de borrado.

Normalmente, **el borrado de una tabla es irreversible**, y no hay ninguna petición de confirmación, por lo que conviene ser muy cuidadoso con esta operación.

(7.7) modificar tablas

(7.7.1) cambiar de nombre

La orden **RENAME** permite el cambio de nombre de cualquier objeto. Sintaxis:

```
RENAME nombreViejo TO nombreNuevo
```

(7.7.2) borrar contenido de tablas

La orden **TRUNCATE TABLE** seguida del nombre de una tabla, hace que se elimine el contenido de la tabla, pero no la tabla en sí. Incluso borra del archivo de datos el espacio ocupado por la tabla.

(7.7.3) modificar tablas

La versátil instrucción **ALTER TABLE** permite hacer cambios en la estructura de una tabla.

añadir columnas

```
ALTER TABLE nombreTabla ADD(nombreColumna TipoDatos  
[Propiedades] [,columnaSiguiente tipoDatos  
[propiedades]...)
```

Permite añadir nuevas columnas a la tabla. Se deben indicar su tipo de datos y sus propiedades si es necesario (al estilo de CREATE TABLE).

Las nuevas columnas se añaden al final, no se puede indicar otra posición.
Ejemplo:

```
ALTER TABLE facturas ADD (fecha DATE)
```

borrar columnas

```
ALTER TABLE nombreTabla DROP(columna  
[,columnaSiguiente,...]);
```

Elimina la columna indicada de manera irreversible e incluyendo los datos que contenía. No se puede eliminar la última columna (habrá que usar DROP TABLE).

```
ALTER TABLE facturas DROP (fecha);
```

modificar columna

Permite cambiar el tipo de datos y propiedades de una determinada columna.
Sintaxis:

```
ALTER TABLE nombreTabla MODIFY(columna tipo [propiedades]  
[columnaSiguiente tipo [propiedades] ...])
```

Los cambios que se permiten son (en Oracle):

- ⦿ Incrementar precisión o anchura de los tipos de datos
- ⦿ Sólo se puede reducir la anchura si la anchura máxima de un campo si esa columna posee nulos en todos los registros, o todos los valores son tan pequeños como la nueva anchura o no hay registros
- ⦿ Se puede pasar de **CHAR** a **VARCHAR2** y viceversa (si no se modifica la anchura)
- ⦿ Se puede pasar de **DATE** a **TIMESTAMP** y viceversa

Ejemplo:

```
ALTER TABLE facturas MODIFY (fecha TIMESTAMP);
```

renombrar columna

Esto permite cambiar el nombre de una columna. Sintaxis

```
ALTER TABLE nombreTabla  
RENAME COLUMN nombreAntiguo TO nombreNuevo
```

Ejemplo:

```
ALTER TABLE facturas RENAME fecha TO fechaYhora;
```

(7.8) valor por defecto

A cada columna se le puede asignar un valor por defecto durante su creación mediante la propiedad **DEFAULT**. Se puede poner esta propiedad durante la creación o modificación de la tabla, añadiendo la palabra **DEFAULT** tras el tipo de datos del campo y colocando detrás el valor que se desea por defecto.

Ejemplo:

```
CREATE TABLE articulo (cod NUMBER(7), nombre  
VARCHAR2(25),  
precio NUMBER(11,2) DEFAULT 3.5);
```

La palabra **DEFAULT** se puede añadir durante la creación o la modificación de la tabla (comando **ALTER TABLE**)

(7.9) restricciones

Una restricción es una condición de obligado cumplimiento para una o más columnas de la tabla. A cada restricción se le pone un nombre, en el caso de no poner un nombre (en las que eso sea posible) entonces el propio Oracle le coloca el nombre que es un mnemotécnico con el nombre de tabla, columna y tipo de restricción.

Su sintaxis general es:

```
{CREATE TABLE nombreTabla |  
ALTER TABLE nombreTabla {ADD | MODIFY}}  
(campo tipo [propiedades]  
[[CONSTRAINT nombreRestricción ]] tipoRestricción  
(columnas)  
[,siguienteCampo...]  
[,CONSTRAINT nombrerestricción tipoRestricción (columnas)  
...)
```

Las restricciones tienen un nombre, se puede hacer que sea Oracle el que les ponga nombre, pero entonces será críptico. Por eso es mejor ponerle uno mismo.

Los nombres de restricción no se pueden repetir para el mismo esquema, por lo que es buena idea incluir de algún modo el nombre de la tabla, los campos involucrados y el tipo de restricción en el nombre de la misma. Por ejemplo *pieza_id_pk* podría indicar que el campo *id* de la tabla *pieza* tiene una clave principal (**PRIMARY KEY**).

Oracle aconseja esta regla a la hora de poner nombre a las restricciones:

- ◆ Tres letras para el nombre de la tabla
- ◆ Carácter de subrayado
- ◆ Tres letras con la columna afectada por la restricción
- ◆ Carácter de subrayado
- ◆ Dos letras con la abreviatura del tipo de restricción. La abreviatura puede ser:
 - **NN**. NOT NULL.
 - **PK**. PRIMARY KEY
 - **UK**. UNIQUE
 - **FK**. FOREIGN KEY
 - **CK**. CHECK (validación)

Por ejemplo para hacer que la clave principal de la tabla *Alumnos* sea el *código del alumno*, el nombre de la restricción podría ser:

```
alu_cod_pk
```

(7.9.2) prohibir nulos

La restricción **NOT NULL** permite prohibir los nulos en una determinada tabla. Eso obliga a que la columna tenga que tener obligatoriamente un valor para que sea almacenado el registro.

Se puede colocar durante la creación (o modificación) del campo añadiendo la palabra NOT NULL tras el tipo:

```
CREATE TABLE cliente(dni VARCHAR2(9) NOT NULL);
```

En ese caso el nombre le coloca la propia base de datos (en el caso de Oracle el nombre sería algo como *SY002341* por ejemplo). No es recomendable no poner nombre a las restricciones para controlarlas mejor. Para poner el nombre se usa:

```
CREATE TABLE cliente(dni VARCHAR2(9)  
CONSTRAINT cli_dni_nn NOT NULL);
```

La restricción **NOT NULL** es la única que sólo se puede poner seguida al nombre de la columna a la que se aplica (la razón es que **NOT NULL** sólo se puede aplicar a un campo a la vez)

(7.9.3) valores únicos

Las restricciones de tipo **UNIQUE** obligan a que el contenido de una o más columnas no puedan repetir valores. Nuevamente hay dos formas de colocar esta restricción:

```
CREATE TABLE cliente(dni VARCHAR2(9) UNIQUE);
```

En ese caso el nombre de la restricción la coloca el sistema Oracle. Otra forma es:

```
CREATE TABLE cliente(dni VARCHAR2(9) CONSTRAINT dni_u  
UNIQUE);
```

Esta forma permite poner un nombre a la restricción. Si la repetición de valores se refiere a varios campos, la forma sería:

```
CREATE TABLE alquiler(dni VARCHAR2(9),  
cod_pelicula NUMBER(5),  
CONSTRAINT alquiler_uk UNIQUE(dni,cod_pelicula) ;
```

La coma tras la definición del campo **cod_pelicula** hace que la restricción sea independiente de ese campo. Eso obliga a que, tras **UNIQUE** se indique la lista de campos. Incluso para un solo campo se puede colocar la restricción al final de la lista en lugar de definirlo a continuación del nombre y tipo de la columna.

Las claves candidatas deben llevar restricciones **UNIQUE** y **NOT NULL**

(7.9.4) clave primaria

La clave primaria de una tabla la forman las columnas que indican a cada registro de la misma. La clave primaria hace que los campos que la forman sean **NOT NULL** (sin posibilidad de quedar vacíos) y que los valores de los campos sean de tipo **UNIQUE** (sin posibilidad de repetición).

Si la clave está formada por un solo campo basta con:

```
CREATE TABLE cliente(  
dni VARCHAR2(9) PRIMARY KEY,  
nombre VARCHAR(50)) ;
```

O, poniendo un nombre a la restricción:

```
CREATE TABLE cliente(  
dni VARCHAR2(9) CONSTRAINT cliente_pk PRIMARY KEY,
```



```
nombre VARCHAR(50)) ;
```

Si la clave está formada por más de un campo:

```
CREATE TABLE alquiler(dni VARCHAR2(9),  
    cod_pelicula NUMBER(5),  
    CONSTRAINT alquiler_pk PRIMARY  
KEY(dni,cod_pelicula)) ;
```

(7.9.5) clave secundaria o foránea

Una clave secundaria o foránea, es uno o más campos de una tabla que están relacionados con la clave principal (o incluso con una clave candidata) de otra tabla.

La forma de indicar una clave foránea es:

```
CREATE TABLE alquiler(  
    dni VARCHAR2(9) CONSTRAINT dni_fk REFERENCES  
clientes(dni),  
    cod_pelicula NUMBER(5) CONSTRAINT pelicula_fk  
REFERENCES  
peliculas(cod),  
    CONSTRAINT alquiler_pk PRIMARY KEY(dni,cod_pelicula)  
);
```

Significa esta instrucción (en cuanto a claves foráneas) que el campo **dni** se relaciona con la columna **dni** de la tabla **clientes**.

Si el campo al que se hace referencia es la clave principal, se puede obviar el nombre del campo:

```
CREATE TABLE alquiler(  
    dni VARCHAR2(9) CONSTRAINT dni_fk REFERENCES clientes,  
    cod_pelicula NUMBER(5) CONSTRAINT pelicula_fk  
REFERENCES peliculas,  
    CONSTRAINT alquiler_pk PRIMARY KEY(dni,cod_pelicula)  
);
```

En este caso se entiende que los campos hacen referencia a las claves principales de las tablas referenciadas (si la relación la forma más un campo, el orden de los campos debe de ser el mismo).

Esto forma una relación entre dichas tablas, que además obliga al cumplimiento de la **integridad referencial**. Esta integridad obliga a que cualquier **dni** incluido en la tabla **alquiler** tenga que estar obligatoriamente en la tabla de clientes. De no ser así el registro no será insertado en la tabla (ocurrirá un error).

Otra forma de crear claves foráneas (útil para claves formadas por más de un campo) es:

```
CREATE TABLE existencias(  
    tipo CHAR2(9),  
    modelo NUMBER(3),  
    n_almacen NUMBER(1)  
    cantidad NUMBER(7),  
    CONSTRAINT exi_t_m_fk FOREIGN KEY(tipo,modelo)  
        REFERENCES piezas,  
    CONSTRAINT exi_na_fk FOREIGN KEY(n_almacen)  
        REFERENCES almacenes,  
    CONSTRAINT exi_pk PRIMARY KEY(tipo,modelo,n_almacen)  
);
```

Si la definición de clave secundaria se pone al final hace falta colocar el texto FOREIGN KEY para indicar en qué campos se coloca la restricción de clave foránea. En el ejemplo anterior es absolutamente necesario que la clave principal de la tabla piezas a la que hace referencia la clave la formen las columnas **tipo** y **modelo** y en que estén en ese orden.

La integridad referencial es una herramienta imprescindible de las bases de datos relacionales. Pero provoca varios problemas. Por ejemplo, si borramos un registro en la tabla principal que está relacionado con uno o varios de la secundaria ocurrirá un error, ya que de permitírsenos borrar el registro ocurrirá fallo de integridad (habrá claves secundarios refiriéndose a una clave principal que ya no existe).

