

curso completo de

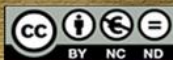


Gestión de Bases de Datos

Usando Oracle SQL y PL/SQL

Apuntes realizados para la asignatura de FP Grado Superior:
Gestión de Bases de Datos
del ciclo Administración de Sistemas Informáticos en Red

Autor: Jorge Sánchez Asenjo (www.jorgesanchez.net)
Versión del documento: 2.4, Año 2013



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/3.0/deed.es>



Atribución-NoComercial-CompartirIgual 3.0 Unported (CC BY-NC-SA 3.0)

Esto es un resumen fácilmente legible del [Texto Legal \(la licencia completa\)](#).

<http://creativecommons.org/licenses/by-nc-sa/3.0/legalcode>

Usted es libre de:

Compartir - copiar, distribuir, ejecutar y comunicar públicamente la obra
hacer obras derivadas

Bajo las condiciones siguientes:



Atribución — Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciante (pero no de una manera que sugiera que tiene su apoyo o que 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.

Entendiendo que:

Renuncia — Alguna de estas condiciones puede **no aplicarse** si se obtiene el permiso del titular de los derechos de autor

Dominio Público — Cuando la obra o alguno de sus elementos se halle en el **dominio público** según la ley vigente aplicable, esta situación no quedará afectada por la licencia.

Otros derechos — Los derechos siguientes no quedan afectados por la licencia de ninguna manera:

- Los derechos derivados de **usos legítimos** u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.
- Los derechos **morales** del autor;
- Derechos que pueden ostentar otras personas sobre la propia obra o su uso, como por ejemplo **derechos de imagen** o de privacidad.

índice

(1) gestión y diseño de bases de datos	9
(1.1) datos y archivos	9
(1.1.1) la necesidad de gestionar datos	9
(1.1.2) sistemas de información.....	10
(1.1.3) archivos	11
(1.1.4) operaciones relacionadas con uso de ficheros en bases de datos.....	16
(1.1.5) tipos de sistemas de información.....	17
(1.1.6) utilidad de los sistemas gestores de bases de datos.....	19
(1.1.7) niveles de abstracción de una base de datos	20
(1.2) componentes de los SGBD	21
(1.2.1) funciones. lenguajes de los SGBD	21
(1.2.2) recursos humanos de las bases de datos.....	22
(1.2.3) estructura multicapa.....	23
(1.2.4) funcionamiento del SGBD.....	25
(1.3) arquitectura de los SGBD. estándares	25
(1.3.1) organismos de estandarización.....	25
(1.3.2) ISO/JTC1/SC21/WG3.....	26
(1.3.3) DBTG/Codasyl	26
(1.3.4) ANSI/X3/SPARC	26
(1.3.5) Modelo ANSI/X3/SPARC	27
(1.3.6) proceso de creación y manipulación de una base de datos actual	30
(1.3.7) formas de ejecución de un SGBD	30
(1.4) tipos de SGBD	31
(1.4.1) introducción	31
(1.4.2) modelo jerárquico	32
(1.4.3) modelo en red (Codasyl)	33
(1.4.4) modelo relacional.....	33
(1.4.5) modelo de bases de datos orientadas a objetos	34
(1.4.6) bases de datos objeto-relacionales	34
(1.4.7) bases de datos NoSQL.....	34
(1.5) diseño conceptual de bases de datos. el modelo entidad / relación _	35
(1.5.1) introducción	35
(1.5.2) componentes del modelo	35
(1.5.3) relaciones	37
(1.5.4) atributos.....	40

(1.5.5) modelo entidad relación extendido	41
(2) bases de datos relacionales.....	49
(2.1) el modelo relacional _____	49
(2.1.1) introducción	49
(2.1.2) objetivos.....	50
(2.1.3) historia del modelo relacional.....	50
(2.2) estructura de las bases de datos relacionales _____	52
(2.2.1) relación o tabla.....	52
(2.2.2) tupla	52
(2.2.3) dominio	53
(2.2.4) grado	53
(2.2.5) cardinalidad.....	53
(2.2.6) sinónimos	53
(2.2.7) definición formal de relación	54
(2.2.8) propiedades de las tablas (o relaciones)	54
(2.2.9) tipos de tablas	55
(2.2.10) claves.....	55
(2.2.11) nulos.....	56
(2.3) restricciones _____	57
(2.3.1) inherentes	57
(2.3.2) semánticas	57
(2.4) las 12 reglas de Codd _____	59
(2.5) paso de entidad/relación de Chen al modelo relacional _____	60
(2.5.1) transformación de las entidades fuertes.....	60
(2.5.2) transformación de relaciones.....	60
(2.5.3) entidades débiles	65
(2.5.4) relaciones ISA.....	65
(2.5.5) notas finales.....	66
(2.6) representación de esquemas de bases de datos relacionales _____	66
(2.6.1) grafos relacionales.....	67
(2.6.2) esquemas relacionales derivados del modelo entidad/relación.....	67
(2.7) normalización _____	70
(2.7.1) problemas del esquema relacional	70
(2.7.2) formas normales.....	71
(2.7.3) primera forma normal (1FN).....	72
(2.7.4) dependencias funcionales.....	72
(2.7.5) segunda forma normal (2FN)	73
(2.7.6) tercera forma normal (3FN).....	74
(2.7.7) forma normal de Boyce-Codd (FNBC o BCFN).....	75
(2.7.8) cuarta forma normal (4FN). dependencias multivaluadas.....	76
(2.7.9) quinta forma normal (5FN)	77

(2.7.10) forma normal de dominio clave (FNDC)	79
(3) SQL (I). DDL y DML	81
(3.1) notas previas	81
(3.1.1) versión de SQL.....	81
(3.1.2) formato de las instrucciones en los apuntes.....	81
(3.2) introducción	83
(3.2.1) objetivos	83
(3.2.2) historia del lenguaje SQL.....	83
(3.2.3) funcionamiento	84
(3.2.4) proceso de las instrucciones SQL.....	85
(3.3) elementos del lenguaje SQL	85
(3.3.1) código SQL	85
(3.3.2) normas de escritura.....	86
(3.4) DDL	86
(3.4.1) introducción	86
(3.4.2) creación de bases de datos.....	86
(3.4.3) objetos de la base de datos.....	87
(3.4.4) creación de tablas	87
(3.4.5) tipos de datos	88
(3.4.6) dominios.....	92
(3.4.7) consultar las tablas del usuario	93
(3.4.8) borrar tablas	94
(3.4.9) modificar tablas.....	95
(3.4.10) restricciones.....	97
(3.5) DML	106
(3.5.1) introducción	106
(3.5.2) inserción de datos	106
(3.5.3) actualización de registros	107
(3.5.4) borrado de registros	108
(3.6) transacciones	109
(3.6.2) COMMIT	109
(3.6.3) ROLLBACK.....	109
(3.6.4) estado de los datos durante la transacción	109
(3.7) otras instrucciones DDL	110
(3.7.1) índices.....	110
(3.7.2) secuencias.....	111
(3.7.3) sinónimos	113
(4) SQL (II). Consultas	115
(4.1) consultas de datos con SQL. DQL	115
(4.1.1) capacidades.....	115

(4.1.2) sintaxis sencilla del comando SELECT	115
(4.2) cálculos	116
(4.2.1) aritméticos.....	116
(4.2.2) concatenación de textos	116
(4.3) condiciones	117
(4.3.1) operadores de comparación.....	117
(4.3.2) valores lógicos	118
(4.3.3) BETWEEN.....	118
(4.3.4) IN	119
(4.3.5) LIKE.....	119
(4.3.6) IS NULL	119
(4.3.7) precedencia de operadores.....	120
(4.4) ordenación	120
(4.5) funciones	121
(4.5.1) funciones	121
(4.5.2) funciones numéricas.....	122
(4.5.3) funciones de caracteres	123
(4.5.4) funciones de trabajo con nulos.....	125
(4.5.5) funciones de expresiones regulares.....	126
(4.5.6) funciones de fecha y manejo de fechas e intervalos	131
(4.5.7) funciones de conversión.....	133
(4.5.8) función DECODE.....	135
(4.5.9) funciones XML	136
(4.6) obtener datos de múltiples tablas	141
(4.6.1) producto cruzado o cartesiano de tablas	141
(4.6.2) asociando tablas.....	142
(4.6.3) relaciones sin igualdad.....	142
(4.6.4) sintaxis SQL 1999.....	143
(4.7) agrupaciones	146
(4.7.1) funciones de cálculo con grupos.....	147
(4.7.2) condiciones HAVING.....	148
(4.8) subconsultas	149
(4.8.1) uso de subconsultas simples	149
(4.8.2) uso de subconsultas de múltiples filas	150
(4.8.3) consultas EXISTS.....	151
(4.9) combinaciones especiales	152
(4.9.1) uniones	152
(4.9.2) intersecciones	152
(4.9.3) diferencia.....	152
(4.10) consultas avanzadas	153
(4.10.1) consultas con ROWNUM.....	153

(4.10.2) consultas sobre estructuras jerárquicas	154
(4.11) consultas de agrupación avanzada	157
(4.11.1) ROLLUP.....	157
(4.11.2) CUBE	158
(4.11.3) GROUPING.....	160
(4.11.4) GROUPING SETS.....	162
(4.11.5) conjuntos de agrupaciones combinadas	163
(4.12) DQL en instrucciones DML	163
(4.12.1) relleno de registros a partir de filas de una consulta	163
(4.12.2) subconsultas en la instrucción UPDATE.....	164
(4.12.3) subconsultas en la instrucción DELETE	165
(4.13) vistas	165
(4.13.1) introducción	165
(4.13.2) creación de vistas.....	166
(4.13.3) mostrar la lista de vistas	167
(4.13.4) borrar vistas	167
(5) PL/SQL	169
(5.1) introducción al SQL procedimental	169
(5.1.2) funciones que pueden realizar los programas PL/SQL.....	169
(5.1.3) conceptos básicos	170
(5.2) escritura de PL/SQL	170
(5.2.1) estructura de un bloque PL/SQL.....	170
(5.2.2) escritura de instrucciones PL/SQL.....	171
(5.3) variables	172
(5.3.1) uso de variables	172
(5.3.2) DBMS_OUTPUT.PUT_LINE.....	174
(5.3.3) alcance de las variables	174
(5.3.4) operadores y funciones.....	175
(5.3.5) paquetes estándar	175
(5.4) instrucciones SQL permitidas	176
(5.4.1) instrucciones SELECT en PL/SQL	176
(5.4.2) instrucciones DML y de transacción	176
(5.5) instrucciones de control de flujo	177
(5.5.1) instrucción IF	177
(5.5.2) sentencia CASE.....	179
(5.5.3) bucles	179
(5.6) cursores	182
(5.6.1) introducción	182
(5.6.2) procesamiento de cursores	182
(5.6.3) declaración de cursores	182

(5.6.4) apertura de cursores.....	182
(5.6.5) instrucción FETCH	183
(5.6.6) cerrar el cursor.....	183
(5.6.7) atributos de los cursores	183
(5.6.8) variables de registro.....	185
(5.6.9) cursores y registros	186
(5.6.10) cursores avanzados	187
(5.7) excepciones	189
(5.7.1) introducción.....	189
(5.7.2) captura de excepciones.....	190
(5.7.3) excepciones predefinidas.....	190
(5.7.4) excepciones sin definir	192
(5.7.5) funciones de uso con excepciones	193
(5.7.6) excepciones de usuario	193
(5.8) procedimientos	194
(5.8.1) introducción	194
(5.8.2) estructura de un procedimiento	194
(5.8.3) desarrollo de procedimientos.....	195
(5.8.4) parámetros.....	195
(5.8.5) borrar procedimientos	197
(5.9) funciones	197
(5.9.1) introducción	197
(5.9.2) sintaxis	198
(5.9.3) uso de funciones.....	198
(5.9.4) utilizar funciones desde SQL.....	199
(5.9.5) eliminar funciones	200
(5.9.6) recursividad	200
(5.9.7) mostrar procedimientos almacenados.....	200
(5.10) paquetes	200
(5.10.1) introducción.....	200
(5.10.2) creación de paquetes.....	201
(5.11) triggers	202
(5.11.1) introducción.....	202
(5.11.2) creación de triggers.....	203
(5.11.3) sintaxis de la creación de triggers.....	204
(5.11.4) referencias NEW y OLD.....	205
(5.11.5) IF INSERTING, IF UPDATING e IF DELETING.....	206
(5.11.6) triggers de tipo INSTEAD OF	207
(5.11.7) administración de triggers.....	208
(5.11.8) restricciones de los triggers	208
(5.11.9) orden de ejecución de los triggers.....	208
(5.11.10) problemas con las tablas mutantes	209

(1) gestión y diseño de bases de datos

(1.1) datos y archivos

(1.1.1) 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 Internet.

El propio nombre **Informática** hace referencia al hecho de ser una ciencia que trabaja con información. Desde los albores de la creación de ordenadores, la información se ha considerado como uno de los pilares de las computadoras digitales. Por ello las bases de datos son una de las aplicaciones más antiguas de la informática.

En informática se conoce como **dato** a cualquier elemento informativo que tenga relevancia para el sistema. Desde el inicio de la informática se ha reconocido al dato como al elemento fundamental de trabajo en un ordenador. Por ello se han realizado numerosos estudios y aplicaciones para mejorar la gestión que desde las computadoras se realiza de los datos.

Inicialmente los datos que se necesitaba almacenar y gestionar eran pocos, pero poco a poco han ido creciendo. En la actualidad las numerosas aplicaciones de Internet han producido enormes sistemas de información que incluso para poder gestionarles requieren decenas de máquinas haciendo la información accesible desde cualquier parte del planeta y en un tiempo rápido. Eso ha requerido que la ciencia de las bases de datos esté en continua renovación para hacer frente a esas enormes necesidades.