Por ello se nos pueden ofrecer soluciones a añadir tras la cláusula REFERENCES. Son:

- ◆ **ON DELETE SET NULL.** Coloca nulos todas las claves secundarias relacionadas con la borrada.
- ◆ **ON DELETE CASCADE.** Borra todos los registros cuya clave secundaria es igual que la clave del registro borrado.
- ◆ **ON DELETE SET DEFAULT.** Coloca en el registro relacionado el valor por defecto en la columna relacionada
- ◆ **ON DELETE NOTHING.** No hace nada.

En el caso explicado se aplicarían las cláusulas cuando se eliminen filas de la clave principal relacionada con la clave secundaria. En esas cuatro cláusulas se podría sustituir la palabra DELETE por la palabra **UPDATE**, haciendo que el funcionamiento se refiera a cuando se modifica un registro de la tabla principal; en muchas bases de datos se admite el uso tanto de ON DELETE como de ON UPDATE.

En la base de datos Oracle sólo se permite utilizar ON DELETE SET NULL u ON DELETE CASCADE. No se admite el uso de ON UPDATE.

La sintaxis completa para añadir claves foráneas es:

```
CREATE TABLE tabla(lista_de_campos
    CONSTRAINT nombreRestriccion FOREIGN KEY
(listaCampos)
    REFERENCES tabla(clavePrincipalRelacionada)
    [ON DELETE {SET NULL | CASCADE}]
);
```

Si es de un solo campo existe esta alternativa:

```
CREATE TABLE tabla(lista_de_campos tipos propiedades,
    nombreCampoClaveSecundaria
    CONSTRAINT nombreRestriccion
    REFERENCES
tabla(clavePrincipalRelacionada)
    [ON DELETE {SET NULL | CASCADE}]
);
```

Ejemplo:

```
CREATE TABLE alquiler(dni VARCHAR2(9),
    cod_pelicula NUMBER(5),
    CONSTRAINT alquiler_pk PRIMARY
KEY(dni,cod_pelicula),
    CONSTRAINT dni_fk FOREIGN KEY (dni)
    REFERENCES clientes(dni)
    ON DELETE SET NULL ON UPDATE CASCADE,
    CONSTRAINT pelicula_fk FOREIGN KEY (cod_pelicula)
    REFERENCES peliculas(cod)
    ON DELETE CASCADE
);
```

(7.9.6) restricciones de validación

Son restricciones que dictan una condición que deben cumplir los contenidos de una columna. Una misma columna puede tener múltiples CHECKS en su definición (se pondrían varios CONSTRAINT seguidos, sin comas).

Ejemplo:

```
CREATE TABLE ingresos(cod NUMBER(5) PRIMARY KEY,  
concepto VARCHAR2(40) NOT NULL,  
importe NUMBER(11,2) CONSTRAINT importe_min  
CHECK (importe>0)  
CONSTRAINT importe_max  
CHECK (importe<8000)  
);
```

En este caso la CHECK prohíbe añadir datos cuyo importe no esté entre 0 y 8000

Para poder hacer referencia a otras columnas hay que construir la restricción de forma independiente a la columna (es decir al final de la tabla):

```
CREATE TABLE ingresos(cod NUMBER(5) PRIMARY KEY,  
concepto VARCHAR2(40) NOT NULL,  
importe_max NUMBER(11,2),  
importe NUMBER(11,2),  
CONSTRAINT importe_maximo  
CHECK (importe<importe_max)  
);
```

(7.9.7) añadir restricciones

Es posible querer añadir restricciones tras haber creado la tabla. En ese caso se utiliza la siguiente sintaxis:

```
ALTER TABLE tabla  
ADD [CONSTRAINT nombre] tipoDeRestricción(columnas);
```

tipoRestricción es el texto **CHECK**, **PRIMARY KEY** o **FOREIGN KEY**. Las restricciones **NOT NULL** deben indicarse mediante **ALTER TABLE .. MODIFY** colocando **NOT NULL** en el campo que se modifica.

(7.9.8) borrar restricciones

Sintaxis:

```
ALTER TABLE tabla  
DROP PRIMARY KEY | UNIQUE(campos) |  
CONSTRAINT nombreRestricción [CASCADE]
```

La opción **PRIMARY KEY** elimina una clave principal (también quitará el índice **UNIQUE** sobre las campos que formaban la clave. **UNIQUE** elimina índices únicos. La opción **CONSTRAINT** elimina la restricción indicada.

La opción **CASCADE** hace que se eliminen en cascada las restricciones de integridad que dependen de la restricción eliminada.

Por ejemplo en:

```
CREATE TABLE curso(  
  cod_curso CHAR(7) PRIMARY KEY,  
  fecha_inicio DATE,  
  fecha_fin DATE,  
  tItulo VARCHAR2(60),  
  cod_siguientecurso CHAR(7),  
  CONSTRAINT fecha_ck CHECK(fecha_fin>fecha_inicio),  
  CONSTRAINT cod_ste_fk FOREIGN KEY(cod_siguientecurso)  
    REFERENCES curso ON DELETE SET NULL);
```

Tras esa definición de tabla, esta instrucción:

```
ALTER TABLE curso DROP PRIMARY KEY;
```

Produce este error (en la base de datos Oracle):

```
ORA-02273: a esta clave única/primaria hacen referencia  
algunas claves ajenas
```

Para ello habría que utilizar esta instrucción:

```
ALTER TABLE curso DROP PRIMARY KEY CASCADE;
```

Esa instrucción elimina la restricción de clave secundaria antes de eliminar la principal.

También produce error esta instrucción:

```
ALTER TABLE curso DROP(fecha_inicio);  
ERROR en línea 1:  
ORA-12991: se hace referencia a la columna en una  
restricción de multicolumna
```

Sistemas Gestores de Bases de Datos

(Unidad 7) DDL

El error se debe a que no es posible borrar una columna que forma parte de la definición de una instrucción. La solución es utilizar CASCADE CONSTRAINT elimina las restricciones en las que la columna a borrar estaba implicada:

```
ALTER TABLE curso DROP(fecha_inicio) CASCADE CONSTRAINTS;
```

Esta instrucción elimina la restricción de tipo CHECK en la que aparecía la *fecha_inicio* y así se puede eliminar la columna.

(7.9.9) desactivar restricciones

A veces conviene temporalmente desactivar una restricción para saltarse las reglas que impone. La sintaxis es:

```
ALTER TABLE tabla DISABLE CONSTRAINT nombre [CASCADE]
```

La opción CASCADE hace que se desactiven también las restricciones dependientes de la que se desactivó.

(7.9.10) activar restricciones

Anula la desactivación. Formato:

```
ALTER TABLE tabla ENABLE CONSTRAINT nombre [CASCADE]
```

Sólo se permite volver a activar si los valores de la tabla cumplen la restricción que se activa. Si hubo desactivado en cascada, habrá que activar cada restricción individualmente.

(7.9.11) cambiar de nombre a las restricciones

Para hacerlo se utiliza este comando:

```
ALTER TABLE table RENAME CONSTRAINT  
nombreViejo TO nombreNuevo;
```

(7.9.12) mostrar restricciones

El trabajo con restricciones ya se ha visto que es complejo. Por eso todas las bases de datos suelen proporcionar una vista (o más) del diccionario de datos que permite consultar las restricciones. En el caso de Oracle, se puede utilizar la vista del diccionario de datos **USER_CONSTRAINTS**.

Esta vista permite identificar las restricciones colocadas por el usuario (**ALL_CONSTRAINTS** permite mostrar las restricciones de todos los usuarios, pero sólo está permitida a los administradores). En esa vista aparece toda la información que el diccionario de datos posee sobre las restricciones.

En ella tenemos las siguientes columnas interesantes:

OWNER	VARCHAR2(20)	Indica el nombre del usuario propietario de la tabla
CONSTRAINT_NAME	VARCHAR2(30)	Nombre de la restricción
CONSTRAINT_TYPE	VARCHAR2(1)	Tipo de restricción: <input type="radio"/> C. De tipo CHECK o NOT NULL <input type="radio"/> P. PRIMARY KEY <input type="radio"/> R. FOREIGN KEY <input type="radio"/> U. UNIQUE
TABLE_NAME	VARCHAR2(30)	Nombre de la tabla en la que se encuentra la restricción

En el diccionario de datos hay otra vista que proporciona información sobre restricciones, se trata de **USER_CONS_COLUMNS**, en dicha tabla se muestra información sobre las columnas que participan en una restricción. Así si hemos definido una clave primaria formada por los campos *uno* y *dos*, en la tabla **USER_CONS_COLUMNS** aparecerán dos entradas, una para el primer campo del índice y otra para el segundo. Se indicará además el orden de aparición en la restricción. Ejemplo (resultado de la instrucción **SELECT * FROM USER_CONS_COLUMNS**):

OWNER	CONSTRAINT_NAME	TABLE_NAME	COLUMN_NAME	POSITION
JORGE	EXIS_PK	EXISTENCIAS	TIPO	1
JORGE	EXIS_PK	EXISTENCIAS	MODELO	2
JORGE	EXIS_PK	EXISTENCIAS	N_ALMACEN	3
JORGE	PIEZA_FK	EXISTENCIAS	TIPO	1
JORGE	PIEZA_FK	EXISTENCIAS	MODELO	2
JORGE	PIEZA_PK	PIEZA	TIPO	1
JORGE	PIEZA_PK	PIEZA	MODELO	2

En esta tabla **USER_CONS_COLUMNS** aparece una restricción de clave primaria sobre la tabla *existencias*, esta clave está formada por las columnas (*tipo*, *modelo* y *n_almacen*) y en ese orden. Una segunda restricción llamada *pieza_fk* está compuesta por *tipo* y *modelo* de la tabla existencias. Finalmente la restricción *pieza_pk* está formada por *tipo* y *modelo*, columnas de la tabla pieza.

Para saber de qué tipo son esas restricciones, habría que acudir a la vista **USER_CONSTRAINTS**.

(7.10) secuencias

Una secuencia sirve para generar automáticamente números distintos. Se utilizan para generar valores para campos que se utilizan como clave forzada (claves cuyo valor no interesa, sólo sirven para identificar los registros de una tabla). Es

decir se utilizan en los identificadores de las tablas (campos que comienzan con la palabra *id*), siempre y cuando no importe qué número se asigna a cada fila.

Es una rutina interna de la base de datos la que realiza la función de generar un número distinto cada vez. Las secuencias se almacenan independientemente de la tabla, por lo que la misma secuencia se puede utilizar para diversas tablas.

(7.10.1) creación de secuencias

Sintaxis:

```
CREATE SEQUENCE secuencia
[INCREMENT BY n]
  [START WITH n]
[ {MAXVALUE n | NOMAXVALUE} ]
[ {MINVALUE n | NOMINVALUE} ]
[ {CYCLE | NOCYCLE} ]
```

Donde:

- ***secuencia***. Es el nombre que se le da al objeto de secuencia
- **INCREMENT BY**. Indica cuánto se incrementa la secuencia cada vez que se usa. Por defecto se incrementa de uno en uno
- **START WITH**. Indica el valor inicial de la secuencia (por defecto 1)
- **MAXVALUE**. Máximo valor que puede tomar la secuencia. Si no se toma **NOMAXVALUE** que permite llegar hasta el 10^{27}
- **MINVALUE**. Mínimo valor que puede tomar la secuencia. Por defecto -10^{26}
- **CYCLE**. Hace que la secuencia vuelva a empezar si se ha llegado al máximo valor.

Ejemplo:

```
CREATE SEQUENCE numeroPlanta
INCREMENT 100
STARTS WITH 100
MAXVALUE 2000
```

(7.10.2) ver lista de secuencias

La vista del diccionario de datos de Oracle **USER_SEQUENCES** muestra la lista de secuencias actuales. La columna **LAST_NUMBER** muestra cual será el siguiente número de secuencia disponible

(7.10.3) uso de la secuencia

Los métodos **NEXTVAL** y **CURRVAL** se utilizan para obtener el siguiente número y el valor actual de la secuencia respectivamente. Ejemplo de uso:

```
SELECT numeroPlanta.NEXTVAL FROM DUAL;
```

Eso muestra en pantalla el siguiente valor de la secuencia. Realmente **NEXTVAL** incrementa la secuencia y devuelve el valor actual. **CURRVAL** devuelve el valor de la secuencia, pero sin incrementar la misma.

Ambas funciones pueden ser utilizadas en:

- ◆ Una consulta **SELECT** que no lleve **DISTINCT**, ni grupos, ni sea parte de una vista, ni sea subconsulta de otro **SELECT**, **UPDATE** o **DELETE**
- ◆ Una subconsulta **SELECT** en una instrucción **INSERT**
- ◆ La cláusula **VALUES** de la instrucción **INSERT**
- ◆ La cláusula **SET** de la instrucción **UPDATE**

No se puede utilizar (y siempre hay tentaciones para ello) como valor para la cláusula **DEFAULT** de un campo de tabla.

Su uso más habitual es como apoyo al comando **INSERT**:

```
INSERT INTO plantas(num, uso)  
VALUES(numeroPlanta.NEXTVAL, 'Suites');
```

(7.10.4) modificar secuencias

Se pueden modificar las secuencias, pero la modificación sólo puede afectar a los futuros valores de la secuencia, no a los ya utilizados. Sintaxis:

```
ALTER SEQUENCE secuencia  
[INCREMENT BY n]  
[START WITH n]  
[ {MAXVALUE n | NOMAXVALUE} ]  
[ {MINVALUE n | NOMINVALUE} ]  
[ {CYCLE | NOCYCLE} ]
```

(7.10.5) borrar secuencias

Lo hace el comando **DROP SEQUENCE** seguido del nombre de la secuencia a borrar.

(7.11) sinónimos

Un sinónimo es un nombre que se asigna a un objeto cualquiera. Normalmente es un nombre menos descriptivo que el original a fin de facilitar la escritura del nombre del objeto en diversas expresiones.

(7.11.1) creación

Sintaxis:

```
CREATE [PUBLIC] SYNONYM nombre FOR objeto;
```

objeto es el objeto al que se referirá el sinónimo. La cláusula **PUBLIC** hace que el sinónimo esté disponible para cualquier usuario (sólo se permite utilizar si disponemos de privilegios administrativos).

(7.11.2) borrado

```
DROP SYNONYM nombre
```

(7.11.3) lista de sinónimos

La vista **USER_SYNONYMS** permite observar la lista de sinónimos del usuario, la vista **ALL_SYNONYMS** permite mostrar la lista completa de sinónimos.

(Unidad 8)

PL/SQL

(8.1) introducción

Casi todos los grandes Sistemas Gestores de Datos incorporan utilidades que permiten ampliar el lenguaje **SQL** para producir pequeñas utilidades que añaden al **SQL** mejoras de la programación estructurada (bucles, condiciones, funciones,...)

Aunque **SQL** es un lenguaje fácil, un usuario normal de la base de datos tendría serias dificultades si tiene que manejar la base de datos con este lenguaje.

PL/SQL es una mejora de **SQL** que es implementada por el precompilador de **Oracle**. Es una extensión procedimental del lenguaje **SQL**; es decir, se trata de un lenguaje creado para dar a **SQL** nuevas posibilidades. Esas posibilidades permiten utilizar condiciones y bucles al estilo de los lenguajes de tercera generación (como **Basic**, **Cobol**, **C++**, **Java**, etc.).

En otros sistemas gestores de bases de datos existen otros lenguajes procedimentales: **SQL Server** utiliza **Transact SQL**, **Informix** usa **Informix 4GL**,...

Lo interesante del lenguaje **PL/SQL** es que integra **SQL** por lo que gran parte de su sintaxis procede de dicho lenguaje.

PL/SQL es un lenguaje pensado para la gestión de datos. La creación de aplicaciones sobre la base de datos se realiza con otras herramientas (**Oracle Developer** o lenguajes externos como **Visual Basic** o **Java**).

El código **PL/SQL** puede almacenarse:

- ◆ En la propia base de datos
- ◆ En archivos externos

(8.1.2) conceptos básicos

programa **PL/SQL**

Conjunto de bloques (contienen órdenes **SQL**) los cuales pueden componerse de más bloques.

procedimiento

Programa PL/SQL almacenado para gestionar datos

función

Programa PL/SQL que a partir de unos datos de entrada obtiene un resultado. Una función puede invocar a otra función (o a otro procedimiento).

trigger (*disparador*)

Programa PL/SQL que se ejecuta cuando ocurre un determinado suceso a un objeto de la base de datos.

paquete

Colección de procedimientos y funciones. Similar a lo que se conoce como librería.

(8.2) variables

(8.2.1) estructura de un bloque PL/SQL

Ya se ha comentado antes que los programas PL/SQL se agrupan en estructuras llamadas **bloques**. Cuando un bloque no tiene nombre, se le llama **bloque anónimo**. Un bloque consta de tres secciones:

- ◆ **Declaraciones.** Define e inicializa las variables, constantes, excepciones de usuario y cursores utilizados en el bloque. Va precedida de la palabra **DECLARE**
- ◆ **Comandos ejecutables.** Sentencias para manipular la base de datos y los datos del programa. Todas estas sentencias van precedidas por la palabra **BEGIN**.
- ◆ **Tratamiento de excepciones.** Para indicar las acciones a realizar en caso de error. Van precedidas por la palabra **EXCEPTION**
- ◆ **Final del bloque.** La palabra **END** da fin al bloque.

La estructura en sí es:

```
DECLARE
    declaraciones (opcional)
BEGIN
    instrucciones ejecutables (obligatorio)
EXCEPTION
    instrucciones de manejo de errores (opcional)
END; (obligatorio)
```

A los bloques se les puede poner nombre usando:

```
PROCEDURE nombre IS  
bloque
```

eso para un procedimiento, para una función:

```
FUNCTION nombre  
RETURN tipoDedatos IS  
bloque
```

si asignar nombre se convierte en un bloque anónimo.

(8.2.2) uso de variables

declarar variables

Las variables se declaran en el apartado **DECLARE** del bloque. PL/SQL no acepta entrada ni salida de datos por sí mismo (hay que combinarle con otras herramientas). La sintaxis de declaración es:

```
DECLARE  
identificador [CONSTANT] tipoDeDatos [:= valorInicial];
```

Ejemplos:

```
DECLARE  
pi CONSTANT NUMBER(9,7) := 3.1415927;  
radio NUMBER(5);  
area NUMBER(14,2) := 23.12;
```

El operador **:=** sirve para asignar valores a una variable. Este operador permite inicializar la variable con un valor determinado. La palabra **CONSTANT** indica que la variable no puede ser modificada (es una constante). Si no se inicia la variable, ésta contendrá el valor NULL.

Los identificadores de Oracle deben de tener 30 caracteres, empezar por letra y continuar con letras, números o guiones bajos (**_**) (también vale el signo de dólar (**\$**) y la almohadilla (**#**)<9. No debería coincidir con nombres de columnas de las tablas ni con palabras reservadas (como SELECT).

En PL/SQL sólo se puede declarar una variable por línea.

tipos de datos para las variables

Las variables PL/SQL pueden pertenecer a uno de los siguientes datos (sólo se listan los tipos básicos, los llamados **escalares**), la mayoría son los mismos del SQL de Oracle.

tipo de datos	descripción
CHAR(n)	Texto de anchura fija
VARCHAR2(n)	Texto de anchura variable
NUMBER[(p,s)]	Número. Opcionalmente puede indicar el tamaño del número (p) y el número de decimales (s)
DATE	Almacena fechas
TIMESTAMP	Almacena fecha y hora
INTERVAL YEAR TO MONTH	Almacena intervalos de años y meses
INTERVAL DAY TO SECOND	Almacena intervalos de días, horas, minutos y segundos
LONG	Para textos de más de 32767 caracteres
LONG RAW	Para datos binarios. PL/SQL no puede mostrar estos datos directamente
INTEGER	Enteros de -32768 a 32767
BINARY_INTEGER	Enteros largos (de -2.147.483.647 a 2.147.483.648)
PLS_INTEGER	Igual que el anterior pero ocupa menos espacio
BOOLEAN	Permite almacenar los valores TRUE (verdadero) y FALSE (falso)
BINARY_DOUBLE	Disponible desde la versión 10g, formato equivalente al double del lenguaje C. Representa números decimales en coma flotante.
BINARY_FLOAT	Otro tipo añadido en la versión 10g, equivalente al float del lenguaje C.

expresión %TYPE

Se utiliza para dar a una variable el mismo tipo de otra variable o el tipo de una columna de una tabla de la base de datos. La sintaxis es:

```
identificador variable | tabla.columna%TYPE;
```

Ejemplo:

```
nom personas.nombre%TYPE;  
precio NUMBER(9,2);  
precio_iva precio%TYPE;
```

La variable **precio_iva** tomará el tipo de la variable precio (es decir **NUMBER(9,2)**) la variable **nom** tomará el tipo de datos asignado a la columna **nombre** de la tabla **personas**.

(8.2.3) DBMS_OUTPUT.PUT_LINE

Para poder mostrar datos (fechas, textos y números), Oracle proporciona una función llamada **put_line** en el paquete **dbms_output**. Ejemplo:

```
DECLARE
  a NUMBER := 17;
BEGIN
  DBMS_OUTPUT.PUT_LINE(a);
END;
```

Eso escribiría el número 17 en la pantalla. Pero para ello se debe habilitar primero el paquete en el entorno de trabajo que utilizemos. En el caso de iSQL*Plus hay que colocar la orden interna (no lleva punto y coma):

```
SET SERVEROUTPUT ON
```

hay que escribirla antes de empezar a utilizar la función.

(8.2.4) alcance de las variables

Ya se ha comentado que en PL/SQL puede haber un bloque dentro de otro bloque. Un bloque puede anidarse dentro de:

- ◆ Un apartado **BEGIN**
- ◆ Un apartado **EXCEPTION**

Hay que tener en cuenta que las variables declaradas en un bloque concreto, son eliminadas cuando éste acaba (con su END correspondiente). Ejemplo:

```
DECLARE
  v NUMBER := 2;
BEGIN
  v:=v*2;
  DECLARE
    z NUMBER := 3;
  BEGIN
    z:=v*3;
    DBMS_OUTPUT.PUT_LINE(z); --escribe 12
    DBMS_OUTPUT.PUT_LINE(v); --escribe 4
  END;
  DBMS_OUTPUT.PUT_LINE(v*2); --escribe 8
  DBMS_OUTPUT.PUT_LINE(z); --error
END;
```

En el ejemplo anterior, se produce un error porque **z** no es accesible desde ese punto, el bloque interior ya ha finalizado. Sin embargo desde el bloque interior sí se puede acceder a **v**

(8.3) escritura de instrucciones PL/SQL

(8.3.1) comentarios

Pueden ser de dos tipos:

- ◆ **Comentarios de varias líneas.** Comienzan con **/*** y terminan con ***/**
- ◆ **Comentarios de línea simple.** Son los que utilizan los signos **--** (doble guión). El texto a la derecha de los guiones se considera comentario (el de la izquierda no)

Ejemplo:

```
DECLARE
  v NUMBER := 17;
BEGIN
  /* Este es un comentario que
  ocupa varias líneas */
  v:=v*2; -- este sólo ocupa esta línea
  DBMS_OUTPUT.PUT_LINE(v) -- escribe 34
END;
```

(8.3.2) operadores y funciones

operadores

En PL/SQL se permiten utilizar todos los operadores de SQL: los operadores aritméticos (**+*/**), condicionales (**> < != <> >= <= OR AND NOT**) y de cadena (**||**).