Pero incluso podemos remontarnos más al hablar de datos. El ser humano desde siempre ha necesitado gestionar datos; de esta forma se controlaban almacenes de alimentos, controles de inventario y otras muchos sistemas de datos. Como herramienta el ser humano al principio sólo poseía su memoria y cálculo y como mucho la ayuda de sus dedos.

La escritura fue la herramienta que permitió al ser humano poder gestionar bases cada vez más grandes de datos. Además de permitir compartir esa información entre diferentes personas, también posibilitó que los datos se guardaran de manera continua e incluso estuvieran disponibles para las siguientes generaciones. Los problemas actuales con la privacidad ya aparecieron con la propia escritura y así el cifrado de

datos es una técnica tan antigua como la propia escritura para conseguir uno de los todavía requisitos fundamentales de la gestión de datos, la **seguridad**.

Para poder almacenar datos y cada vez más datos, el ser humano ideó nuevas herramientas archivos, cajones, carpetas y fichas en las que se almacenaban los datos.

Antes de la aparición del ordenador, el tiempo 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 adaptó sus herramientas para que los elementos que el usuario maneja en el ordenador se parezcan a los que utilizaba manualmente. Así en informática se sigue hablado de ficheros, formularios, carpetas, directorios,....

(1.1.2) sistemas de información

la empresa como sistema

Según la RAE, la definición de sistema es *“Conjunto de cosas que ordenadamente relacionadas entre sí contribuyen a un determinado objeto”* .

La clientela fundamental del profesional de la informática es la empresa. La empresa se puede entender como un sistema formado por diversos objetos: el **capital**, los **recursos humanos**, los **inmuebles**, los **servicios** que presta, etc.

El sistema completo que forma la empresa, por otra parte, se suele dividir en los siguientes subsistemas:

- **Subsistema productivo.** También llamado subsistema real o físico. Representa la parte de la empresa encargada de gestionar la producción de la misma.
- **Subsistema financiero.** Encargado de la gestión de los bienes económicos de la empresa
- **Subsistema directivo.** Encargado de la gestión organizativa de la empresa

Hay que hacer notar que cada subsistema se asocia a un departamento concreto de la empresa.

sistemas de información

Los sistemas que aglutinan los elementos que intervienen para gestionar la información que manejan los subsistemas empresariales es lo que se conoce como Sistemas de Información. Se suele utilizar las siglas **SI** o **IS** (de *Information Server*) para referirse a ello).

Realmente un sistema de información sólo incluye la información que nos interesa de la empresa y los elementos necesarios para gestionar esa información.

Un sistema de información genérico está formado por los siguientes elementos:

- **Recursos físicos.** Carpetas, documentos, equipamiento, discos,...
- **Recursos humanos.** Personal que maneja la información
- **Protocolo.** Normas que debe cumplir la información para que sea manejada (formato de la información, modelo para los documentos,...)

Las empresas necesitan implantar estos sistemas de información debido a la competencia que las obliga a gestionar de la forma más eficiente sus datos para una mayor calidad en la organización de las actividades de los subsistemas empresariales.

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 electrónico**), los componentes son:

- **Datos.** Se trata de la información relevante que almacena y gestiona el sistema de información. Ejemplos de datos son: *Sánchez*, *12764569F*, *Calle Mayo 5*, *Azul...*
- **Hardware.** Equipamiento físico que se utiliza para gestionar los datos. cada uno de los dispositivos electrónicos que permiten el funcionamiento del sistema de información.
- **Software.** Aplicaciones informáticas que se encargan de la gestión de la base de datos y de las herramientas que facilitan su uso.
- **Recursos humanos.** Personal que maneja el sistema de información.

(1.1.3) archivos

Los ficheros o archivos son la herramienta fundamental de trabajo en una computadora todavía a día de hoy. Las computadoras siguen almacenando la información en ficheros, eso sí de estructura cada vez más compleja.

Los datos deben de ser almacenados en componentes de almacenamiento permanente, lo que se conoce como **memoria secundaria** (discos duros u otras unidades de disco). En esas memorias, los datos se estructuran en archivos (también llamados ficheros).

Un fichero es una secuencia de números binarios que organiza información relacionada a un mismo aspecto.

En general sobre los archivos se pueden realizar las siguientes operaciones:

- **Abrir (*open*).** Prepara el fichero para su proceso.
- **Cerrar (*close*).** Cierra el fichero impidiendo su proceso inmediato.
- **Leer (*read*).** Obtiene información del fichero.
- **Escribir (*write*).** Graba información en el fichero.
- **Posicionarse (*seek*).** Coloca el puntero de lectura en una posición concreta del mismo (no se puede realizar en todos los tipos de ficheros).
- **Fin de fichero (*eof*).** Indica si hemos llegado al final del fichero.

Cuando los ficheros almacenan datos, se dice que constan de **registros**. Cada registro contiene datos relativos a un mismo elemento u objeto. Por ejemplo en un fichero de personas, cada registro contiene datos de una persona. Si el archivo contiene datos de 1000 personas, constará de 1000 registros.

A continuación se explican los tipos más habituales de ficheros.

ficheros secuenciales

En estos ficheros, los datos se organizan secuencialmente en el orden en el que fueron grabados. Para leer los últimos datos hay que leer los anteriores. Es decir leer el registro número nueve, implica leer previamente los ocho anteriores.

ventajas

- Rápidos para obtener registros contiguos de una base de datos
- No hay huecos en el archivo al grabarse los datos seguidos, datos más compactos.

desventajas

- Consultas muy lentas al tener que leer todos los datos anteriores al dato que queremos leer
- Algoritmos de lectura y escritura más complejos
- No se pueden eliminar registros del fichero (se pueden marcar de manera especial para que no sean tenidos en cuenta, pero no se pueden borrar)
- El borrado provoca archivos que no son compactos
- La ordenación de los datos requiere volver a crearle de nuevo

ficheros de acceso directo o aleatorio

Se puede leer una posición concreta del fichero, con saber la posición (normalmente en bytes) del dato a leer. Cuando se almacenan registros, posicionarnos en el quinto registro se haría de golpe, lo único necesitamos saber el tamaño del registro, que en este tipo de ficheros debe de ser el mismo. Suponiendo que cada registro ocupa 100 bytes, el quinto registro comienza en la posición 400. Lo que se hace es colocar el llamado **puntero de archivo** en esa posición y después leer.

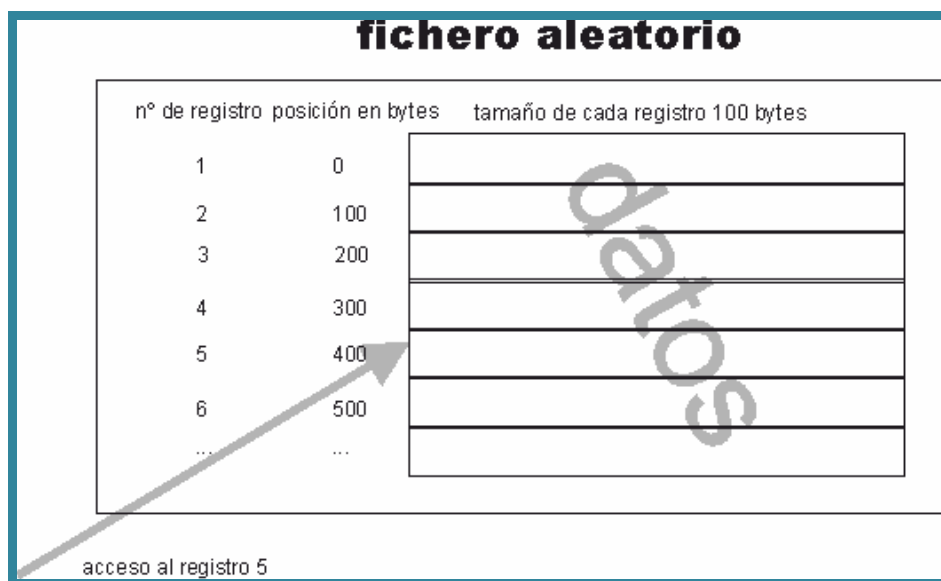


Ilustración 1, Ejemplo de fichero de acceso directo

ventajas

- Acceso rápido al no tener que leer los datos anteriores
- La modificación de datos es más sencilla
- Permiten acceso secuencial
- Permiten leer y escribir a la vez
- Aptos para organizaciones **relativas directas**, en las que la clave del registro se relaciona con su posición en el archivo

desventajas

- Salvo en archivos relativos directos, no es apto por sí mismo para usar en bases de datos, ya que los datos se organizan en base a una clave
- No se pueden borrar datos (sí marcar para borrado, pero generarán huecos)
- Las consultas sobre multitud de registros son más lentas que en el caso anterior.

ficheros secuenciales encadenados

Son ficheros secuenciales gestionados mediante punteros, datos especiales que contienen la dirección de cada registro del fichero. Cada registro posee ese puntero que indica la dirección del siguiente registro y que se puede modificar en cualquier momento. El puntero permite recorrer los datos en un orden concreto.

Cuando aparece un nuevo registro, se añade al final del archivo, pero los punteros se reordenan para que se mantenga el orden.

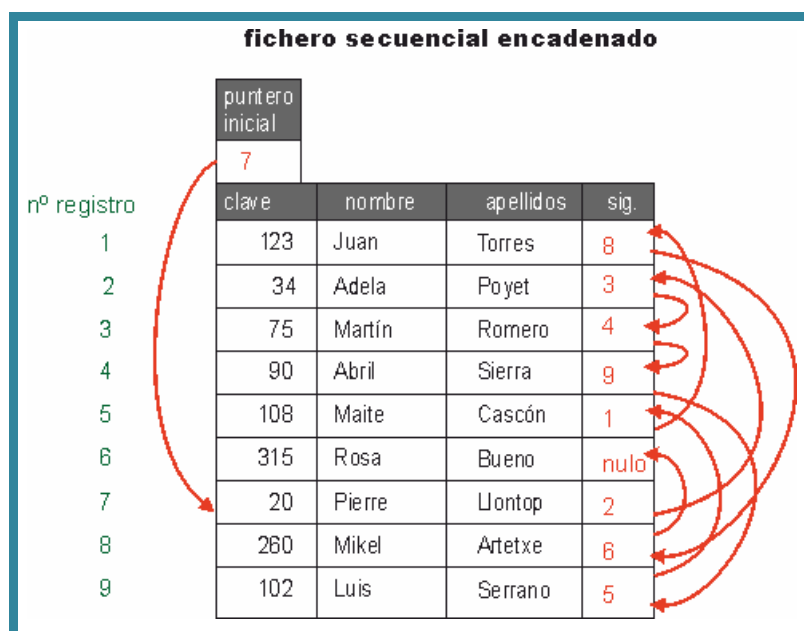


Ilustración 2, ejemplo de fichero secuencial encadenado. Los punteros le recorren por la clave

ventajas

- El fichero mantiene el orden en el que se añadieron los registros y un segundo orden en base a una clave
- La ordenación no requiere reorganizar todo el fichero, sino sólo modificar los punteros
- Las mismas ventajas que el acceso secuencial
- En esta caso sí se borran los registros y al reorganizar, se perderán definitivamente

desventajas

- No se borran los registros, sino que se marcan para ser ignorados. Por lo que se malgasta espacio
- Añadir registros o modificar las claves son operaciones que requieren recalcular los punteros

ficheros secuenciales indexados

Se utilizan dos ficheros para los datos, uno posee los registros almacenados de forma secuencial, pero que permite su acceso aleatorio. El otro posee una tabla con punteros a la posición ordenada de los registros. Ese segundo fichero es el **índice**, una tabla con la ordenación deseada para los registros y la posición que ocupan en el archivo.

El archivo de índices posee unas cuantas entradas sólo en las que se indica la posición de ciertos valores claves en el archivo (cada 10, 15, 20, ... registros del archivo principal se añade una entrada en el de índices). El archivo principal tiene que estar siempre ordenado y así cuando se busca un registro, se busca su valor clave en la tabla de índices, la cual poseerá la posición del registro buscado. Desde esa posición se busca secuencialmente el registro hasta encontrarlo.

Existe un archivo llamado de **desbordamiento** u **overflow** en el que se colocan los nuevos registros que se van añadiendo (para no tener que ordenar el archivo principal cada vez que se añade un nuevo registro) este archivo está desordenado. Se utiliza sólo si se busca un registro y no se encuentra en el archivo principal. En ese caso se recorre todo el archivo de overflow hasta encontrarlo.

Para no tener demasiados archivos en overflow (lo que restaría velocidad), cada cierto tiempo se reorganiza el archivo principal. Ejemplo:



Ilustración 3, ejemplo de fichero secuencial indexado

ventajas

- El archivo está siempre ordenado en base a una clave
- La búsqueda de datos es rapidísima
- Permite la lectura secuencial (que además será en el orden de la clave)
- El borrado de registros es posible (aunque más problemático que en el caso anterior)

desventajas

- Para un uso óptimo hay que reorganizar el archivo principal y esta operación es muy costosa ya que hay que reescribir de nuevo y de forma ordenada todo el archivo.
- La adición de registros requiere más tiempo que en los casos anteriores al tener que reordenar los índices

ficheros indexado-encadenados

Utiliza punteros e índices, es una variante encadenada del caso anterior. Hay un fichero de índices equivalente al comentado en el caso anterior y otro fichero de tipo encadenado con punteros a los siguientes registros. Cuando se añaden registros se añaden en un tercer registro llamado de desbordamiento u *overflow*. En ese archivo los datos se almacenan secuencialmente, se accede a ellos si se busca un dato y no se encuentra en la tabla de índices.