A estos operadores, PL/SQL añade el operador de potencia ******. Por ejemplo **4**3** es **4³**.

funciones

Se pueden utilizar las funciones de Oracle procedentes de SQL (**TO_CHAR, SYSDATE, NVL, SUBSTR, SIN**, etc., etc.) excepto la función **DECODE** y las funciones de grupo (**SUM, MAX, MIN, COUNT**,...)

A estas funciones se añaden diversas procedentes de paquetes de Oracle o creados por los programadores y las funciones **GREATEST** y **LEAST**

(8.3.3) instrucciones SQL permitidas

instrucciones SELECT en PL/SQL

PL/SQL admite el uso de un SELECT que permite almacenar valores en variables. Es el llamado **SELECT INTO**.

Su sintaxis es:

```
SELECT listaDeCampos
INTO listaDeVariables
FROM tabla
[JOIN ...]
WHERE condición]
```

La cláusula INTO es obligatoria en PL/SQL y además la expresión SELECT **sólo puede devolver una única fila**; de otro modo, ocurre un error.

Ejemplo:

```
DECLARE
    v_salario NUMBER(9,2);
    v_nombre VARCHAR2(50);
BEGIN
    SELECT salario,nombre INTO v_salario, v_nombre
    FROM empleados WHERE id_empleado=12344;
    SYSTEM_OUTPUT.PUT_LINE('El nuevo salario será de ' ||
        salario*1.2 || 'euros');
END;
```

instrucciones DML y de transacción

Se pueden utilizar instrucciones DML dentro del código ejecutable. Se permiten las instrucciones **INSERT**, **UPDATE**, **DELETE** y **MERGE**; con la ventaja de que en PL/SQL pueden utilizar variables.

Las instrucciones de transacción **ROLLBACK** y **COMMIT** también están permitidas para anular o confirmar instrucciones.

(8.3.4) instrucciones de control de flujo

instrucción IF

Se trata de una sentencia tomada de los lenguajes estructurados. Desde esta sentencia se consigue que ciertas instrucciones se ejecuten o no dependiendo de una condición

Sistemas Gestores de Bases de Datos

(Unidad 8) PL/SQL

sentencia IF simple

Sintaxis:

```
IF condicion THEN
    instrucciones
END IF;
```

Las instrucciones se ejecutan en el caso de que la condición sea verdadera. La condición es cualquier expresión que devuelva verdadero o falso. Ejemplo:

```
IF departamento=134 THEN
    salario := salario * 13;
    departamento := 123;
END IF;
```

sentencia IF-THEN-ELSE

Sintaxis:

```
IF condición THEN
    instrucciones
ELSE
    instrucciones
END IF;
```

En este caso las instrucciones bajo el ELSE se ejecutan si la condición es falsa.

sentencia IF-THEN-ELSIF

Cuando se utilizan sentencias de control es común desear anidar un IF dentro de otro IF.

Ejemplo:

```
IF saldo>90 THEN
    DBMS_OUTPUT.PUT_LINE('Saldo mayor que el esperado');
ELSE
    IF saldo>0 THEN
        DBMS_OUTPUT.PUT_LINE('Saldo menor que el
esperado');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Saldo NEGATIVO');
    END IF;
END IF;
```

Otra solución es utilizar esta estructura:

```
IF condición1 THEN
    instrucciones1
ELSIF condición2 THEN
    instrucciones3
[ELSIF.... ]
ELSE
    instruccionesElse
END IF;
```

En este IF (que es el más completo) se evalúa la primera condición; si es verdadera se ejecutan las primeras instrucciones y se abandona el IF; si no es así se mira la siguiente condición y si es verdadera se ejecutan las siguientes instrucciones, si es falsa se va al siguiente ELSIF a evaluar la siguiente condición, y así sucesivamente. La cláusula ELSE se ejecuta sólo si no se cumple ninguna de las anteriores condiciones. Ejemplo (equivalente al anterior):

```
IF saldo>90 THEN
    DBMS_OUTPUT.PUT_LINE('Saldo mayor que el esperado');
ELSIF saldo>0 THEN
    DBMS_OUTPUT.PUT_LINE('Saldo menor que el esperado');
ELSE
    DBMS_OUTPUT.PUT_LINE('Saldo NEGATIVO');
END IF;
```

sentencia CASE

La sentencia CASE devuelve un resultado tras evaluar una expresión. Sintaxis:

```
CASE selector
    WHEN expresion1 THEN resultado1
    WHEN expresion2 THEN resultado2
    ...
    [ELSE resultadoElse]
END;
```

Sistemas Gestores de Bases de Datos

(Unidad 8) PL/SQL

Ejemplo:

```
texto:= CASE actitud
        WHEN 'A' THEN 'Muy buena '
        WHEN 'B' THEN 'Buena '
        WHEN 'C' THEN 'Normal '
        WHEN 'D' THEN 'Mala '
        ELSE 'Desconocida '
END;
```

Hay que tener en cuenta que la sentencia CASE sirve para devolver un valor y no para ejecutar una instrucción.

Otra posibilidad es utilizar este otro formato de CASE:

```
CASE
  WHEN condición1 THEN resultado1
  WHEN condición2 THEN resultado2
  ...
  [ELSE resultadoElse]
END;
```

Este segundo formato facilita la escritura de sentencias CASE más complicadas. Por ejemplo:

```
aprobado:= CASE
  WHEN actitud='A' AND nota>=4 THEN TRUE
  WHEN nota>=5 AND (actitud='B' OR actitud='C') THEN TRUE
  WHEN nota>=7 THEN TRUE
  ELSE FALSE
END;
```

Bucles

bucle LOOP

Se trata de una instrucción que contiene instrucción que se repiten indefinidamente (bucle infinito). Se inicia con la palabra **LOOP** y finaliza con la palabra **END LOOP** y dentro de esas palabras se colocan las instrucciones que se repetirán.

Lógicamente no tiene sentido utilizar un bucle infinito, por eso existe una instrucción llamada **EXIT** que permite abandonar el bucle. Cuando Oracle encuentra esa instrucción, el programa continua desde la siguiente instrucción al **END LOOP**.

Lo normal es colocar **EXIT** dentro de una sentencia **IF** a fin de establecer una condición de salida del bucle. También se puede acompañar a la palabra **EXIT** de

la palabra **WHEN** seguida de una condición. Si se condición es cierta, se abandona el bucle, sino continuamos dentro.

Sintaxis

```
LOOP
  instrucciones
  ...
  EXIT [WHEN condición]
END LOOP;
```

Ejemplo (bucle que escribe los números del 1 al 10):

```
DECLARE
  cont NUMBER :=1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE(cont);
    EXIT WHEN cont=10;
    cont:=cont+1;
  END LOOP;
END;
```

bucle WHILE

Genera un bucle cuyas instrucciones se repiten mientras la condición que sigue a la palabra WHILE sea verdadera. Sintaxis:

```
WHILE condición LOOP
  instrucciones
END LOOP;
```

En este bucle es posible utilizar (aunque no es muy habitual en este tipo de bucle) la instrucción **EXIT** o **EXIT WHEN**. La diferencia con el anterior es que este es más estructurado (más familiar para los programadores de lenguajes como Basic, Pascal, C, Java,...)

Sistemas Gestores de Bases de Datos

(Unidad 8) PL/SQL

Ejemplo (escribir números del 1 al 10):

```
DECLARE
    cont NUMBER :=1;
BEGIN
    WHILE cont<=10 LOOP
        DBMS_OUTPUT.PUT_LINE(cont);
        cont:=cont+1;
    END LOOP;
END;
```

bucle FOR

Se utilizar para bucles con contador, bucles que se recorren un número concreto de veces. Para ello se utiliza una variable (contador) que no tiene que estar declarada en el **DECLARE**, esta variable es declarada automáticamente en el propio **FOR** y se elimina cuando éste finaliza.

Se indica el valor inicial de la variable y el valor final (el incremento irá de uno en uno). Si se utiliza la cláusula **REVERSE**, entonces el contador cuenta desde el valor alto al bajo restando 1. Sintaxis:

```
FOR contador IN [REVERSE] valorBajo..valorAlto
    instrucciones
END LOOP;
```

bucles anidados

Se puede colocar un bucle dentro de otro sin ningún problema, puede haber un **WHILE** dentro de un **FOR**, un **LOOP** dentro de otro **LOOP**, etc.

Hay que tener en cuenta que en ese caso, la sentencia **EXIT** abandonaría el bucle en el que estamos:

```
FOR i IN 1..10 LOOP
    FOR j IN 1..30 LOOP
        EXIT WHEN j=5;
        ...
    END LOOP;
    ...
END LOOP;
```

El bucle más interior sólo cuenta hasta que *j* vale 5 ya que la instrucción **EXIT** abandona el bucle más interior cuando *j* llega a ese valor.

No obstante hay una variante de la instrucción **EXIT** que permite salir incluso del bucle más exterior. Eso se consigue poniendo una etiqueta a los bucles que se deseen. Una etiqueta es un identificador que se coloca dentro de los signos **<<** y **>>** delante del bucle. Eso permite poner nombre al bucle.

Por ejemplo:

```
<<buclei>>
FOR i IN 1..10 LOOP
  FOR j IN 1..30 LOOP
    EXIT buclei WHEN j=5;
    ...
  END LOOP;
  ...
END LOOP buclei;
```

En este caso cuando *j* vale 5 se abandonan ambos bucles. No es obligatorio poner la etiqueta en la instrucción END LOOP (en el ejemplo en la instrucción **END LOOP buclei**), pero se suele hacer por dar mayor claridad al código.

(8.4) cursores

(8.4.1) introducción

Los cursores representan consultas **SELECT** de **SQL** que devuelven más de un resultado y que permiten el acceso a cada fila de dicha consulta. Lo cual significa que el cursor siempre tiene un puntero señalando a una de las filas del **SELECT** que representa el cursor.

Se puede recorrer el cursor haciendo que el puntero se mueva por las filas. Los cursores son las herramientas fundamentales de PL/SQL

(8.4.2) procesamiento de cursores

Los cursores se procesan en tres pasos:

- (1) **Declarar el cursor**
- (2) **Abrir el cursor.** Tras abrir el cursor, el puntero del cursor señalará a la primera fila (si la hay)
- (3) **Procesar el cursor.** La instrucción **FETCH** permite recorrer el cursor registro a registro hasta que el puntero llegue al final (se dice que hasta que el cursor esté vacío)
- (4) **Cerrar el cursor**

(8.4.3) declaración de cursores

Sintaxis:

```
CURSOR nombre IS sentenciaSELECT;
```

La sentencia **SELECT** indicada no puede tener apartado **INTO**. Lógicamente esta sentencia sólo puede ser utilizada en el apartado **DECLARE**.

Ejemplo:

```
CURSOR cursorProvincias IS
  SELECT p.nombre, SUM(poblacion) AS poblacion
  FROM localidades l
  JOIN provincias p USING (n_provincia)
  GROUP BY p.nombre;
```

(8.4.4) apertura de cursores

```
OPEN cursor;
```

Esta sentencia abre el cursor, lo que significa:

- (1) Reservar memoria suficiente para el cursor
- (2) Ejecutar la sentencia **SELECT** a la que se refiere el cursor
- (3) Colocar el puntero en la primera fila

Si la sentencia **SELECT** del cursor no devuelve registros, Oracle no devolverá una excepción.

(8.4.5) instrucción **FETCH**

La sentencia **FETCH** es la encargada de recorrer el cursor e ir procesando los valores del mismo:

```
FETCH cursor INTO listaDeVariables
```

Esta instrucción almacena el contenido de la fila a la que apunta actualmente el puntero en la lista de variables indicada, que tiene tener el mismo tipo y número que las columnas representadas en el cursor. Tras esta instrucción el puntero de registros avanza a la siguiente posición.

Esta instrucción se coloca dentro de un bucle a fin de ir procesando cada fila del cursor. Ejemplo:

```
FETCH cursorProvincias INTO v_nombre, v_poblacion;
```

Una instrucción **FETCH** lee una sola fila y su contenido lo almacena en variables.

Por ello se usa siempre dentro de bucles:

```
LOOP
  FETCH cursorProvincias INTO (v_nombre, v_poblacion);
  EXIT WHEN... --aquí se pondría la condición de salida
  ... --instrucciones de proceso de los datos del cursor
END LOOP;
```

(8.4.6) cerrar el cursor

```
CLOSE cursor;
```

Al cerrar el cursor se libera la memoria que ocupa y se impide su procesamiento. Tras cerrar el cursor se podría abrir de nuevo.

(8.4.7) atributos de los cursores

Para poder procesar adecuadamente los cursores se pueden utilizar una serie de atributos que devuelven verdadero o falso según la situación actual del cursor. Se utilizan indicando el nombre del cursor e inmediatamente el nombre del atributo (por ejemplo *cursorProvincias%ISOPEN*)

%ISOPEN

Devuelve verdadero si el cursor ya está abierto.

%NOTFOUND

Devuelve verdadero si la última instrucción FETCH no devolvió ningún valor. Ejemplo:

```
DECLARE
  CURSOR cursorProvincias IS
    SELECT p.nombre, SUM(poblacion) AS poblacion
    FROM LOCALIDADES l
    JOIN PROVINCIAS p USING (n_provincia)
    GROUP BY p.nombre;

  v_nombre PROVINCIAS.nombre%TYPE;
  v_poblacion LOCALIDADES.poblacion%TYPE;
```

```
BEGIN
  OPEN cursorProvincias;
  LOOP
    FETCH cursorProvincias INTO v_nombre,
      v_poblacion;
    EXIT WHEN cursorProvincias%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_nombre || ', ' ||
      v_poblacion);
  END LOOP;
  CLOSE cursorProvincias;
END;
```

En el ejemplo anterior se recorre el cursor hasta que el FETCH no devuelve ninguna fila. Lo que significa que el programa anterior muestra el nombre de cada provincia seguida de una coma y de la población de la misma.

%FOUND

Instrucción contraria a la anterior, devuelve verdadero si el último FETCH devolvió una fila.

%ROWCOUNT

Indica el número de filas que se han recorrido en el cursor (inicialmente vale cero). Es decir, indica cuántos FETCH se han aplicado sobre el cursor.

(8.4.8) variables de registro

intro

Los registros son una estructura estática de datos presente en casi todos los lenguajes clásicos. Se trata de un tipo de datos que se compone de datos más simple. Por ejemplo el registro *persona* se compondría de los datos simples *nombre, apellidos, dirección, fecha de nacimiento*, etc.

En PL/SQL su interés radica en que cada fila de una tabla o vista se puede interpretar como un registro, ya que cada fila se compone de datos simples. Por ello los registros facilitan la manipulación de los cursores.

declaración

Para utilizar registros, primero hay que definir los datos que componen al registro. Así se define el tipo de registro (por eso se utiliza la palabra TYPE). Después se declarará una variable de registro que sea del tipo declarado (es decir, puede haber varias variables del mismo tipo de registro).

Sintaxis:

```
TYPE nombreTipoRegistro IS RECORD(  
    campo1 tipoCampo1 [:= valorInicial],  
    campo2 tipoCampo2 [:= valorInicial],  
    ...  
    campoN tipoCampoN [:= valorInicial]  
)  
  
nombreVariableDeRegistro nombreTipoRegistro;
```

Ejemplo:

```
TYPE regPersona IS RECORD(  
    nombre VARCHAR2(25),  
    apellido1 VARCHAR2(25),  
    apellido2 VARCHAR2(25),  
    fecha_nac DATE  
);  
alvaro regPersona;  
laura regPersona;
```

uso de registros

Para rellenar los valores de los registros se indica el nombre de la variable de registro seguida de un punto y el nombre del campo a rellenar:

```
alvaro.nombre := 'Alvaro';  
alvaro.fecha_nac := TO_DATE('2/3/2004');
```

%ROWTYPE

Al declarar registros, se puede utilizar el modificador %ROWTYPE que sirve para asignar a un registro la estructura de una tabla. Por ejemplo:

```
DECLARE  
    regPersona personas%ROWTYPE;
```

personas debe ser una tabla. **regPersona** tendrá los mismos campos y tipos que las columnas de la tabla **personas**.

(8.4.9) cursores y registros

uso de FETCH con registros

Una de las desventajas, con lo visto hasta ahora, de utilizar **FETCH** reside en que necesitamos asignar todos los valores de cada fila del cursor a una variable. Por lo que si una fila tiene 10 columnas, habrá que declarar 10 variables.

En lugar de ello se puede utilizar una variable de registro y asignar el resultado de **FETCH** a esa variable. Ejemplo (equivalente al de la página 144):

```
DECLARE
    CURSOR cursorProvincias IS
        SELECT p.nombre, SUM(poblacion) AS poblacion
        FROM LOCALIDADES l
        JOIN PROVINCIAS p USING (n_provincia)
        GROUP BY p.nombre;

    rProvincias cursorProvincias%ROWTYPE;
BEGIN
    OPEN cursorProvincias;
    LOOP
        FETCH cursorProvincias INTO rProvincias;
        EXIT WHEN cursorProvincias%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(rProvincias.nombre || ', ' ||
                               rProvincias.poblacion);
    END LOOP;
    CLOSE cursorProvincias;
END;
```

bucle FOR de recorrido de cursores

Es la forma más habitual de recorrer todas las filas de un cursor. Es un bucle **FOR** que se encarga de realizar tres tareas:

- (1) Abre un cursor (realiza un **OPEN** sobre el cursor antes de empezar el bucle)
- (2) Recorre todas las filas de un cursor (cada vez que se entra en el interior del **FOR** se genera un **FETCH** implícito) y almacena el contenido de cada fila en una variable de registro (que no es necesario declarar en la zona **DECLARE**)
- (3) Cierra el cursor (cuando finaliza el **FOR**)

Sintaxis:

```
FOR variableRegistro IN cursor LOOP  
  ..instrucciones  
END LOOP;
```

Esa sintaxis es equivalente a:

```
OPEN cursor;  
LOOP  
  FETCH cursor INTO variableRegistro;  
  EXIT WHEN cursor%NOTFOUND;  
  ...instrucciones  
END LOOP;
```

Ejemplo (equivalente al ejemplo anterior):

```
DECLARE  
  CURSOR cursorProvincias IS  
    SELECT p.nombre, SUM(poblacion) AS poblacion  
    FROM LOCALIDADES l  
    JOIN PROVINCIAS p USING (n_provincia)  
    GROUP BY p.nombre;  
BEGIN  
  FOR rProvincias IN cursorProvincias LOOP  
    DBMS_OUTPUT.PUT_LINE(rProvincias.nombre || ', ' ||  
                          rProvincias.poblacion);  
  END LOOP;  
END;
```

Naturalmente este código es más sencillo de utilizar y más corto que los anteriores.

(8.4.10) uso avanzado de cursores

cursores con parámetros

En muchas ocasiones se podría desear que el resultado de un cursor dependa de una variable. Por ejemplo al presentar una lista de personal, hacer que aparezca el cursor de un determinado departamento y puesto de trabajo.

Para hacer que el cursor varíe según esos parámetros, se han de indicar los mismos en la declaración del cursor. Para ello se pone entre paréntesis su nombre y tipo tras el nombre del cursor en la declaración.

Ejemplo:

```
DECLARE
    CURSOR cur_personas(dep NUMBER, pue VARCHAR2(20)) IS
        SELECT nombre, apellidos
        FROM empleados
        WHERE departamento=dep AND puesto=pue;
BEGIN
    OPEN cur_personas(12, 'administrativo');
    .....
    CLOSE cur_personas;
END
```

Es al abrir el cursor cuando se indica el valor de los parámetros, lo que significa que se puede abrir varias veces el cursor y que éste obtenga distintos resultados dependiendo del valor del parámetro.

Se pueden indicar los parámetros también en el bucle FOR:

```
DECLARE
    CURSOR cur_personas(dep NUMBER, pue VARCHAR2(20)) IS
        SELECT nombre, apellidos
        FROM empleados
        WHERE departamento=dep AND puesto=pue;
BEGIN
    FOR r IN cur_personas(12, 'administrativo') LOOP
        .....
    END LOOP;
END
```

actualizaciones al recorrer registros

En muchas ocasiones se realizan operaciones de actualización de registros sobre el cursor que se está recorriendo. Para evitar problemas se deben bloquear los registros del cursor a fin de detener otros procesos que también desearan modificar los datos.

Esta cláusula se coloca al final de la sentencia **SELECT** del cursor (iría detrás del **ORDER BY**). Opcionalmente se puede colocar el texto **NOWAIT** para que el programa no se quede esperando en caso de que la tabla esté bloqueada por otro usuario. Se usa el texto **OF** seguido del nombre del campo que se modificará (no es necesaria esa cláusula, pero se mantiene para clarificar el código).

Sintaxis:

```
CURSOR ...  
  SELECT ...  
  FOR UPDATE [OF campo] [NOWAIT]
```

Ejemplo:

```
DECLARE  
  CURSOR c_emp IS  
    SELECT id_emp, nombre, n_departamento, salario  
    FROM empleados, departamentos  
    WHERE empleados.id_dep=departamentos.id_dep  
          AND empleados.id_dep=80  
    FOR UPDATE OF salario NOWAIT;
```

A continuación en la instrucción UPDATE que modifica los registros se puede utilizar una nueva cláusula llamada WHERE CURRENT OF seguida del nombre de un cursor, que hace que se modifique sólo el registro actual del cursor. Ejemplo:

```
FOR r_emp IN c_emp LOOP  
  IF r_emp.salario<1500 THEN  
    UPDATE empleados SET salario = salario *1.30  
    WHERE CURRENT OF c_emp;
```

(8.5) excepciones

(8.5.1) introducción

Se llama excepción a todo hecho que le sucede a un programa que causa que la ejecución del mismo finalice. Lógicamente eso causa que el programa termine de forma anormal.

Las excepciones se debe a:

- ◆ Que ocurra un error detectado por Oracle (por ejemplo si un **SELECT** no devuelve datos ocurre el error **ORA-01403** llamado **NO_DATA_FOUND**).
- ◆ Que el propio programador las lance (comando **RAISE**).

Las excepciones se pueden capturar a fin de que el programa controle mejor la existencia de las mismas.