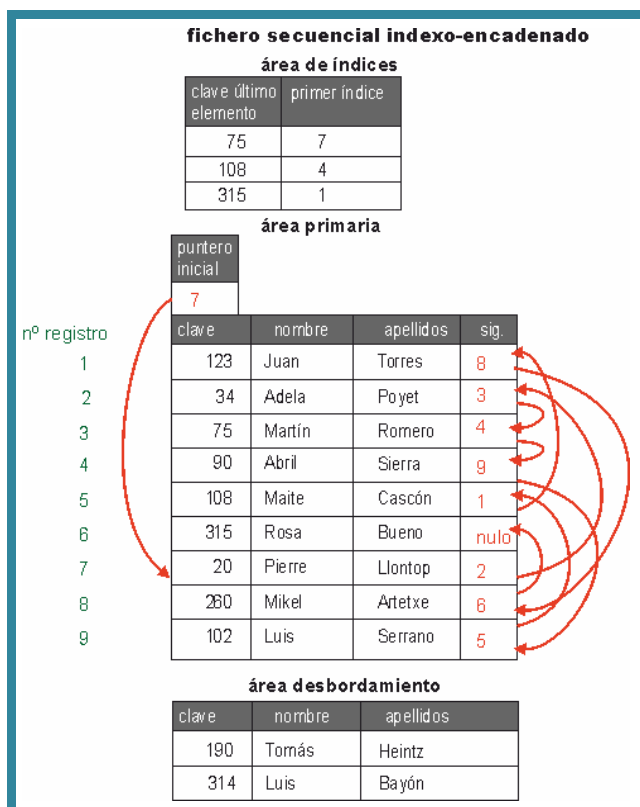


Ilustración 4, Ejemplo de archivo secuencial indexado y encadenado

ventajas

- Posee las mismas ventajas que los archivos secuenciales indexados, además de una mayor rapidez al reorganizar el fichero (sólo se modifican los punteros)

desventajas

- Requieren compactar los datos a menudo para reorganizar índices y quitar el fichero de desbordamiento.

(1.1.4) operaciones relacionadas con uso de ficheros en bases de datos

borrado y recuperación de registros

Algunos de los tipos de ficheros vistos anteriormente no admiten el borrado real de datos, sino que sólo permiten añadir un dato que indica si el registro está borrado o no. Esto es interesante ya que permite anular una operación de borrado. Por ello esta técnica de marcar registros, se utiliza casi siempre en todos los tipos de archivos.

En otros casos los datos antes de ser eliminados del todo pasan a un fichero especial (conocido como **papelera**) en el que se mantienen durante cierto tiempo para su posible recuperación.

fragmentación y compactación de datos

La fragmentación en un archivo hace referencia a la posibilidad de que éste tenga huecos interiores debido a borrado de datos u a otras causas. Causa los siguientes problemas:

- Mayor espacio de almacenamiento
- Lentitud en las operaciones de lectura y escritura del fichero

Por ello se requiere **compactar** los datos. Esta técnica permite eliminar los huecos interiores a un archivo. Las formas de realizarla son:

- **Reescribir el archivo para eliminar los huecos.** Es la mejor, pero lógicamente es la más lenta al requerir releer y reorganizar todo el contenido del fichero.
- **Aprovechar huecos.** De forma que los nuevos registros se inserten en esos huecos. Esta técnica suele requerir un paso previo para reorganizar esos huecos.

compresión de datos

En muchos casos para ahorrar espacio de almacenamiento, se utilizan técnicas de compresión de datos. La ventaja es que los datos ocupan menos espacio y la desventaja es que al manipular los datos hay que descomprimirlos lo que hace que la manipulación de los datos sea lenta.

cifrado de datos

Otra de las opciones habituales sobre ficheros de datos es utilizar técnicas de cifrado para proteger los ficheros en caso de que alguien no autorizado se haga con el fichero. Para leer un fichero de datos, haría falta descifrar el fichero. Para descifrar necesitamos una clave o bien aplicar métodos de descifrado; lógicamente cuanto mejor sea la técnica de cifrado, más difícil será descifrar los datos mediante la fuerza bruta.

(1.1.5) 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 son 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 la seguridad del sistema.** Ya que cada aplicación se crea independientemente; es por tanto muy difícil establecer criterios de seguridad uniformes.



Ilustración 5, Sistemas de Información orientados al proceso

A estos sistemas se les llama sistemas de gestión de ficheros. Se consideran también así a los sistemas que utilizan programas ofimáticos (como **Word** o **Excel** por ejemplo) para gestionar sus datos (muchas pequeñas empresas utilizan esta forma de administrar 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.

Cuando una aplicación modifica un dato, dicho dato la modificación será visible para el resto de aplicaciones.

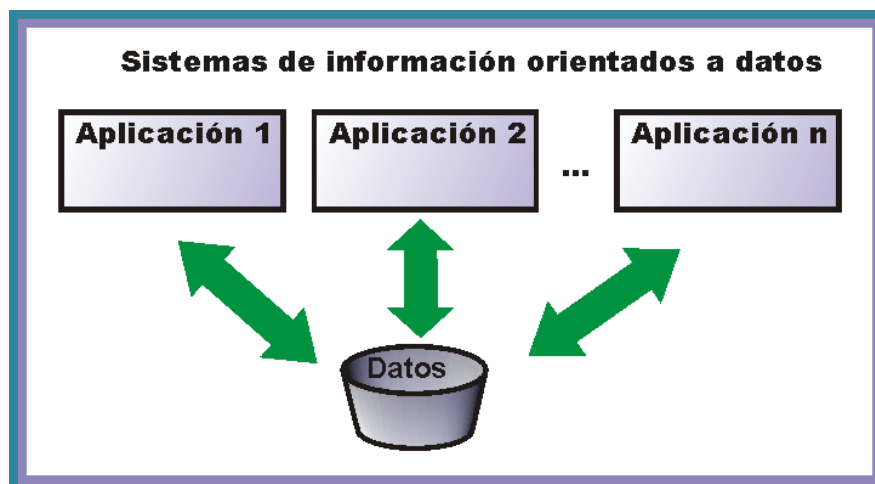


Ilustración 6, Sistemas de información orientados a datos

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.

(1.1.6) utilidad 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.

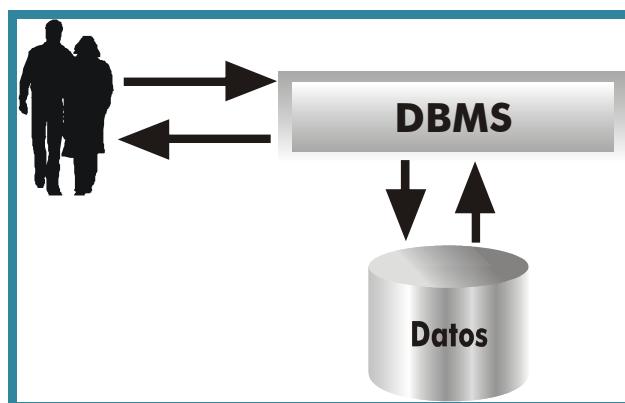


Ilustración 7, Esquema del funcionamiento y utilidad de un sistema gestor de bases de datos

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

(1.1.7) 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 de vista:

- **Nivel externo.** Esta es la visión de los datos que poseen los usuarios del Sistema de Información.
- **Nivel físico.** Esta es la forma en la que realmente están almacenados los datos.

Realmente la base de datos es la misma, pero se la puede observar desde estos dos puntos de vista. Al igual que una casa se la pueda observar pensando en los materiales concretos con los que se construye o bien pensando en ella con el plano en papel.

En todo sistema de información digital, los usuarios ven los datos desde las aplicaciones creadas por los programadores. A ese nivel se manejan formularios, informes en pantalla o en papel,...

Pero la realidad física de esos datos, tal cual se almacenan en los discos queda oculta a los usuarios. Esa forma de ver la base de datos está reservada a los administradores. Es el nivel físico el que permite ver la base de datos en función de cómo realmente se están almacenando en el ordenador, en qué carpeta, qué archivos se usan,...

En el caso de los Sistemas de Base de datos, se añade un tercer nivel, un tercer punto de vista, es el **nivel conceptual**. Ese nivel se sitúa entre el físico y el externo.

En cada nivel se manejan esquemas de la base de datos, al igual que al construir una casa, los distintos profesionales manejan distintos tipos de planos (eléctricos, de albañilería, de tuberías de agua,...). Con lo cual una base de datos requiere diseñar al menos tres esquemas (en realidad son más).

esquema físico

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

En este esquema se habla de archivos, directorios o carpetas, unidades de disco, servidores,...

esquema conceptual

Se trata de un esquema teórico de los datos en el que figuran organizados en estructuras reconocibles del mundo real y en el que también aparece la forma de relacionarse los datos. Este esquema es el paso que permite modelar un problema real a su forma correspondiente en el ordenador.

Este esquema es la base de datos de todos los demás. Como se verá más adelante, es el primer paso a realizar al crear una base de datos. En definitiva es el plano o modelo

general de la base de datos. El esquema conceptual lo realizan **diseñadores/as** o **analistas**.

esquema externo

En realidad son varios. Se trata de la visión de los datos que poseen los **usuarios y usuarias 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 dónde realmente se están almacenando los datos.

Los esquemas externos los realizan las **programadoras/es** según las indicaciones formales de los y las **analistas**.

Realmente cada aplicación produce un esquema externo diferente (aunque algunos pueden coincidir) o **vista 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. Se dice que esta función es la que permite definir las tres estructuras de la base de datos (relacionadas con los tres niveles de abstracción).

- Estructura interna
- Estructura conceptual
- Estructura externa

Realmente esta función trabaja con **metadatos**. Los metadatos es la información de la base de datos que realmente sirve para describir a los datos. Es decir, **Sánchez Rodríguez** y **Crespo** son datos; pero **Primer Apellido** es un metadato. También son datos decir que la base de datos contiene **Alumnos** o que el **dni** lo forman 9 caracteres de los cuales los 8 primeros son números y el noveno un carácter en mayúsculas.

La función de definición sirve pues para **crear, eliminar o modificar metadatos**. Para ello permite usar un **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 un **lenguaje de modificación de datos** o **DML**. Mediante ese lenguaje se puede:

- Añadir datos
- Eliminar datos

- Modificar datos
- Buscar datos

Actualmente se suele distinguir aparte la función de buscar datos en la base de datos (**función de consulta**). Para lo cual se proporciona un **lenguaje de consulta de datos** o **DQL**.

función de control

Mediante esta función los administradores poseen mecanismos para proteger los datos; de modo que se permite a cada usuario ver ciertos datos y otros no; o bien usar ciertos recursos concretos de la base de datos y prohibir otros.

Es decir simplemente permite controlar la seguridad de la base de datos. 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.** Organizadores y coordinadores del proyecto a desarrollar y máximos responsables del mismo. 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. Normalmente son además los **diseñadores de la base de datos** (especialmente de los esquemas interno y conceptual) y los directores de la programación de la misma.
- **Administradores/as de las bases de datos.** Encargados de crear el esquema interno de la base de datos, que incluye la planificación de copia de seguridad, gestión de usuarios y permisos y creación de los objetos de la base de datos.
- **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 como la creación de copias de seguridad por ejemplo o el arreglo de problemas de red por ejemplo).

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.

- **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

El proceso que realiza un SGBD está en realidad formado por varias capas que actúan como interfaces entre el usuario y los datos. Fue el propio organismo ANSI (en su modelo X3/SPARC que luego se comenta) la que introdujo una mejora de su modelo de bases de datos 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 modela el funcionamiento de la base de datos en una sucesión de capas cuya finalidad es ocultar y proteger la parte interna de las bases de datos.

Desde esta óptica para llegar a los datos hay que pasar una serie de capas que desde la parte más externa poco a poco van entrando más en la realidad física de la base de datos. Esa estructura se muestra en la Ilustración 8.



Ilustración 8, Modelo de referencia de las facilidades de usuario

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. Son, en definitiva, las páginas web y las aplicaciones con las que los usuarios manejan la base de datos. Permite abstraer la realidad de la base de datos a las usuarias y usuarios, mostrando la información de una forma más humana.

capa de acceso a datos

La capa de acceso a datos es la que permite comunicar a las aplicaciones de usuario con el diccionario de datos. Es un software (un driver o controlador en realidad) que se encarga traducir las peticiones del usuario para que lleguen de forma correcta a la base de datos y ésta pueda responder de forma adecuada.

diccionario de datos

Se trata del elemento que posee todos los metadatos. Gracias a esta capa las solicitudes de los clientes (que son conceptuales antes de llegar aquí) se traducen en instrucciones que hacen referencia al esquema interno de la base de datos.

núcleo

El núcleo de la base de datos es la encargada de traducir todas las instrucciones requeridas y prepararlas para su correcta interpretación por parte del sistema. Realiza la traducción física de las peticiones.

sistema operativo

Es una capa externa al software SGBD pero es la única capa que realmente accede a los datos en sí. En realidad los SGBD no acceden directamente al disco, sino que piden al Sistema Operativo que lo haga.