(8.5.2) captura de excepciones

La captura se realiza utilizando el bloque EXCEPTION que es el bloque que está justo antes del END del bloque. Cuando una excepción ocurre, se comprueba el bloque EXCEPTION para ver si ha sido capturada, si no se captura, el error se propaga a Oracle que se encargará de indicar el error existente.

Las excepciones pueden ser de estos tipos:

- ◆ **Excepciones predefinidas de Oracle.** Que tienen ya asignado un nombre de excepción.
- ◆ **Excepciones de Oracle sin definir.** No tienen nombre asignado pero se les puede asignar.
- ◆ **Definidas por el usuario.** Las lanza el programador.

La captura de excepciones se realiza con esta sintaxis:

```
DECLARE
    sección de declaraciones
BEGIN
    instrucciones
EXCEPTION
    WHEN excepción1 [OR excepción2 ...] THEN
        instrucciones que se ejecutan si suceden esas excepciones
    [WHEN excepción3 [OR...]] THEN
        instrucciones que se ejecutan si suceden esas excepciones
    [WHEN OTHERS THEN]
        instrucciones que se ejecutan si suceden otras excepciones
END;
```

Cuando ocurre una determinada excepción, se comprueba el primer WHEN para comprobar si el nombre de la excepción ocurrida coincide con el que dicho WHEN captura; si es así se ejecutan las instrucciones, si no es así se comprueba el siguiente **WHEN** y así sucesivamente.

Si existen cláusula **WHEN OTHERS**, entonces las excepciones que no estaban reflejadas en los demás apartados WHEN ejecutan las instrucciones del WHEN OTHERS. Ésta cláusula debe ser la última.

(8.5.3) excepciones predefinidas

Oracle tiene las siguientes excepciones predefinidas. Son errores a los que Oracle asigna un nombre de excepción. Están presentes los más comunes:

Nombre de excepción	Número de error	Ocurre cuando..
ACCESS_INTO_NULL	ORA-06530	Se intentan asignar valores a un objeto que no se había inicializado
CASE_NOT_FOUND	ORA-06592	Ninguna opción WHEN dentro de la instrucción CASE captura el valor, y no hay instrucción ELSE
COLLECTION_IS_NULL	ORA-06531	Se intenta utilizar un varray o una tabla anidada que no estaba inicializada
CURSOR_ALREADY_OPEN	ORA-06511	Se intenta abrir un cursor que ya se había abierto
DUP_VAL_ON_INDEX	ORA-00001	Se intentó añadir una fila que provoca que un índice único repita valores
INVALID_CURSOR	ORA-01001	Se realizó una operación ilegal sobre un cursor
INVALID_NUMBER	ORA-01722	Falla la conversión de carácter a número
LOGIN_DENIED	ORA-01017	Se intenta conectar con Oracle usando un nombre de usuario y contraseña inválidos
NO_DATA_FOUND	ORA-01403	El SELECT de fila única no devolvió valores
PROGRAM_ERROR	ORA-06501	Error interno de Oracle
ROWTYPE_MISMATCH	ORA-06504	Hay incompatibilidad de tipos entre el cursor y las variables a las que se intentan asignar sus valores
STORAGE_ERROR	ORA-06500	No hay memoria suficiente
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Se hace referencia a un elemento de un varray o una tabla anidada usando un índice mayor que los elementos que poseen
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Se hace referencia a un elemento de un varray o una tabla anidada usando un índice cuyo valor está fuera del rango legal
SYS_INVALID_ROWID	ORA-01410	Se convierte un texto en un número de identificación de fila (ROWID) y el texto no es válido
TIMEOUT_ON_RESOURCE	ORA-00051	Se consumió el máximo tiempo en el que Oracle permite esperar al recurso
TOO_MANY_ROWS	ORA-01422	El SELECT de fila única devuelve más de una fila
VALUE_ERROR	ORA-06502	Hay un error aritmético, de conversión, de redondeo o de tamaño en una operación
ZERO_DIVIDE	ORA-01476	Se intenta dividir entre el número cero.

Ejemplo:

```
DECLARE
  x NUMBER :=0;
  y NUMBER := 3;
  res NUMBER;
BEGIN
  res:=y/x;
  DBMS_OUTPUT.PUT_LINE(res);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('No se puede dividir por cero') ;
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error inesperado') ;
END;
```

(8.5.4) excepciones sin definir

Pueden ocurrir otras muchas excepciones que no están en la lista anterior. En ese caso aunque no tienen un nombre asignado, sí tienen un número asignado. Ese número es el que aparece cuando Oracle muestra el mensaje de error tras la palabra ORA.

Por ejemplo en un error por restricción de integridad Oracle lanza un mensaje encabezado por el texto: **ORA-02292** Por lo tanto el error de integridad referencia es el —02292.

Si deseamos capturar excepciones sin definir hay que:

- (1) Declarar un nombre para la excepción que capturaremos. Eso se hace en el apartado DECLARE con esta sintaxis:

```
nombreDeExcepción EXCEPTION;
```

- (2) Asociar ese nombre al número de error correspondiente mediante esta sintaxis en el apartado DECLARE (tras la instrucción del paso 1):

```
PRAGMA EXCEPTION_INIT(nombreDeExcepción, nºDeExcepción);
```

- (3) En el apartado EXCEPTION capturar el nombre de la excepción como si fuera una excepción normal.

Ejemplo:

```
DECLARE
    e_integridad EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_integridad, -2292);
BEGIN
    DELETE FROM piezas WHERE tipo='TU' AND modelo=6;
EXCEPTION
    WHEN e_integridad THEN
        DBMS_OUTPUT.PUT_LINE('No se puede borrar esa pieza' ||
            ' porque tiene existencias relacionadas');
END;
```

(8.5.5) funciones de uso con excepciones

Se suelen usar dos funciones cuando se trabaja con excepciones:

- ◆ **SQLCODE**. Retorna el código de error del error ocurrido
- ◆ **SQLERRM**. Devuelve el mensaje de error de Oracle asociado a ese número de error.

Ejemplo:

```
EXCEPTION
    ...
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Ocurrió el error ' ||
            SQLCODE || 'mensaje: ' || SQLERRM);
END;
```

(8.5.6) excepciones de usuario

El programador puede lanzar sus propias excepciones simulando errores del programa. Para ello hay que:

- (1) Declarar un nombre para la excepción en el apartado DECLARE, al igual que para las excepciones sin definir:

```
miExcepcion EXCEPTION;
```

- (2) En la sección ejecutable (BEGIN) utilizar la instrucción **RAISE** para lanzar la excepción:

```
RAISE miExcepcion;
```

Sistemas Gestores de Bases de Datos

(Unidad 8) PL/SQL

- (3) En el apartado de excepciones capturar el nombre de excepción declarado:

```
EXCEPTION
...
WHEN miExcepcion THEN
....
```

Otra forma es utilizar la función **RAISE_APPLICATION_ERROR** que simplifica los tres pasos anteriores. Sintaxis:

```
RAISE_APPLICATION_ERROR(nºDeError, mensaje
[, {TRUE|FALSE}]);
```

Esta instrucción se coloca en la sección ejecutable o en la de excepciones y sustituye a los tres pasos anteriores. Lo que hace es lanzar un error cuyo número debe de estar entre el -20000 y el -20999 y hace que Oracle muestre el mensaje indicado. El tercer parámetro opciones puede ser **TRUE** o **FALSE** (por defecto TRUE) e indica si el error se añade a la pila de errores existentes.

Ejemplo:

```
DECLARE

BEGIN
    DELETE FROM piezas WHERE tipo='ZU' AND modelo=26;
    IF SQL%NOTFOUND THEN
        RAISE_APPLICATION_ERROR(-20001,'No existe esa
pieza');
    END IF;
END;
```

En el ejemplo, si la pieza no existe, entonces **SQL%NOTFOUND** devuelve verdadero ya que el DELETE no elimina ninguna pieza. Se lanza la excepción de usuario -20001 haciendo que Oracle utilice el mensaje indicado. Oracle lanzará el mensaje:

```
ORA-20001: No existe esa pieza
```

(8.6) procedimientos

(8.6.1) introducción

Un procedimiento es un bloque PL/SQL al que se le asigna un nombre. Un procedimiento se crea para que realice una determinada tarea de gestión.

Los procedimientos son compilados y almacenados en la base de datos. Gracias a ellos se consigue una reutilización eficiente del código, ya que se puede invocar al procedimiento las veces que haga falta desde otro código o desde una herramienta de desarrollo como Oracle Developer. Una vez almacenados pueden ser modificados de nuevo.

(8.6.2) estructura de un procedimiento

La sintaxis es:

```
CREATE [OR REPLACE] PROCEDURE nombreProcedimiento
[(parámetro1 [modelo] tipoDatos
  [,parámetro1 [modelo] tipoDatos [,...]])]
IS | AS
  secciónDeDeclaraciones
BEGIN
  instrucciones
[EXCEPTION
  controlDeExcepciones]
END;
```

La opción **REPLACE** hace que si ya existe un procedimiento con ese nombre, se reemplaza el anterior con el que se crea ahora. Los parámetros son la lista de variables que necesita el procedimiento para realizar su tarea; se indica el tipo de los mismos, pero no su tamaño (VARCHAR2 y no VARCHAR2(50)). En los parámetros, el modo pueden ser las palabras IN, OUT o IN OUT (se explican más adelante).

La palabra **DECLARE** no se utiliza, la sección de declaraciones figura tras las palabras **IS** o **AS**.

(8.6.3) desarrollo de procedimientos

Los pasos para desarrollar procedimientos son:

- (1) Escribir el código en un archivo **.sql** desde cualquier editor.
- (2) Compilar el código desde un editor como **iSQL*Plus** o cualquier otro que realice esa tarea. El resultado es un archivo de **código P**, el procedimiento estará creado.

- 3) Ejecutar el procedimiento para realizar su tarea, eso se puede hacer las veces que haga falta (en **iSQL*Plus**, el comando que ejecuta un procedimiento es el comando **EXECUTE**, en otros entornos se suele crear un bloque anónimo que llame al anterior).

(8.6.4) parámetros

Los procedimientos permiten utilizar parámetros para realizar su tarea. Por ejemplo supongamos que queremos crear el procedimiento **ESCRIBIR** para escribir en el servidor (como hace **DBMS_OUTPUT.PUT_LINE**) lógicamente dicho procedimiento necesita saber lo que queremos escribir. Ese sería el parámetro, de esa forma el procedimiento sería:

```
CREATE OR REPLACE PROCEDURE
Escribir(texto VARCHAR)
IS
BEGIN
    dbms_output.put_line(texto);
END;
```

Para invocarle:

```
BEGIN
...
Escribir('Hola');
```

Cuando se invoca a un procedimiento, si éste no tiene parámetros, se pueden omitir los paréntesis (es decir la llama al procedimiento **actualizar()** se puede hacer sin los paréntesis)

parámetros IN y parámetros OUT

Hay tres tipos de parámetros en PL/SQL:

- ◆ **Parámetros IN.** Son los parámetros que en otros lenguajes se denominan como parámetros por valor. El procedimiento recibe una copia del valor o variable que se utiliza como parámetro al llamar al procedimiento. Estos parámetros pueden ser: valores literales (**18** por ejemplo), variables (**v_num** por ejemplo) o expresiones (como **v_num+18**). A estos parámetros se les puede asignar un valor por defecto.
- ◆ **Parámetros OUT.** Relacionados con el paso por variable de otros lenguajes. Sólo pueden ser variables y no pueden tener un valor por defecto. Se utilizan para que el procedimiento almacene en ellas algún valor. Es decir, los parámetros OUT son variables sin declarar que se envían al procedimiento de modo que si en el procedimiento cambian su valor, ese valor permanece en ellas cuando el procedimiento termina,

- ◆ **Parámetros IN OUT.** Son una mezcla de los dos anteriores. Se trata de variables declaradas anteriormente cuyo valor puede ser utilizado por el procedimiento que, además, puede almacenar un valor en ellas. No se las puede asignar un valor por defecto.

Se pueden especificar estas palabras en la declaración del procedimiento (es el modo del procedimiento). Si no se indica modo alguno, se supone que se está utilizando IN (que es el que más se usa).

Ejemplo:

```
CREATE OR REPLACE PROCEDURE consultarEmpresa
(v_Nombre VARCHAR2, v_CIF OUT VARCHAR2, v_dir OUT
VARCHAR2)
IS
BEGIN
    SELECT cif, direccion INTO v_CIF, v_dir
    FROM EMPRESAS
    WHERE nombre LIKE '%' || v_nombre || '%';
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        dbms_output.put_line('No se encontraron datos');
    WHEN TOO_MANY_ROWS THEN
        dbms_output.put_line('Hay más de una fila con esos'
        || ' datos');
END;
```

El procedimiento consulta las empresas cuyo nombre tenga el texto enviado en **v_nombre**, captura los posibles errores y en caso de que la consulta sea buena almacena el cif y la dirección de la empresa en los parámetros **v_CIF** y **v_dir**.

La llamada al procedimiento anterior podría ser:

```
DECLARE
    v_c VARCHAR2(50);
    v_d VARCHAR2(50);
BEGIN
    consultarEmpresa('Hernández', v_c, v_d);
    DBMS_OUTPUT.PUT_LINE(v_c);
    DBMS_OUTPUT.PUT_LINE(v_d);
END;
```

Las variables **v_c** y **v_d** almacenarán (si existe una sola empresa con el texto **Hernández**) el CIF y la dirección de la empresa buscada.

Los procedimientos no pueden leer los valores que posean las variables OUT, sólo escribir en ellas. Si se necesitan ambas cosas es cuando hay que declararlas con el texto IN OUT.

(8.6.5) borrar procedimientos

El comando DROP PROCEDURE seguido del nombre del bloque que se elimina es el encargado de realizar esta tarea.

(8.7) funciones

(8.7.1) introducción

Las funciones son un tipo especial de procedimiento que sirven para calcular un determinado valor. Todo lo comentado en el apartado anterior es válido para las funciones, la diferencia estriba **sólo** en que éstas devuelven un valor.

(8.7.2) sintaxis

```
CREATE [OR REPLACE] FUNCTION nombreProcedimiento
[(parámetro1 [modelo] tipoDatos
  [,parámetro1 [modelo] tipoDatos [,...]])]
RETURN tipoDeDatos
IS|AS
  secciónDeDeclaraciones
BEGIN
  instrucciones
[EXCEPTION
  controlDeExcepciones]
END;
```

Si comparamos con la declaración de las funciones, la palabra PROCEDURE se modifica por la palabra FUNCTION (indicando que es una función y no un procedimiento) y aparece la cláusula RETURN justo antes de la palabra IS que sirve para indicar el tipo de datos que poseerá el valor retornado por la función.

(8.7.3) uso de funciones

Las funciones se crean igual que los procedimientos y, al igual que éstos, se almacenan en la base de datos. Toda función ha de devolver un valor, lo cual implica utilizar la instrucción **RETURN** seguida del valor que se devuelve.

Ejemplo:

```
CREATE OR REPLACE FUNCTION cuadrado
(x NUMBER)
RETURN NUMBER
IS
BEGIN
    RETURN x*x;
END;
```

La función descrita calcula el cuadrado de un número. Una llamada podría ser:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE( cuadrado(9) );
END;
```

Las funciones de PL/SQL se utilizan como las funciones de cualquier lenguaje estructurado, se pueden asignar a una variable, utilizar para escribir, etc. Además dentro de una función se puede invocar a otra función.

(8.7.4) utilizar funciones desde SQL

Una ventaja fantástica de las funciones es la posibilidad de utilizarlas desde una instrucción SQL. Por ejemplo:

```
CREATE OR REPLACE FUNCTION precioMedio
RETURN NUMBER
IS
v_precio NUMBER(11,4);
BEGIN
    SELECT AVG(precio_venta) INTO v_precio
    FROM PIEZAS;
    RETURN v_precio;
END;
```

Esta función devuelve el precio medio de la tabla de piezas. Una vez compilada y almacenada la función, se puede invocar desde una instrucción SQL cualquiera.

Por ejemplo:

```
SELECT * FROM PIEZAS  
WHERE precioMedio > precio_venta;
```

Esa consulta obtiene los datos de las piezas cuyo precio sea menor que el precio medio.

Hay que tener en cuenta que para que las funciones puedan ser invocadas desde SQL, éstas tienen que cumplir que:

- ◆ Sólo valen funciones que se hayan almacenado
- ◆ Sólo pueden utilizar parámetros de tipo IN
- ◆ Sus parámetros deben ser de tipos compatibles con el lenguaje SQL (no valen tipos específicos de PL/SQL como BOOLEAN por ejemplo)
- ◆ El tipo devuelto debe ser compatible con SQL
- ◆ No pueden contener instrucciones DML
- ◆ Si una instrucción DML modifica una determinada tabla, en dicha instrucción no se puede invocar a una función que realice consultas sobre la misma tabla
- ◆ No pueden utilizar instrucciones de transacciones (COMMIT, ROLLBACK,...)
- ◆ La función no puede invocar a otra función que se salte alguna de las reglas anteriores.

(8.7.5) eliminar funciones

Sintaxis:

```
DROP FUNCTION nombreFunción;
```

(8.7.6) recursividad

En PL/SQL la recursividad (el hecho de que una función pueda llamarse a sí misma) está permitida. Este código es válido:

```
CREATE FUNCTION Factorial
(n NUMBER)
IS
BEGIN
  IF (n<=1) THEN
    RETURN 1
  ELSE
    RETURN n * Factorial(n-1);
  END IF;
END;
```

(8.7.7) mostrar procedimientos almacenados

La vista **USER_PROCEDURES**, contiene una fila por cada procedimiento o función que tenga almacenado el usuario actual.

(8.8) paquetes

(8.8.1) introducción

Los paquetes sirven para agrupar bajo un mismo nombre funciones y procedimientos. Facilitan la modularización de programas y su mantenimiento.

Los paquetes constan de dos partes:

- ◆ **Especificación.** Que sirve para declarar los elementos de los que consta el paquete. En esta especificación se indican los procedimientos y variables **públicos** del paquete (los que se podrán invocar fuera del paquete). De los procedimientos sólo se indica su nombre y parámetros (sin el cuerpo).
- ◆ **Cuerpo.** En la que se especifica el funcionamiento del paquete. Consta de la definición de los procedimientos indicados en la especificación. Además se pueden declarar y definir variables y procedimientos **privados** (sólo visibles para el cuerpo del paquete, no se pueden invocar desde fuera del mismo).

Los paquetes se editan, se compilan (obteniendo su código P) y se ejecutan al igual que los procedimientos y funciones

(8.8.2) creación de paquetes

Conviene almacenar la especificación y el cuerpo del paquete en dos archivos de texto (**.sql**) distintos para su posterior mantenimiento.

especificación

Sintaxis:

```
CREATE [OR REPLACE] PACKAGE nombrePaquete
IS|AS
    variables, constantes, cursores y excepciones públicas
    cabecera de procedimientos y funciones
END nombrePaquete;
```

Ejemplo:

```
CREATE OR REPLACE PACKAGE paquetel IS
    v_cont NUMBER := 0;
    PROCEDURE reset_cont(v_nuevo_cont NUMBER);
    FUNCTION devolver_cont
        RETURN NUMBER;
END paquetel;
```

cuerpo

```
CREATE [OR REPLACE] PACKAGE BODY nombrePaquete
IS|AS
    variables, constantes, cursores y excepciones privadas
    cuerpo de los procedimientos y funciones
END nombrePaquete;
```

Ejemplo:

```
CREATE OR REPLACE PACKAGE BODY paquetel IS
    PROCEDURE reset_cont(v_nuevo_cont NUMBER)
    IS
    BEGIN
        v_cont:=v_new_cont;
    END reset_cont;

    FUNCTION devolver_cont
    IS
    BEGIN
        RETURN v_cont;
    END devolver_cont;
END paquetel;
```

uso de los objetos definidos en los paquetes

Desde dentro del paquete, para utilizar otra función o procedimiento o variable dentro del mismo paquete, basta con invocarla por su nombre.

Si queremos utilizar un objeto de un paquete, fuera del mismo, entonces se antepone el nombre del paquete a la función. Por ejemplo **paquete1.reset_cont(4)** (en el ejemplo anterior).

Para ejecutar un paquete desde SQL*Plus, se usa la orden **EXECUTE**. Por ejemplo: **EXECUTE paquete1.reset_cont(4)**

uso de cursores en los paquetes

Además las variables, en la cabecera del paquete se pueden definir cursores. Para ello se indica el nombre del cursor, los parámetros (si hay) y el tipo devuelto (normalmente con **%ROWTYPE**).

En el cuerpo del paquete el cursor se usa como habitualmente. La única razón para declarar un cursor en un paquete, es por si necesitamos acceder a él desde fuera.

(8.9) triggers

(8.9.1) introducción

Se llama **trigger** (o **disparador**) al código que se ejecuta automáticamente cuando se realiza una determinada acción sobre la base de datos. El código se ejecuta independientemente de la aplicación que realizó dicha operación.

De esta forma tenemos tres tipos triggers:

- ◆ **Triggers de tabla.** Se trata de triggers que se disparan cuando ocurre una acción DML sobre una tabla.
- ◆ **Triggers de vista.** Se lanzan cuando ocurre una acción DML sobre una vista.
- ◆ **Triggers de sistema.** Se disparan cuando se produce un evento sobre la base de datos (conexión de un usuario, borrado de un objeto,...)

En este manual sólo se da cabida a los del primer y segundo tipo. Por lo que se dará por echo en todo momento que nos referiremos siempre a ese tipo de triggers.

Los triggers se utilizan para:

- ◆ Ejecutar acciones relacionadas con la que **dispara** el trigger
- ◆ Centralizar operaciones globales
- ◆ Realizar tareas administrativas de forma automática
- ◆ Evitar errores
- ◆ Crear reglas de integridad complejas

El código que se lanza con el trigger es **PL/SQL**. No es conveniente realizar excesivos triggers, sólo los necesarios, de otro modo se ralentiza en exceso la base de datos.

(8.9.2) creación de triggers

elementos de los triggers

Puesto que un trigger es un código que se dispara, al crearle se deben indicar las siguientes cosas:

- (1) El evento que da lugar a la ejecución del trigger (**INSERT**, **UPDATE** o **DELETE**)
- (2) Cuando se lanza el evento en relación a dicho evento (**BEFORE** (antes), **AFTER** (después) o **INSTEAD OF** (en lugar de))
- (3) Las veces que el trigger se ejecuta (tipo de trigger: de instrucción o de fila)
- (4) El cuerpo del trigger, es decir el código que ejecuta dicho trigger

cuándo ejecutar el trigger

En el apartado anterior se han indicado los posibles tiempos para que el trigger se ejecute. Éstos pueden ser:

- ◆ **BEFORE.** El código del trigger se ejecuta antes de ejecutar la instrucción DML que causó el lanzamiento del trigger.
- ◆ **AFTER.** El código del trigger se ejecuta después de haber ejecutado la instrucción DML que causó el lanzamiento del trigger.
- ◆ **INSTEAD OF.** El trigger sustituye a la operación DML. Se utiliza para vistas que no admiten instrucciones DML.

tipos de trigger

Hay dos tipos de trigger

- ◆ **De instrucción.** El cuerpo del trigger se ejecuta una sola vez por cada evento que lance el trigger. Esta es la opción por defecto. El código se ejecuta aunque la instrucción DML no genere resultados.
- ◆ **De fila.** El código se ejecuta una vez por cada fila afectada por el evento. Por ejemplo si hay una cláusula **UPDATE** que desencadena un trigger y dicho **UPDATE** actualiza 10 filas; si el trigger es de fila se ejecuta una vez por cada fila, si es de instrucción se ejecuta sólo una vez.

(8.9.3) sintaxis de la creación de triggers

triggers de instrucción