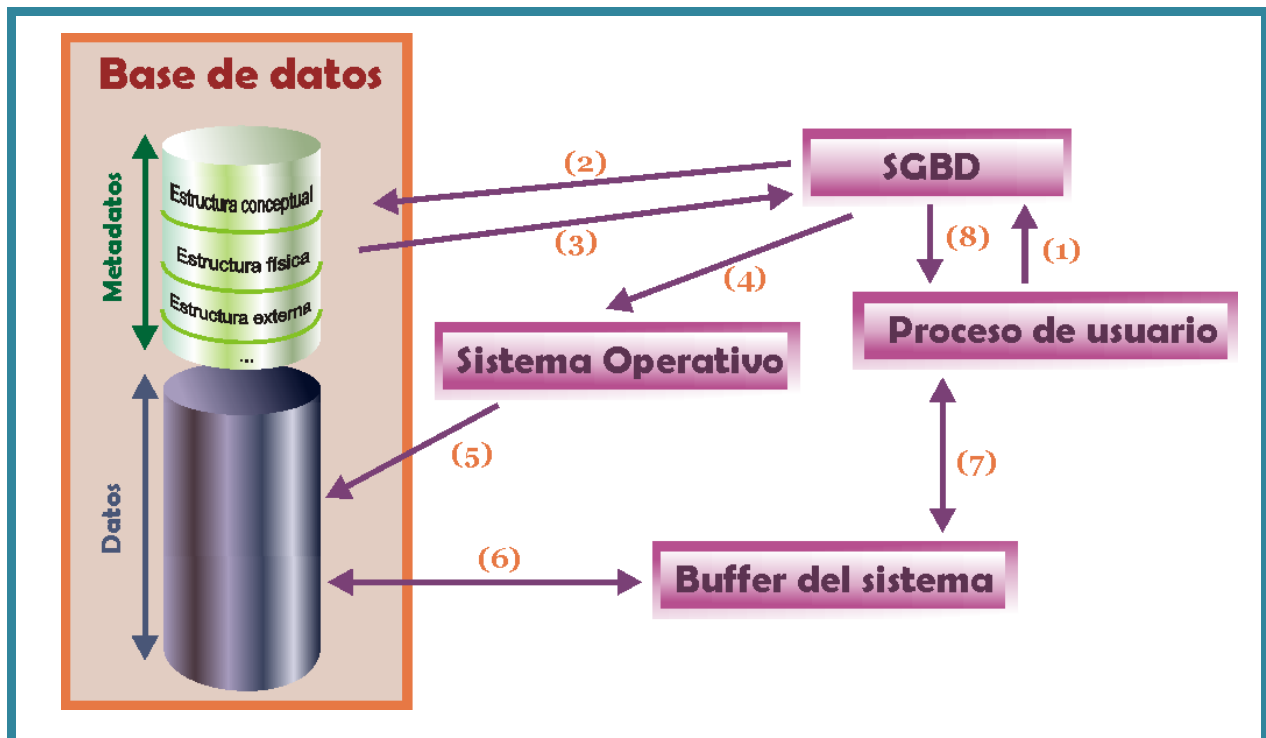


Ilustración 9, Esquema del funcionamiento de un SGBD

(1.2.4) funcionamiento del SGBD

La Ilustración 9 presenta el funcionamiento típico de un SGBD. En ella se reproduce la comunicación entre un proceso de usuario que desea acceder a los datos y el SGBD:

- (4) El proceso lanzado por el usuario llama al SGBD indicando la porción de la base de datos que se desea tratar
- (5) 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 (normalmente los metadatos se guardan una zona de memoria global y no en el disco)
- (6) El SGBD obtiene el esquema físico
- (7) El SGBD traduce la llamada a los métodos de acceso del Sistema Operativo que permiten acceder realmente a los datos requeridos
- (8) El Sistema Operativo accede a los datos tras traducir las órdenes dadas por el SGBD
- (9) Los datos pasan del disco a una memoria intermedia o *buffer*. En ese buffer se almacenarán los datos según se vayan recibiendo
- (10) Los datos pasan del buffer al área de trabajo del usuario (ATU) del proceso del usuario. Los pasos 6 y 7 se repiten hasta que se envíe toda la información al proceso de usuario.
- (11) En el caso de que haya errores en cualquier momento del proceso, 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** (éste sólo para las bases de datos orientadas a objetos). 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) ISO/JTC1/SC21/WG3

- **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. Es la parte, en definitiva de ISO, dedicada a la creación de estándares.
- **JTC 1** (*Joint Technical Committee*). Comité parte de IEC dedicado a la tecnología de la información (informática). En el campo de las bases de datos, el subcomité **SC 21** (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.

Entre los trabajos que realiza el grupo WG3 está la normalización de **SQL**, además de otras normas de estandarización.

(1.3.3) DBTG/CodasyI

CodasyI (*CO*nference on *DA*ta *SY*stem *LA*nguages) es el nombre de una conferencia iniciada en el año 1959 y que dio lugar a un organismo con la idea de conseguir un lenguaje estándar para la mayoría de máquinas informáticas. Participaron organismos privados y públicos del gobierno de Estados Unidos con la finalidad de definir estándares. Su primera tarea fue desarrollar el lenguaje **COBOL** y otros elementos del análisis, diseño y la programación de ordenadores.

La tarea real de estandarizar esos lenguajes se la cedieron al organismo ANSI, pero las ideas e inicios de muchas tecnologías se idearon en el consorcio CodasyI.

En 1967 se crea un **grupo de tareas para bases de datos** (*Data Base Task Group*) y este grupo definió el **modelo en red de bases de datos** y su integración con COBOL. A este modelo en red se le denomina **modelo CodasyI** o **modelo DBTG** y fue finalmente aceptado por la ANSI.

(1.3.4) ANSI/X3/SPARC

ANSI (*American National Standards 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).

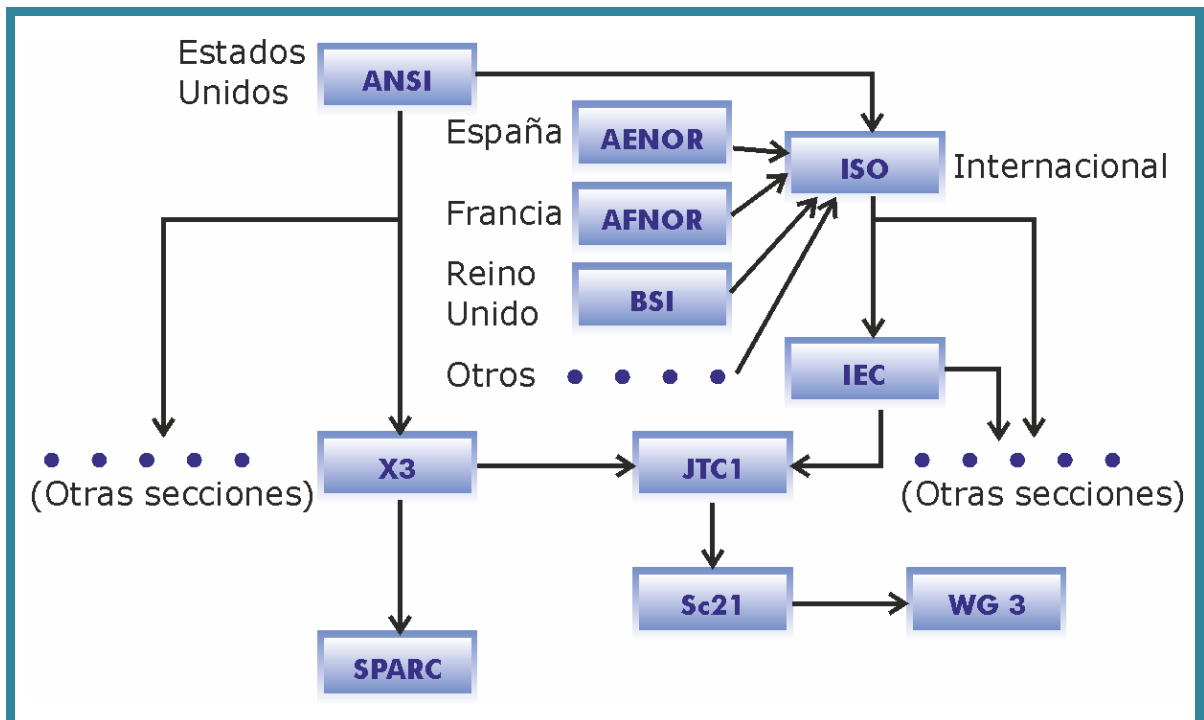


Ilustración 10, Relación entre los organismos de estandarización

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 organismo ANSI ha marcado la referencia para la construcción de SGBD. El modelo definido por el grupo de trabajo SPARC se basa en estudios anteriores en los que se definían tres niveles de abstracción necesarios para gestionar una 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**. Se entiende por modelo, el conjunto de 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.

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.

En definitiva el modelo ANSI es una propuesta teórica sobre cómo debe de funcionar un sistema gestor de bases de datos (sin duda, la propuesta más importante). Su idea es la siguiente:

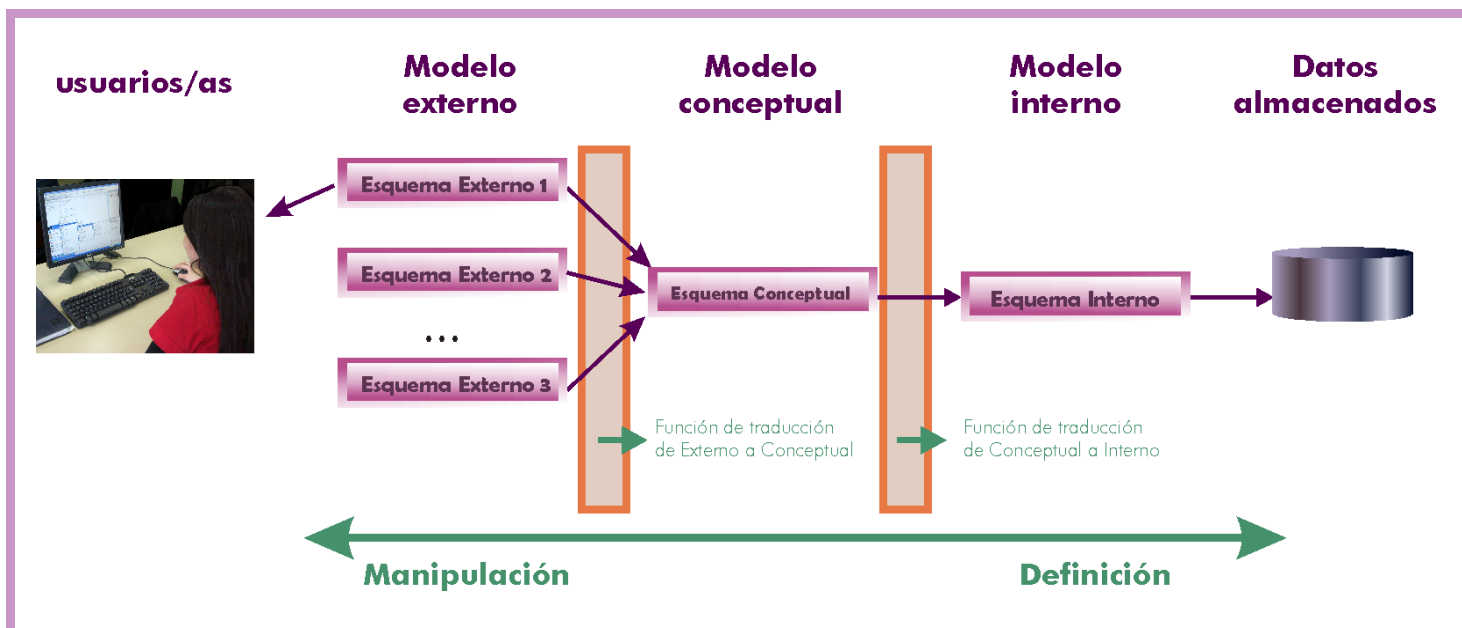


Ilustración 11, Niveles en el modelo ANSI

En la Ilustración 11, el paso de un esquema a otro se realiza utilizando una 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 12) está 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**).

LEYENDA

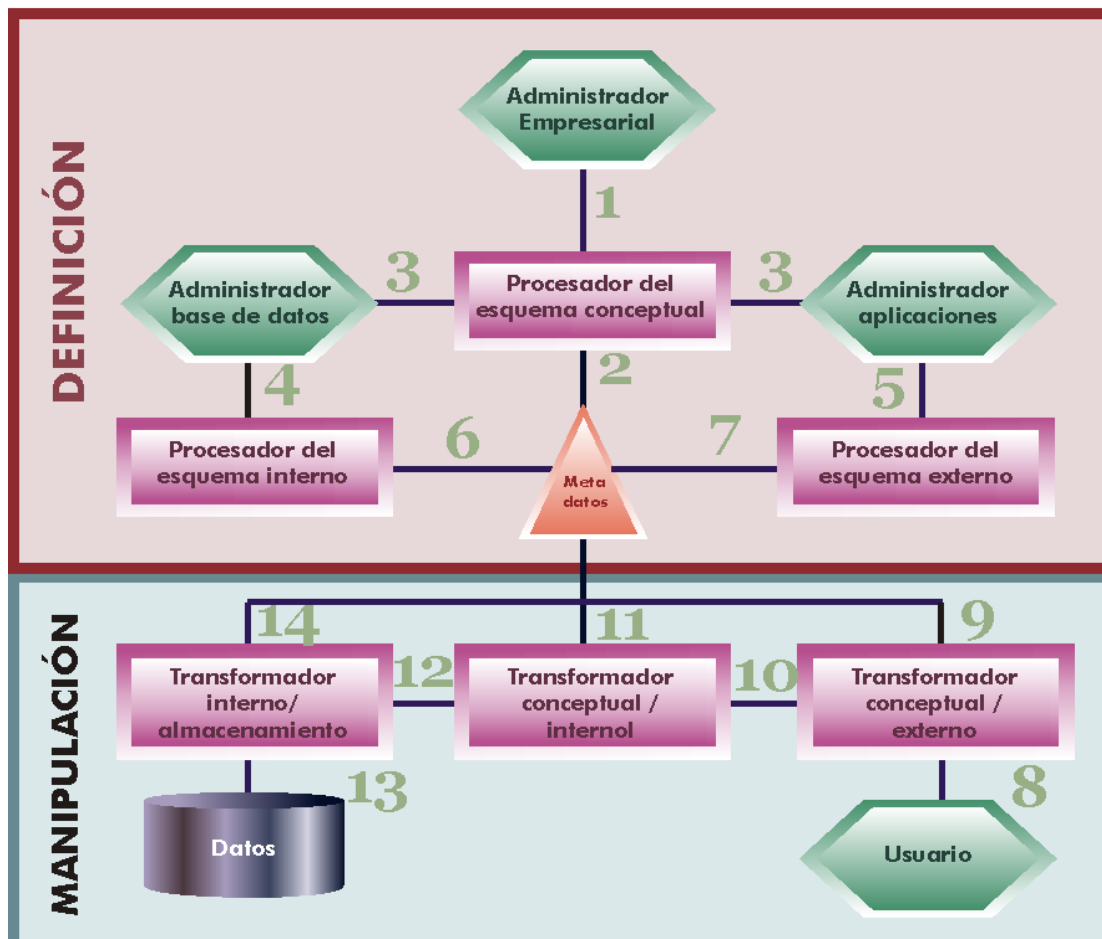


Ilustración 12, Arquitectura ANSI, explicación clásica del funcionamiento de un Sistema Gestor de Bases de Datos incluyendo los recursos humanos implicados

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) proceso de creación y manipulación de una base de datos actual

El modelo ANSI de bases de datos sigue estando vigente y por ello el ciclo de vida de una base de datos continúa atendiendo a las directrices marcadas por el modelo. No obstante sí han cambiado el nombre de los recursos humanos.

fase de creación:

- (4) El **analista** o **diseñador** (equivalente a un administrador de esquemas conceptuales del modelo ANSI) utiliza una **herramienta CASE** para crear el esquema conceptual
- (5) El **administrador** de la base de datos (**DBA**) recoge ese esquema y crea el esquema interno utilizando las herramientas de definición de datos del SGBD y herramientas CASE
- (6) Los **desarrolladores** también recogen el esquema conceptual y utilizan las aplicaciones necesarias para generar los esquemas externos que necesitan los usuarios mediante herramientas de creación de aplicaciones apropiadas y herramientas CASE

fase de manipulación:

- (4) El usuario realiza una consulta utilizando el esquema externo que las facilidades de usuario ponen a su disposición-
- (5) Las aplicaciones las traducen a su forma conceptual utilizando la información del diccionario de datos.
- (6) El esquema conceptual es traducido por la SGBD a su forma interna
- (7) EL SGBD se comunica con el Sistema Operativo para pedir que acceda al disco y archivos en los que están los datos pedidos (se sabe qué archivos concretos son gracias al diccionario de datos) y recoja los datos requeridos
- (8) El Sistema Operativo accede al almacenamiento físico correspondiente y devuelve los datos al SGBD
- (9) El SGBD transforma los datos internos en datos conceptuales y los entrega a la aplicación
- (10) La aplicación muestra los datos habiéndolos traducido en su forma externa. Así los ve el usuario

(1.3.7) formas de ejecución de un SGBD

Actualmente casi todos los Sistemas Gestores de Bases de Datos funcionan de manera semejante, en realidad hay tres posibilidades de funcionamiento:

- **SGBDs monocapa.** Es la más sencilla, pero la que tiene menos escalabilidad (posibilidad de crecer). El Sistema Gestor se instala en una máquina y los usuarios acceden directamente a esa máquina y ese Sistema Gestor. En estos sistemas no se accede de forma remota a la base de datos.
- **SGBDs bicapa.** 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. Hay dos posibilidades:

- **Estructura Cliente-Servidor.** La base de datos está en un solo servidor al que acceden los clientes (incluso simultáneamente).
- **Cliente multi-servidor.** En este caso los clientes acceden a un conjunto de servidores que distribuyen la base de datos. 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. Se usa cuando el número de clientes ha crecido mucho y un solo servidor no podría atender sus peticiones.

➤ **SGBD de tres o más capas.** Es una estructura de tipo cliente/servidor, pero en la que hay al menos una capa intermedia entre las dos. Esa capa se suele encargar de procesar las peticiones y enviarlas al SGBD con el que se comunica. Un ejemplo habitual es que la tercer capa sea un **servidor web** que evita que el cliente se conecte directamente al SGBD. Ese servidor web se encarga de traducir lo que el cliente realiza a una forma entendible por la base de datos. Esta forma de trabajar permite además que para usar una base de datos, baste un simple navegador al cliente.

Puede haber más capas con otros fines. Estas capas siempre están entre el cliente y el servidor.

(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.

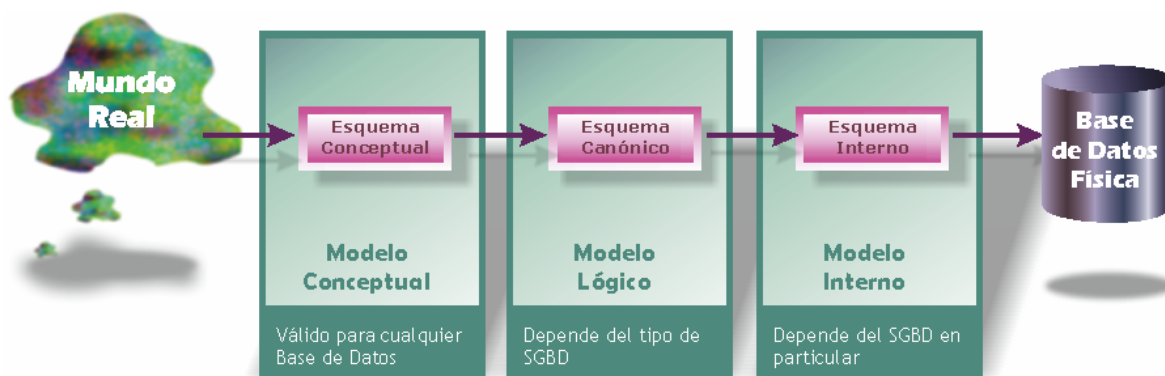


Ilustración 13, Modelos de datos utilizados en el desarrollo de una BD

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 esquemas señalados en la Ilustración 13.

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 está más cerca del modelo físico, el que utiliza internamente el ordenador
- El modelo conceptual es el más cercano al usuario, el lógico es el encargado de establecer el paso entre el modelo conceptual y el modelo físico del sistema.

Algunos ejemplos de modelos conceptuales son:

- **Modelo Entidad Relación**
- **Modelo RM/T**
- **Modelo UML**

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.

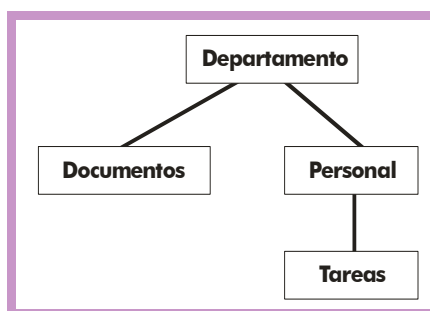


Ilustración 14, Ejemplo de esquema jerárquico

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 (Codasyl)

Es un modelo que ha tenido una gran aceptación (aunque apenas se utiliza actualmente). En especial se hizo popular la forma definida por Codasyl 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 Codasyl restringía un poco las relaciones posibles), pero hace muy complicado su manejo.

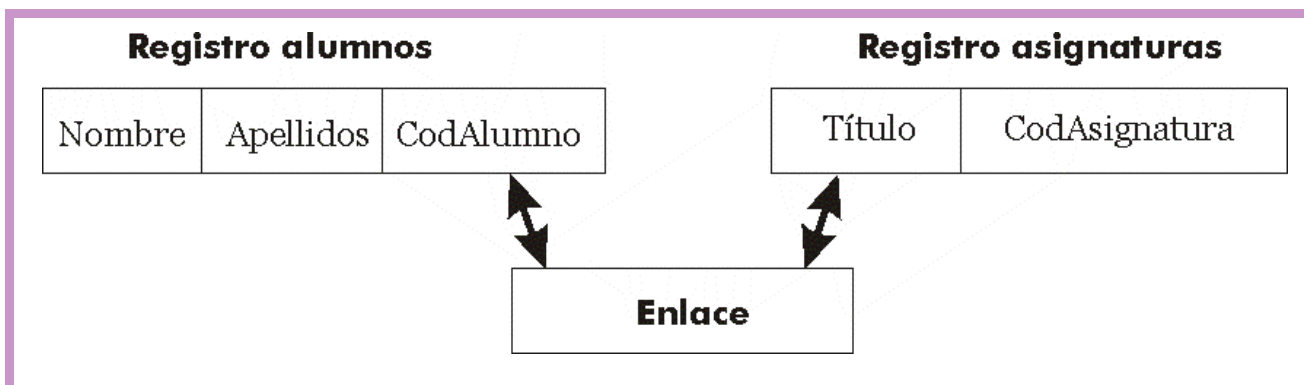


Ilustración 15, ejemplo de diagrama de estructura de datos Codasyl

(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 **OOP**) 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.

(1.4.7) bases de datos NoSQL

Bajo este nombre se agrupan las bases de datos (con arquitecturas muy diversas) pensadas para grabar los datos de manera veloz para así poder atender a miles y miles de peticiones. Es decir, es el modelo de las bases de datos que se utilizan en los grandes servicios de Internet (como **twitter**, **Facebook**, **Amazon**,...).

La idea es que los datos apenas necesitan validarse y relacionarse y lo importante es la disponibilidad de la propia base de datos. El nombre NoSQL, hace referencia a que este modelo de bases de datos rompe con el lenguaje SQL (el lenguaje de las bases de datos relacionales, las bases de datos dominantes hasta la actualidad) para poder manipular los datos con lenguajes de otro tipo.

(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 (es decir en la segunda unidad cambiaremos el concepto de relación).

(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,...

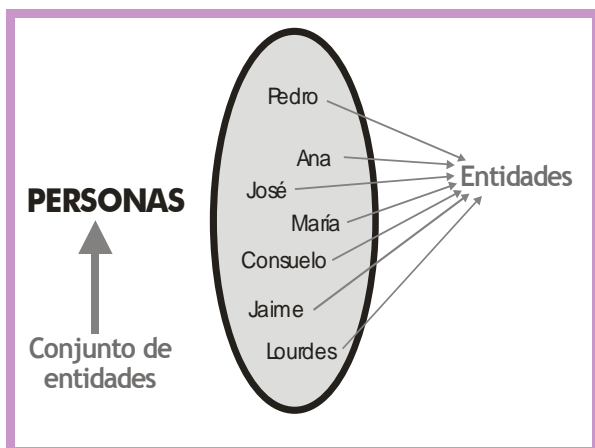


Ilustración 16, Ejemplos de entidad y conjunto de entidad

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 17, 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. Es decir e. Por ejemplo la entidad **tarea laboral** sólo podrá tener existencia si existe la entidad **trabajo**. Las entidades débiles se presentan de esta forma:

TAREAS LABORALES

Ilustración 18, 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:

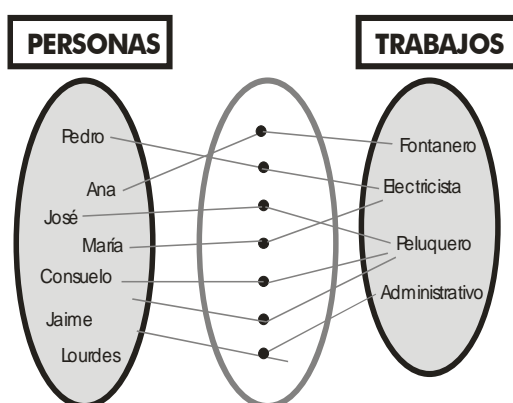


Ilustración 19, ejemplo de relación

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

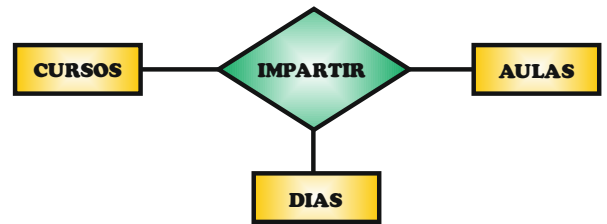
La representación gráfica de las entidades se realiza con un rombo al que se le unen líneas que se dirigen a las entidades, las relaciones tienen nombre (se suele usar un verbo). En el ejemplo anterior podría usarse como nombre de relación, trabajar:



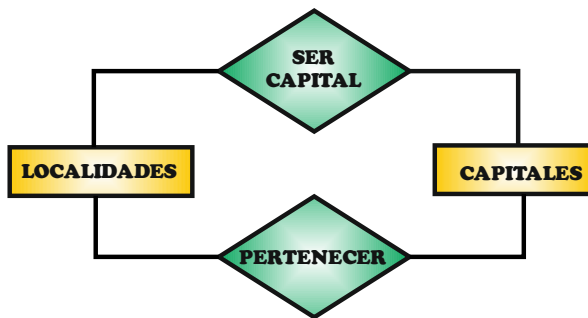
ejemplos de relaciones



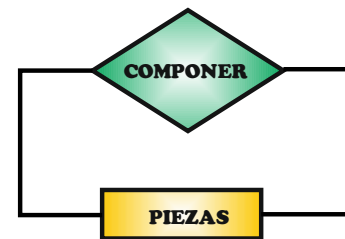
Relación binaria



Relación ternaria



Relación doble



Relación reflexiva

Ilustración 20, Tipos de relaciones

- **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). Normalmente la cardinalidad máxima es **1 ó 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 21, 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)

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:

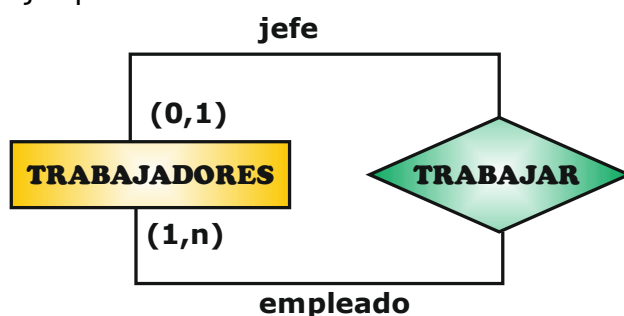


Ilustración 22, Ejemplo de rol. Un trabajador puede ser visto como jefe o como empleado según a qué lado de la relación esté

(1.5.4) atributos

Describen propiedades de las entidades y las relaciones. En este modelo se representan con elipses, dentro de las cuales se coloca el nombre del atributo. Esa elipse se une con una línea a las entidades. Ejemplo:

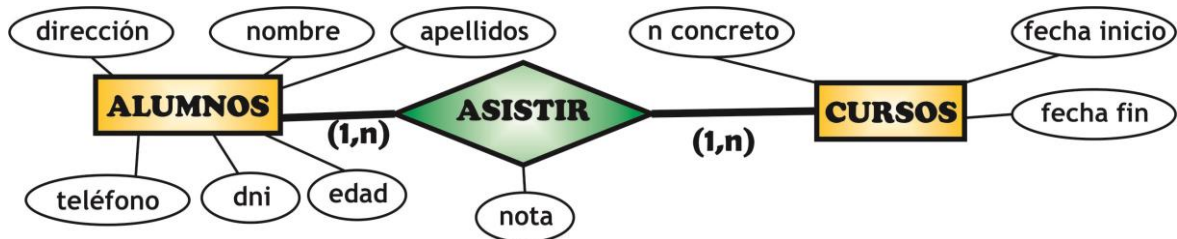
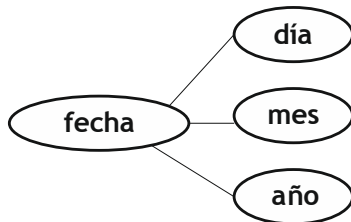


Ilustración 23, Atributos

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 atributos de una entidad cuyos valores son únicos en cada ejemplar de la entidad. Se marcan en el esquema subrayando el nombre del identificador.