```
CREATE [OR REPLACE] TRIGGER nombreDeTrigger
```

```
cláusulaDeTiempo evento1 [OR evento2[,...]]  
ON tabla  
[DECLARE  
    declaraciones  
]  
BEGIN  
    cuerpo  
[EXCEPTION  
    captura de excepciones  
]  
END;
```

La cláusula de tiempo es una de estas palabras: **BEFORE** o **AFTER**. Por su parte el evento tiene esta sintaxis:

```
{INSERT|UPDATE [OF columna1 [,columna2,...]]|DELETE}
```

Los eventos se realizan con la instrucción DML que desencadena el trigger. El apartado **OF** en el **UPDATE** hace que el trigger se ejecute sólo cuando se modifique la columna indicada (o columnas si se pone una lista separada por comas). En la sintaxis del trigger, el apartado **OR** permite asociar más de un evento al trigger (se puede indicar **INSERT OR UPDATE** por ejemplo).

Ejemplo:

```
CREATE OR REPLACE TRIGGER ins_personal  
BEFORE INSERT ON personal  
BEGIN  
    IF(TO_CHAR(SYSDATE,'HH24') NOT IN ('10','11','12'))  
    THEN RAISE_APPLICATION_ERROR(-20001,'Sólo se puede '  
    || 'añadir personal entre las 10 y las 12:59');  
    END IF;  
END;
```

Este trigger impide que se puedan añadir registros a la tabla de personal si no estamos entre las 10 y las 13 horas.

triggers de fila

Sintaxis:

```
CREATE [OR REPLACE] TRIGGER nombreDeTrigger  
cláusulaDeTiempo evento1 [OR evento2[,...]]  
ON tabla  
[REFERENCING {OLD AS nombreViejo | NEW AS nombreNuevo}]
```

```
FOR EACH ROW [WHEN condición]  
[declaraciones]  
cuerpo
```

La diferencia con respecto a los triggers de instrucción está en la línea **REFERENCING** y en **FOR EACH ROW**. Ésta última es la que hace que la instrucción se repita por cada registro afectado por la instrucción DML.

El apartado **WHEN** permite colocar una condición que deben de cumplir los registros para que el trigger se ejecute. Sólo se ejecuta el trigger para los registros que cumplan dicha condición.

El apartado **REFERENCING** es el que permite indicar un nombre para los valores antiguos y otro para los nuevos.

(8.9.4) referencias NEW y OLD

Cuando se ejecutan instrucciones UPDATE, hay que tener en cuenta que se modifican valores antiguos (OLD) para cambiarles por valores nuevos (NEW). Las palabras NEW y OLD permiten acceder a los valores nuevos y antiguos respectivamente.

El apartado **REFERENCING** de la creación de triggers, permite asignar nombres a las palabras **NEW** y **OLD** (en la práctica no se suele utilizar esta posibilidad). Así **NEW.nombre** haría referencia al nuevo nombre que se asigna a una determinada tabla y **OLD.nombre** al viejo.

En el apartado de instrucciones del trigger (el BEGIN) hay que adelantar el símbolo ":" a las palabra NEW y OLD (serían **:NEW.nombre** y **:OLD.nombre**)

Imaginemos que deseamos hacer una auditoria sobre una tabla en la que tenemos un listado de las piezas mecánicas que fabrica una determinada empresa. Esa tabla es PIEZAS y contiene el tipo y el modelo de la pieza (los dos campos forman la clave de la tabla) y el precio de venta de la misma. Deseamos almacenar en otra tabla diferente los cambios de precio que realizamos a las piezas, para lo cual creamos la siguiente tabla:

```
CREATE TABLE piezas_audit(  
  precio_viejo NUMBER(11,4),  
  precio_nuevo NUMBER(11,4),  
  tipo VARCHAR2(2),  
  modelo NUMBER(2),  
  fecha DATE  
);
```

Como queremos que la tabla se actualice automáticamente, creamos el siguiente trigger:

```
CREATE OR REPLACE TRIGGER crear_audit_piezas  
BEFORE UPDATE OF precio_venta
```



```
ON PIEZAS
FOR EACH ROW
WHEN ( OLD.precio_venta<NEW.precio_venta )
BEGIN
    INSERT INTO PIEZAS_AUDIT
    VALUES( :OLD.precio_venta, :NEW.precio_venta,
            :OLD.tipo, :OLD.modelo, SYSDATE ) ;
END;
```

Con este trigger cada vez que se modifiquen un registros de la tabla de piezas, siempre y cuando se esté incrementado el precio, se añade una nueva fila por registro modificado en la tabla de aditorías, observar el uso de NEW y de OLD y el uso de los dos puntos (:NEW y :OLD) en la sección ejecutable.

Cuando se añaden registros, los valores de OLD son todos nulos. Cuando se borran registros, son los valores de NEW los que se borran.

(8.9.5) IF INSERTING, IF UPDATING e IF DELETING

Son palabras que se utilizan para determinar la instrucción que se estaba realizando cuando se lanzó el trigger. Esto se utiliza en triggers que se lanza para varias operaciones (utilizando **INSERT OR UPDATE** por ejemplo). En ese caso se pueden utilizar sentencias IF seguidas de INSERTING, UPDATING o DELETING; éstas palabras devolverán **TRUE** si se estaba realizando dicha operación.

```
CREATE OR REPLACE TRIGGER trigger1
BEFORE INSERT OR DELETE OR UPDATE OF campol ON tabla
FOR EACH ROW
BEGIN
    IF DELETING THEN
        . . . .
    ELSIF INSERTING THEN
        . . . .
    ELSE --estará actualizando, UPDATING
        . . .
    END IF
END;
```

(8.9.6) triggers de tipo INSTEAD OF

Hay un tipo de trigger especial que se llama INSTEAD OF y que sólo se utiliza con las vistas. Una vista es una consulta SELECT almacenada. En general sólo sirven para mostrar datos, pero podrían ser interesantes para actualizar, por ejemplo en esta declaración de vista:

```
CREATE VIEW
```

Sistemas Gestores de Bases de Datos

(Unidad 8) PL/SQL

```
existenciasCompleta(tipo,modelo,precio,
almacen,cantidad) AS
SELECT p.tipo, p.modelo, p.precio_venta,
e.n_almacen, e.cantidad
FROM PIEZAS p, EXISTENCIAS e
WHERE p.tipo=e.tipo AND p.modelo=e.modelo
ORDER BY p.tipo,p.modelo,e.n_almacen;
```

Esta instrucción daría lugar a error

```
INSERT INTO existenciasCompleta
VALUES( 'ZA' ,3,4,3,200);
```

Indicando que esa operación no es válida en esa vista (al utilizar dos tablas). Esta situación la puede arreglar un trigger que inserte primero en la tabla de piezas (sólo si no se encuentra ya insertada esa pieza) y luego inserte en existencias.

Eso lo realiza el trigger de tipo INSTEAD OF, que sustituirá el INSERT original por el indicado por el trigger:

```
CREATE OR REPLACE TRIGGER ins_piezas_exis
INSTEAD OF INSERT
ON existenciascompleta
BEGIN
    INSERT INTO piezas(tipo,modelo,precio_venta)
        VALUES( :NEW.tipo, :NEW.modelo, :NEW.precio);
    INSERT INTO
existencias(tipo,modelo,n_almacen,cantidad)
        VALUES( :NEW.tipo, :NEW.modelo,
                :NEW.almacen, :NEW.cantidad);
END;
```

Este trigger permite añadir a esa vista añadiendo los campos necesarios en las tablas relacionadas en la vista. Se podría modificar el trigger para permitir actualizar, eliminar o borrar datos directamente desde la vista y así cualquier desde cualquier acceso a la base de datos se utilizaría esa vista como si fuera una tabla más.

(8.9.7) administración de triggers

eliminar triggers

Sintaxis:

```
DROP TRIGGER nombreTrigger;
```

desactivar un trigger

Desactiva un trigger temporalmente (se puede activar en cualquier momento).

```
ALTER TRIGGER nombreTrigger DISABLE;
```

activar un trigger

Sintaxis:

```
ALTER TRIGGER nombreTrigger ENABLE;
```

desactivar o activar todos los triggers de una tabla

Eso permite en una sola instrucción operar con todos los triggers relacionados con una determinada tabla (es decir actúa sobre los triggers que tienen dicha tabla en el apartado ON del trigger). Sintaxis:

```
ALTER TABLE nombreTabla {DISABLE | ENABLE} ALL TRIGGERS;
```

(8.9.8) restricciones de los triggers

- ◆ Un trigger no puede utilizar ninguna operación de transacciones (**COMMIT**, **ROLLBACK**, **SAVEPOINT** o **SET TRANSACTION**)
- ◆ No puede llamar a ninguna función o procedimiento que utilice operaciones de transacciones
- ◆ No se pueden declarar variables **LONG** o **LONG RAW**. Las referencias **:NEW** y **:OLD** no pueden hacer referencia a campos de estos tipos de una tabla
- ◆ No se pueden modificar (aunque sí leer y hacer referencia) campos de tipo **LOB** de una tabla

(8.9.9) orden de ejecución de los triggers

Puesto que sobre una misma tabla puede haber varios triggers, es necesario conocer en qué orden se ejecutan los mismos. El orden es:

- (1) Primero disparadores de tipo **BEFORE** de tipo instrucción
- (2) Disparadores de tipo **BEFORE** por cada fila
- (3) Se ejecuta la propia orden.
- (4) Disparadores de tipo **AFTER** con nivel de fila.
- (5) Disparadores de tipo **AFTER** con nivel de instrucción.

(8.9.10) problemas con las tablas mutantes

Una tabla mutante es una tabla que se está modificando debido a una instrucción DML. En el caso de un disparador es la tabla a la que se refiere el disparador en la cláusula **ON**.

Un disparador no puede leer o modificar una tabla mutante ni tampoco leer o modificar una columna de clave externa relativa a una restricción de la tabla a la que se refiere el trigger (sí se podría utilizar y modificar el resto de columnas). Es decir, no podemos hacer consultas ni instrucciones DML sobre una tabla sobre la que ya se ha comenzado a modificar, insertar o eliminar datos.

En realidad **sólo los triggers de fila** no pueden acceder a tablas mutantes. Por ello la solución al problema suele ser combinar triggers de tabla con triggers de fila (conociendo el orden de ejecución de los triggers). Normalmente los triggers de fila almacenan los valores que se desean cambiar dentro de una tabla preparada al efecto o dentro de variables de paquete (accesibles desde cualquier trigger si se crea dentro de dicho paquete).