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

- (4) Deben distinguir a cada ejemplar de la entidad o relación. Es decir no puede haber dos ejemplares con el mismo valor en el identificador.
- (5) Todos los ejemplares de una entidad deben tener el mismo identificador.
- (6) Un identificador puede estar formado por más de un atributo.
- (7) Puede haber varios identificadores *candidatos*, en ese caso hay que elegir el que tenga más importancia en nuestro sistema (el resto pasan a ser alternativos).

Todas las entidades deben de tener un identificador, en el caso de que una entidad no tenga identificador en sus atributos (puede ocurrir, pero hay que ser cauteloso, a veces se trata de entidades que están mal modeladas) entonces hay que añadir un atributo que haga de identificador. El nombre de este atributo artificial es la palabra **id** seguida del nombre de la entidad. Por ejemplo **id_personas**.

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 ISA o relaciones de herencia

Son relaciones que indican tipos de entidades, es decir tendremos entidades que son un (**is a**, en inglés) tipo de entidad.

Se utilizan para unificar entidades agrupándolas en una entidad más general (**generalización**) o bien para dividir una entidad general en entidades más específicas (**especificación**): aunque hoy en día a todas ellas se las suele llamar generalización e incluso **relaciones de herencia**.

Se habla de **superentidad** refiriéndonos a la entidad general sobre las que derivan las otras (que se llaman **subentidades**). En la superentidad se indican los atributos comunes a todas las subentidades, se sobreentiende que las subentidades también tienen esos atributos, pero no se indica en el diagrama.

Normalmente cuando tenemos una especialización las subentidades comparten clave con la **superentidad** (además de los atributos comunes); esto es muy matizable y de hecho hoy en día ningún diseñador intenta distinguir entre si tenemos una especialización o una generalización, porque al final ambas implican los mismos resultados. Sí interesan otras cuestiones en las relaciones ISA.

En la práctica se manejan casi igual ambas; la forma clásica de representar una ISA es:

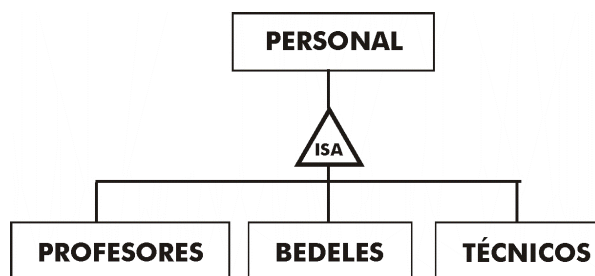


Ilustración 24, Relación ISA. ¿Generalización o especialización? En la práctica no importa

En general se suelen indicar las cardinalidades en las relaciones ISA, pero se suele sobrentender (cuando no se indican explícitamente) que hay un (0,1) encima de cada subentidad (que significa que cada ejemplar de la subentidad solo puede relacionarse como mucho con uno de la subentidad e incluso con ninguno; un empleado de *personal* podría ser o no ser un *profesor*).

Pero se puede perfectamente indicar la cardinalidad (se usa ya la notación de ISA con triángulo hacia abajo que es la más popular en España actualmente):

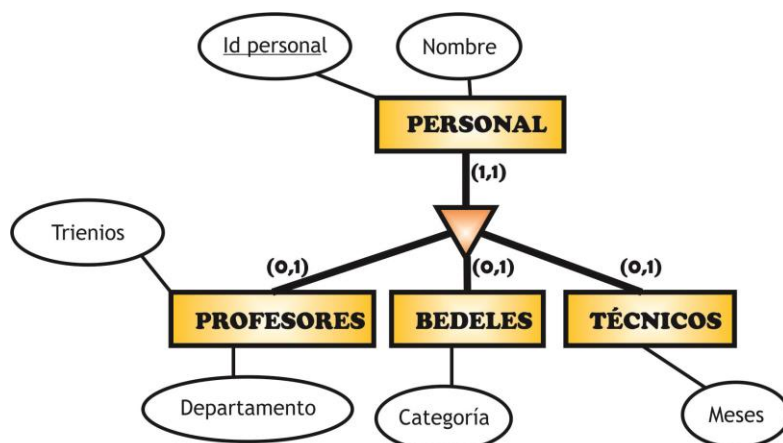


Ilustración 25, Ejemplo de relación ISA

De hecho cualquier cardinalidad sería válida (aunque lo normal es que solo aparezcan ceros y unos).

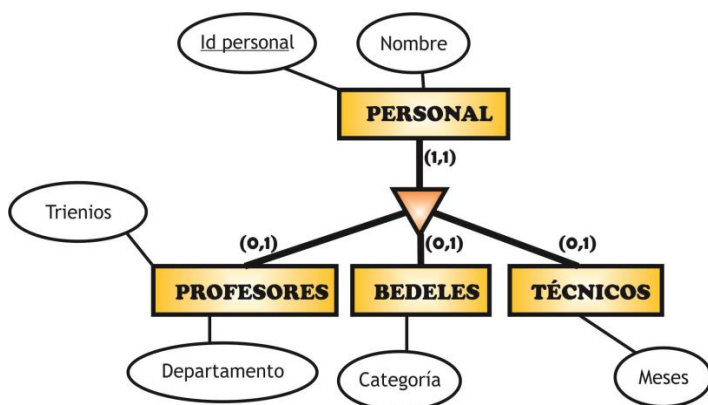


Ilustración 26, En esta relación de herencia, la clave de la superentidad es clave de las subentidades.

En la relación ISA anterior, 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)

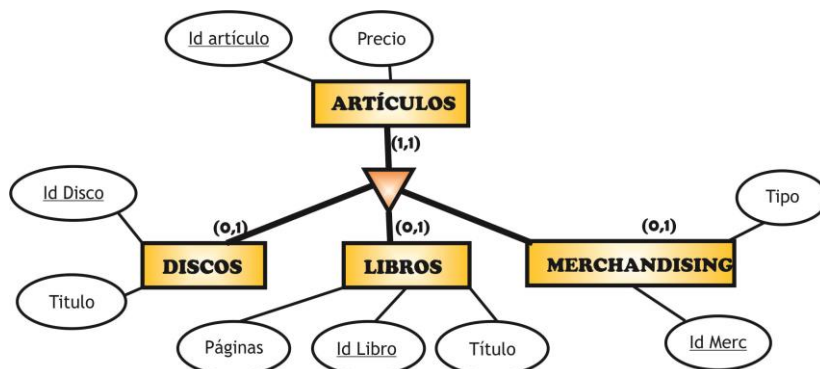


Ilustración 27, La clave de la superentidad no es clave de las subentidades.

En la ilustración anterior se utiliza una clave distinta para cada subentidad (es decir, *discos*, *libros* y *merchandising* tienen clave propia), no la heredan.

Es posible tener esta situación incluso (aunque no es muy habitual):

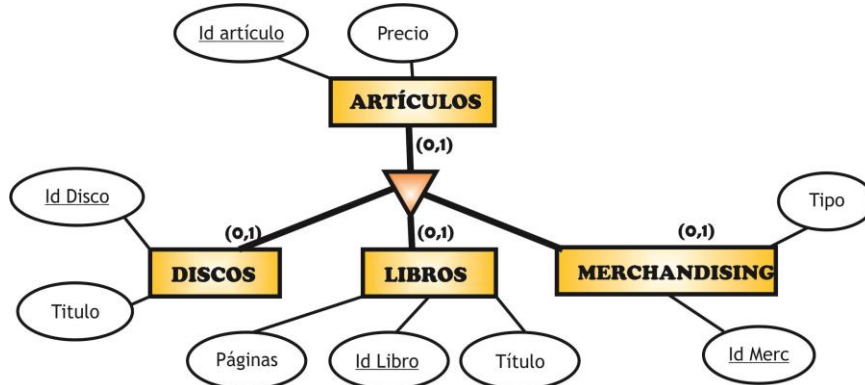


Ilustración 28, Discos, Libros y Merchandising no se relacionan obligatoriamente con la superentidad Artículos

Es un caso en el que no hay relación obligatoria con la superentidad; es decir, un *disco* podría no ser un *artículo* (porque a lo mejor quiero meter discos en mi base de datos que no vendo). No es muy habitual utilizar de esta forma relaciones ISA, pero desde luego es posible.

En muchas ocasiones no se indican las cardinalidades y se sobrentiende que la superentidad tiene un 1,1 y las subentidades 0,1.

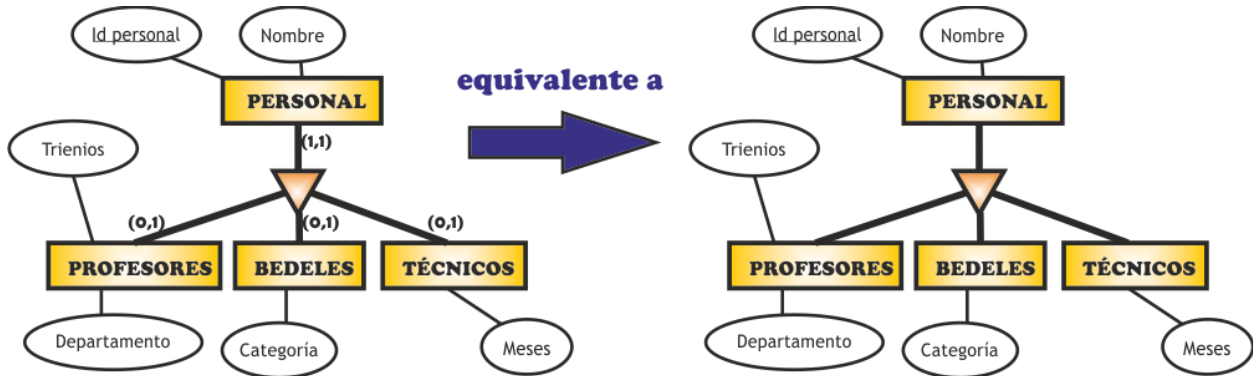


Ilustración 29, A la derecha el mismo gráfico Entidad Relación sin indicar las cardinalidades. Se sobrentiende que son las de la izquierda

exclusividad

En las relaciones ISA (y también en otros tipos de relaciones) se puede indicar el hecho de que cada ejemplar sólo puede 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:

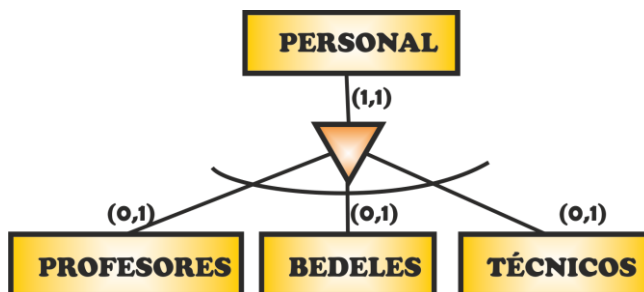


Ilustración 30, Relación ISA con obligatoriedad

En el ejemplo, el personal sólo puede ser o bedel, o profesor o técnico; una y sólo una de las tres cosas (es por cierto la forma más habitual de relación ISA).

tipos de relaciones ISA

Realmente lo que hay que matizar bien en las relaciones ISA es la forma de relacionarse la superentidad con la subentidad. Eso se matiza en base a dos conceptos:

- **Obligatoriedad.** Indica si los ejemplares obligatoriamente se relacionan con ejemplares de las subentidades. Es decir si hay personal que no es profesor ni bedel ni técnico o si fijo es alguna de esas tres profesiones. Hay dos posibilidades:
 - **Relaciones de jerarquía parcial.** Indican que hay ejemplares de la superentidad que no se relacionan con ninguna subentidad (hay personal que no es ni profesor, no bedel ni técnico). Se indican con cardinalidad mínima de cero en la superentidad.
 - **Relaciones de jerarquía total.** Indican que todos los ejemplares de la superentidad se relacionan con alguna subentidad (no hay personal que no sea ni profesor, ni bedel ni técnico). Se indican con cardinalidad mínima de uno en la superentidad.
- **Número de relaciones.** En este caso se mide con cuántas subentidades se relaciona la subentidad; es decir, si hay personal que pueda ser profesor y bedel a la vez o si solo puede ser una cosa. Posibilidades:
 - **Relaciones de jerarquía solapada.** Indican que un ejemplar de la superentidad puede relacionarse con más de una subentidad (el personal puede ser profesor y bedel). Ocurren cuando no hay dibujado un arco de exclusividad.
 - **Relaciones de jerarquía exclusiva.** Indican que un ejemplar de la superentidad sólo puede relacionarse con una subentidad (el personal no puede ser profesor y bedel). Ocurren cuando hay dibujado un arco de exclusividad.

La forma gráfica más aceptada actualmente para representar este tipo de relaciones, sería la siguiente:

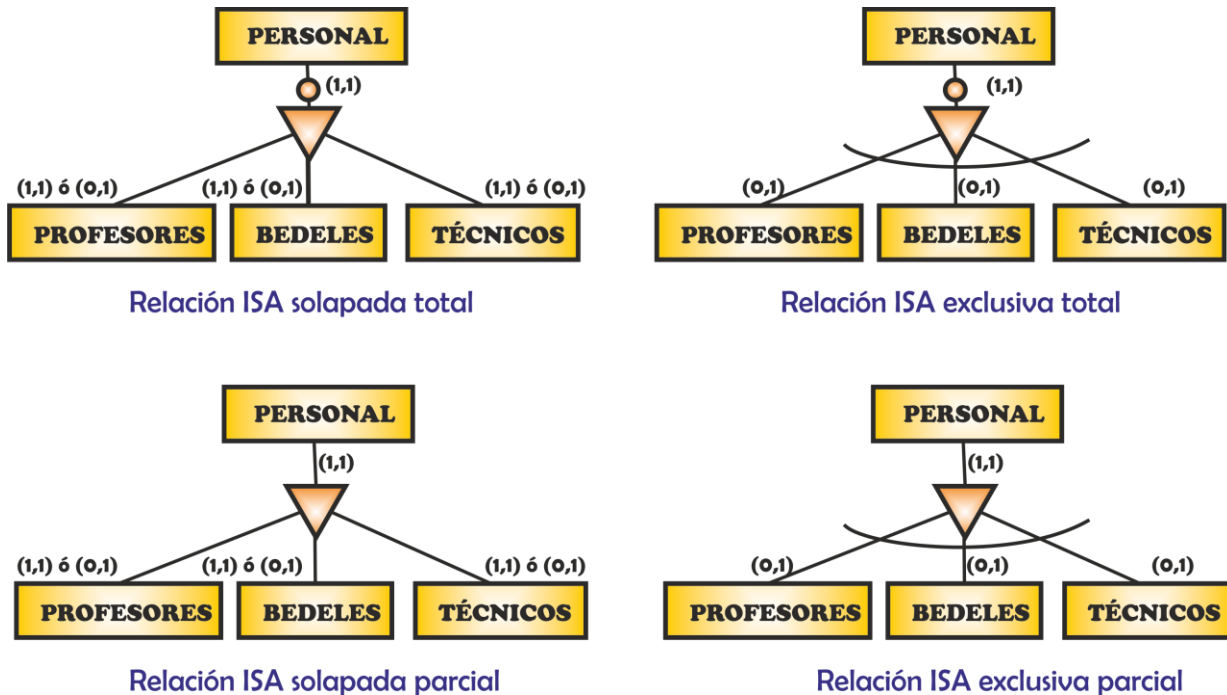


Ilustración 31, Tipos de relaciones ISA

Todos los posibles ejemplos de relaciones ISA atendiendo a la cardinalidad son los expuestos en la Ilustración 31

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:

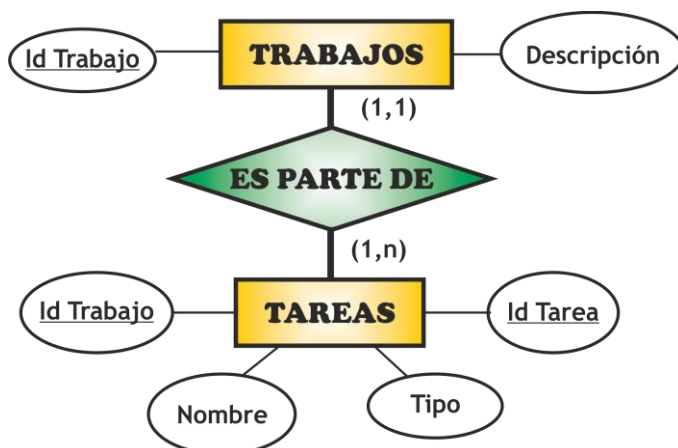


Ilustración 32, Relación candidata a entidad débil

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:

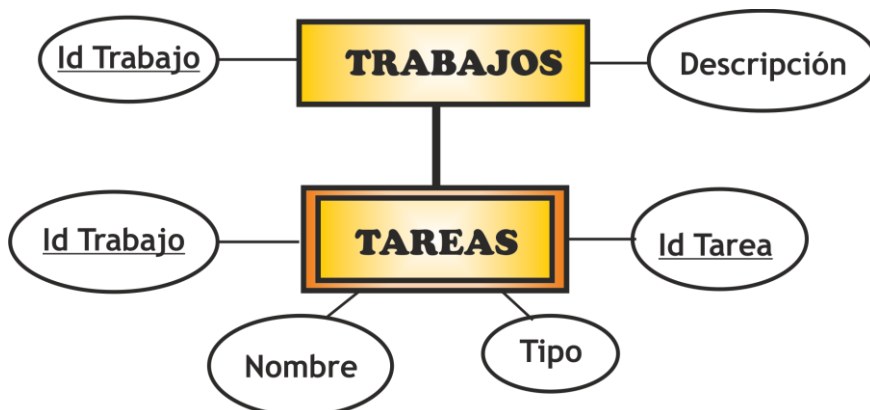


Ilustración 33, Entidad débil relacionada con su entidad fuerte

No hace falta dibujar el rombo de la relación ni la cardinalidad, se sobreentiende el tipo y cardinalidad (1 a n) que posee. No siempre el identificador de la entidad débil incluye el identificador de la entidad fuerte; cuando ocurre se habla de una **dependencia de identificación**, pero no todas las entidades débiles las poseen.

(2)

bases de datos relacionales

(2.1) el modelo relacional

(2.1.1) introducción

Edgar Frank Codd 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. Lo es porque en él se definieron las bases del llamado **Modelo Relacional de Bases de Datos**. Anteriormente el único modelo teórico estandarizado era el **Codasyl** que se utilizó masivamente en los años 70 como paradigma del modelo en red de bases de datos.

Codd se apoya en los trabajos de los matemáticos **Cantor** y **Childs** (cuya teoría de conjuntos es la verdadera base del modelo relacional). Según Codd los datos se agrupan en **relaciones** (actualmente llamadas **tablas**) que es un concepto que se refiere a la estructura que aglutina datos referidos a una misma entidad de forma independiente respecto a su almacenamiento físico.

Lo que Codd intentaba fundamentalmente es evitar que las usuarias y usuarios de la base de datos tuvieran que verse obligadas a aprender los entresijos internos del sistema. Pretendía que los usuarios/as trabajaran de forma sencilla e independiente del funcionamiento físico de la base de datos en sí. Fue un enfoque revolucionario.

Aunque trabajaba para **IBM**, esta empresa no recibió de buen grado sus teorías (de hecho continuó trabajando en su modelo en red **IMS**). 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 la forma de almacenar los datos cambia, los usuarios no tienen siquiera porque percibirlo y seguirán trabajando de la misma forma con la base de datos. Esto permite que los usuarios y usuarias se concentren en qué quieren consultar en la base de datos y no en cómo está realizada la misma.
- **Independencia lógica.** Las aplicaciones que utilizan la base de datos no deben ser modificadas porque se modifiquen elementos de la base de datos. Es decir, añadir, borrar y suprimir datos, no influye en las vistas de los usuarios. De una manera más precisa, gracias a esta independencia el esquema externo de la base de datos es realmente independiente del modelo lógico.
- **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.** Facilidad de manejo (algo cuestionable, pero ciertamente verdadero si comparamos con los sistemas gestores de bases de datos anteriores a este modelo).

(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 de base de datos relacional. Son el System R de IBM. En ese sistema se desarrolla Sequel que con el tiempo cambiará su nombre a SQL.
1974	La Universidad de Berkeley desarrolla Ingres , SGBD relacional basado en cálculo relacional. Utilizaba el lenguaje Quel desarrollado en las universidades y muy popular en la época en ámbitos académicos.
1978	Aparece el lenguaje QBE (Query By Example) lenguaje de acceso relacional a los archivos VSAM de IBM
1979	Aparece Oracle , el primer SGBD comercial relacional (ganando en unas semanas al System/38 de IBM). Implementa SQL y se convertirá en el sistema gestor de bases de datos relacionales líder del mercado. Codd revisa su modelo relacional y lanza el modelo RM/T como un intento de subsanar sus deficiencias.
1981	Aparece Informix como SGBD relacional para Unix
1983	Aparece DB2 , el sistema gestor de bases de datos relacionales de IBM

Año	Hecho
1984	Aparece la base de datos Sybase que llegó a ser la segunda más popular (tras Oracle)
1986	ANSI normaliza el SQL (SQL/ANSI). SQL es ya de hecho el lenguaje principal de gestión de bases de datos relacionales.
1987	ISO también normaliza SQL. Es el SQL ISO(9075)
1988	La versión 6 de Oracle incorpora el lenguaje procedimental PL/SQL
1989	ISO revisa el estándar y publica el estándar SQL Addendum . Microsoft y Sybase desarrollan SQL Server para el sistema operativo OS/2 de Microsoft e IBM . Durante años Sybase y SQL Server fueron el mismo producto.
1990	Versión dos del modelo relacional (RM/V2) realizada por Codd. Propuesta de Michael 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. Aparece el modelo objeto/relacional. Aparece MySQL una base de datos relacional de código abierto con licencia GNU que se hace muy popular entre los desarrolladores de páginas web.
1996	ANSI normaliza el lenguaje procedimental basado en SQL y lo llaman SQL/PSM . Permite técnicas propias de los lenguajes de programación estructurada. Aparece el SGBD abierto PostgreSQL como remodelación de la antigua Ingres, utilizando de forma nativa el lenguaje SQL (en lugar de Quel).
1999	ISO publica un nuevo estándar que incluye características más avanzadas. Se llama SQL 99 (también se le conoce como SQL 200)
2003	ISO publica el estándar SQL 2003 . En él se añade SQL/PSM al estándar.
2006	Estándar ISO. SQL 2006
2008	Estándar ISO. SQL 2008
2011	Estándar ISO. SQL 2011

(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.

No hay que confundir la idea de relación según el modelo de **Codd**, con lo que significa una relación en el modelo Entidad/Relación de **Chen**. No tienen nada que ver

Las relaciones constan de:

- **Atributos**. Referido a cada propiedad de los datos que se almacenan 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 relación. Se corresponde con la idea clásica 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, facilitando la definición de los mismos.

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	=	<u>tabla</u>	=	fichero
tupla	=	<u>fila</u>	=	registro
atributo	=	<u>columna</u>	=	campo
<u>grado</u>	=	nº de columnas	=	nº de campos
<u>cardinalidad</u>	=	nº de filas	=	nº de registros

Se han subrayado en la tabla los términos que se usan más.

(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:

$$\{(A_i:D_i)\}_{i=1}^n \text{ donde } n \text{ es el grado.}$$

- **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:

Clientes		
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 (aunque 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:
 - **Bases.** Independientes, se crean indicando su estructura y sus ejemplares. Contienen tanto datos como metadatos.
 - **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 (con lo que tienen el riesgo de que muestren algunos datos obsoletos).
- **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. Toda tabla en el modelo relacional debe tener al menos una clave candidata (puede incluso haber más)

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 tabla 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 (una variable, un objeto,..) no tiene ningún contenido. Por ejemplo cuando un puntero en lenguaje C señala a **null** se dice que no está señalando a nadie. Al programar en esos lenguajes se trata de un valor que no permite utilizarse en operaciones aritméticas o lógicas.

Las bases de datos relacionales permiten más posibilidades para el valor nulo (**null**), aunque su significado no cambia: valor vacío. No obstante en las bases de datos se utiliza para diversos fines.

En claves secundarias indican que el registro actual no está relacionado con ninguno. En otros atributos indica que la tupla en cuestión carece de dicho atributo: por ejemplo en una tabla de **personas** un valor nulo en el atributo **teléfono** indicaría que dicha persona no tiene teléfono.

Es importante indicar que el texto vacío ' ', no significa lo mismo en un texto que el nulo; como tampoco el valor cero significa nulo.

Puesto que ese valor se utiliza continuamente, resulta imprescindible saber cómo 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**, o usando la aritmética clásica en valores lógicos, el 1 es verdadero, el 0 falso y el 0,5 nulo).

El uso de operadores lógicos con el nulo da lugar a que:

- **verdadero Y (AND) nulo** da como resultado, nulo (siguiendo la aritmética planteada antes: $1 \cdot 0,5 = 0,5$)
- **falso Y (AND) nulo** da como resultado, falso ($0 \cdot 0,5 = 0$)
- **verdadero O (OR) nulo** da como resultado, verdadero ($1 + 0,5 > 1$)
- **falso O nulo** da como resultado nulo ($0 + 0,5 = 0,5$)
- **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 condiciones de obligado cumplimiento por las tuplas 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 la tupla y dominio en el que está inscrito**

(2.3.2) semánticas

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

clave principal (primary key)

También llamada clave primaria. Marca uno o más atributos como identificadores de la tabla. De esa forma en esos atributos las filas de la tabla no podrán repetir valores ni tampoco dejarlos vacíos.

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 quede vacío (es decir impide que pueda contener el valor nulo, **null**).

integridad referencial (**foreign key**)

Sirve para indicar una clave externa (también llamada secundaria y foránea) sobre uno o más atributos. Los atributos marcados de esta forma sólo podrán contener valores que estén relacionados con la clave principal de la tabla que relacionan (llamada tabla principal). Dichos atributos sí podrán contener valores nulos.

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.

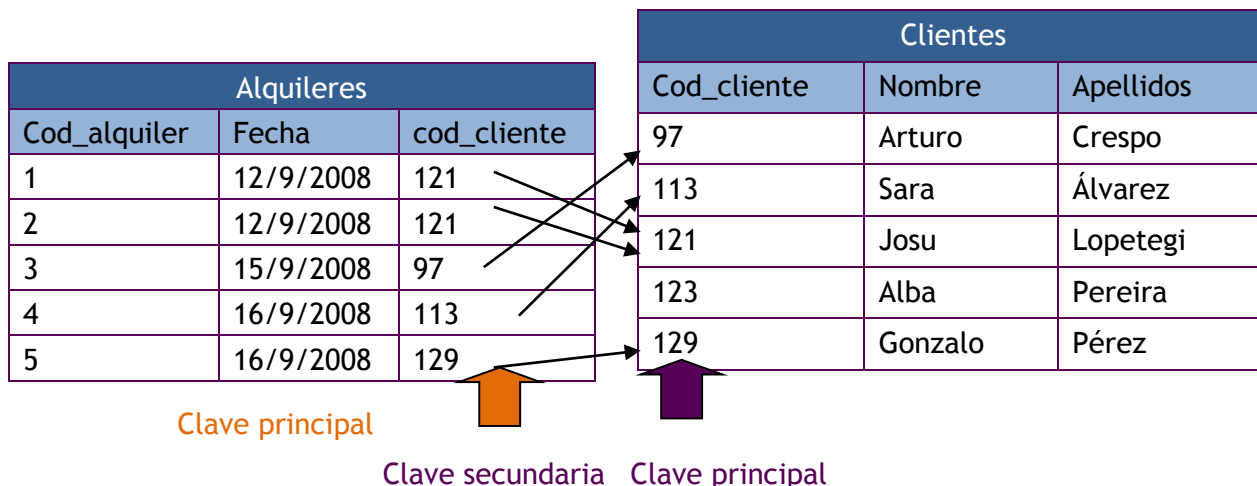


Ilustración 34, Ejemplo de clave secundaria

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 en la tabla principal.

Para solventar esta situación se puede hacer uso de estas opciones:

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

regla de validación (**check**)

Condición lógica que debe de cumplir un dato concreto para darlo por válido. Por ejemplo restringir el campo sueldo para que siempre sea mayor de 1000, sería una regla de validación. También por ejemplo que la fecha de inicio sea mayor que la fecha final.

disparadores o triggers

Se trata de pequeños programas grabados en la base de datos que se ejecutan automáticamente cuando se cumple una determinada condición. Sirven para realizar una serie de acciones cuando ocurre un determinado evento (cuando se añade una tupla, cuando se borra un dato, cuando un usuario abre una conexión...)

Los triggers permiten realizar restricciones muy potentes; pero son las más difíciles de crear.

(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 SGBD debe encargarse de que las vistas muestren la última información. No son válidas vistas que muestren datos que no están al día.
- (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, aun cuando se modifique el almacenamiento. La forma de acceder a los datos no varía porque el esquema físico de la base de datos, cambie.
- (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 funcionen igualmente en una base de datos distribuida que en una que no lo es.
- (12) **No subversión.** Si el SGBD posee un lenguaje procedimental que permita crear bucles de recorrido fila a fila, éste no puede utilizarse para incumplir o evitar las reglas relacionales anteriores. Especialmente la regla 7 no puede ser incumplida por ningún lenguaje del SGBD.

(2.5) paso de entidad/relación de Chen al modelo relacional

(2.5.1) 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:

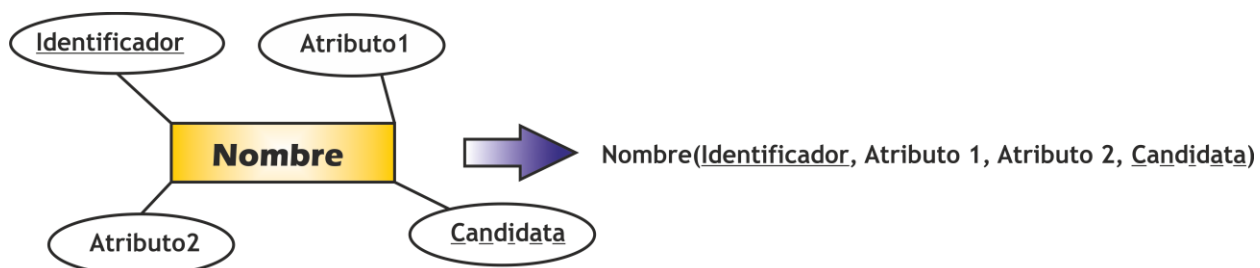


Ilustración 35, Transformación de una entidad fuerte al esquema relacional

(2.5.2) transformación de relaciones

La idea inicial es transformar cada relación del modelo conceptual en una tabla en el modelo relacional. Pero hay casos en los que esta regla tiene matices y no se cumple.

relaciones varios a varios

En las relaciones varios a varios (n a n en la cardinalidad mayor, la cardinalidad menor no cuenta para esta situación), 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.

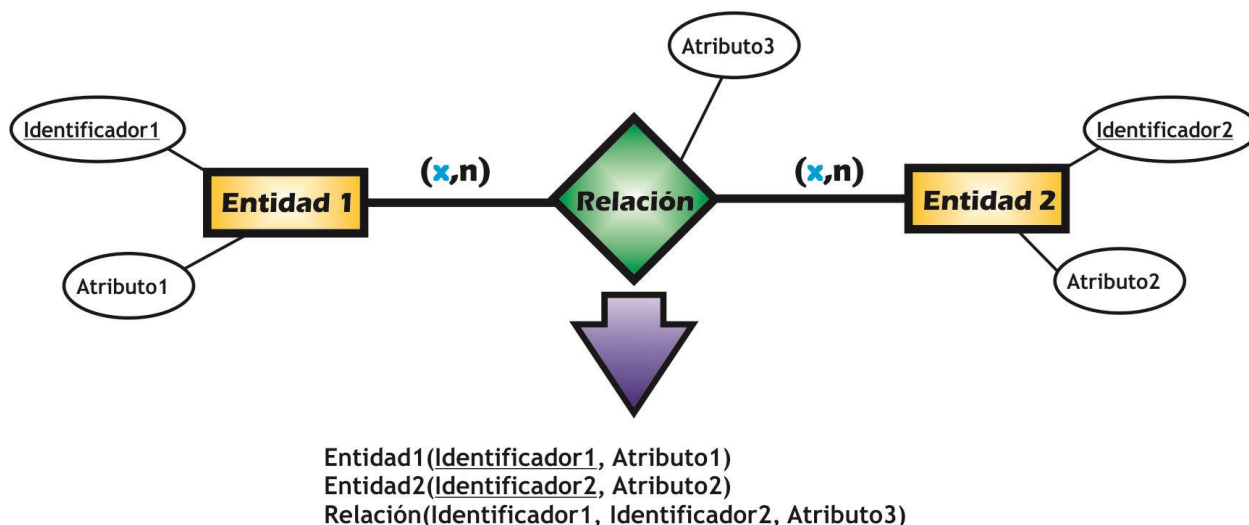


Ilustración 36, Transformación de una relación n a n

En el dibujo anterior los símbolos x referidos a la cardinalidad mínima, indican que ésta no influye en el paso al modelo relacional.

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:

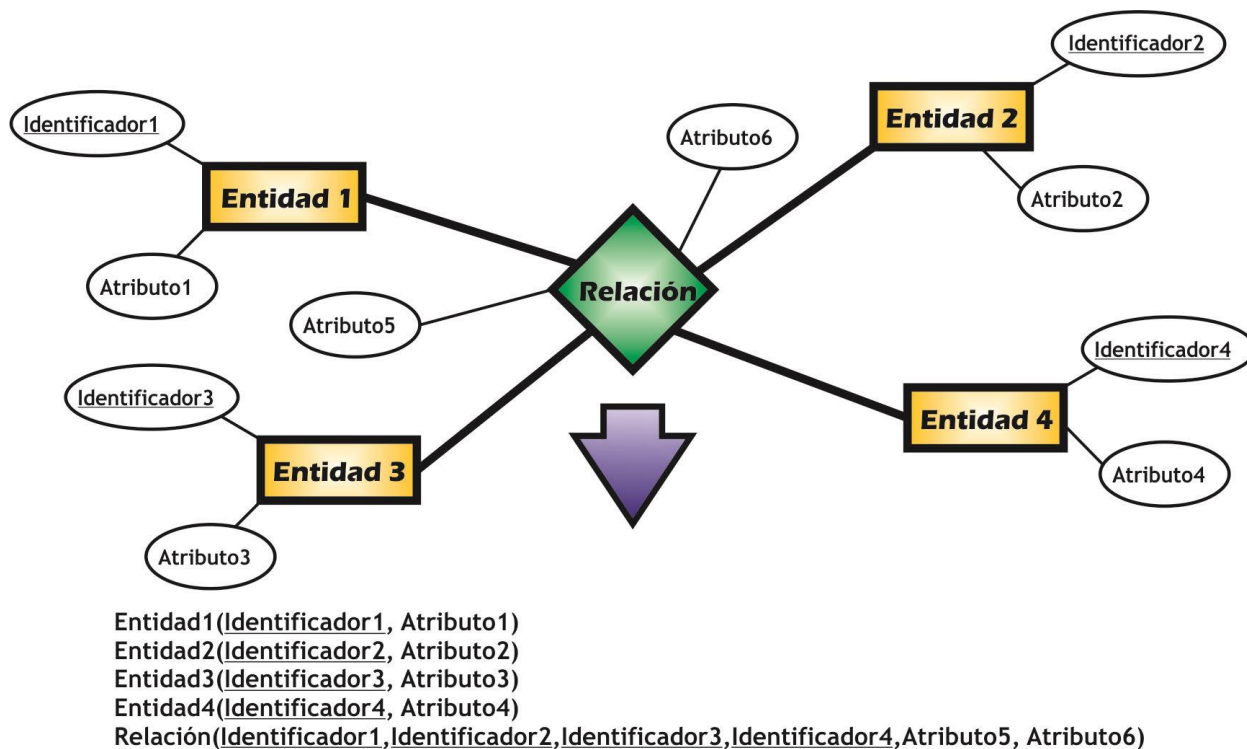
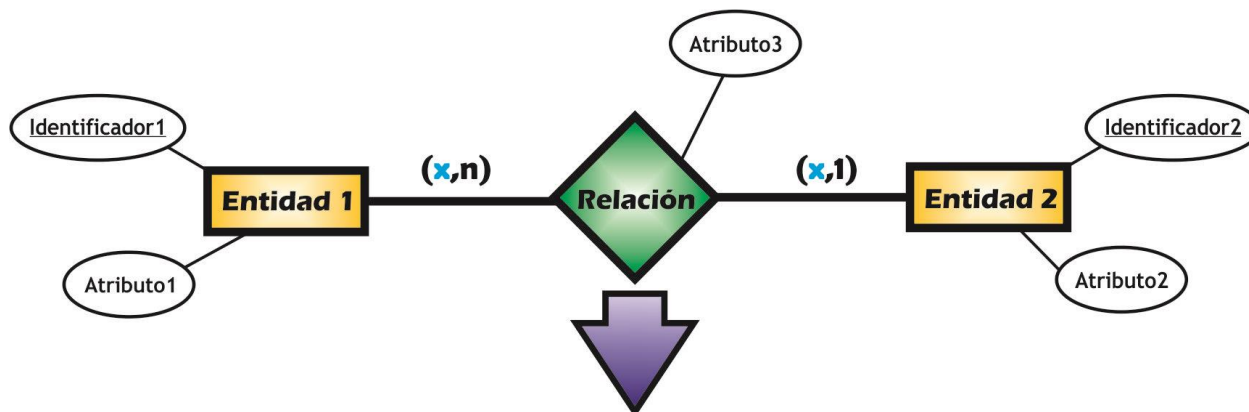


Ilustración 37, Transformación en el modelo relacional de una entidad n -aria

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).



Entidad1(Identificador1,Atributo1, Identificador2, Atributo2)
Entidad2(Identificador2,Atributo3)

Ilustración 38, Transformación de una relación uno a varios

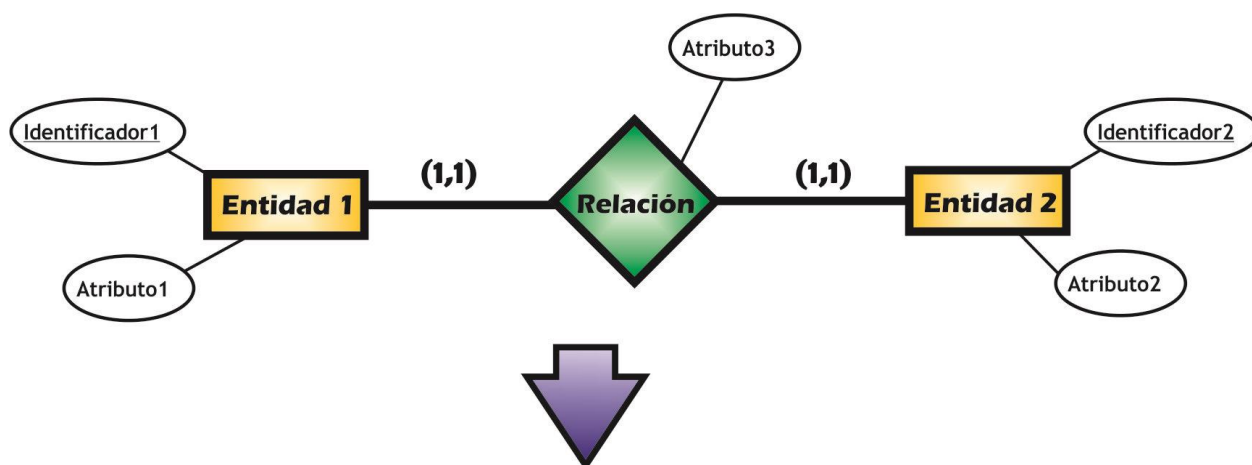
Así en el dibujo, el **Identificador2** en la tabla **Entidad1** pasa a ser una clave secundaria. En el caso de que el número mínimo de la relación sea de **cero** (puede haber ejemplares de la entidad uno sin relacionar), se deberá permitir valores nulos en la clave secundaria (en el ejemplo sería el **Identificador2** en la **Entidad1**). En otro caso no se podrán permitir (ya que siempre habrá un valor relacionado).

relaciones uno a uno

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

- Colocar la clave de una de las entidades como clave externa de la otra tabla (da igual cuál), 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 marcada como **clave alternativa**. El nombre de la tabla sería el de la entidad más importante desde el punto de vista conceptual.

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



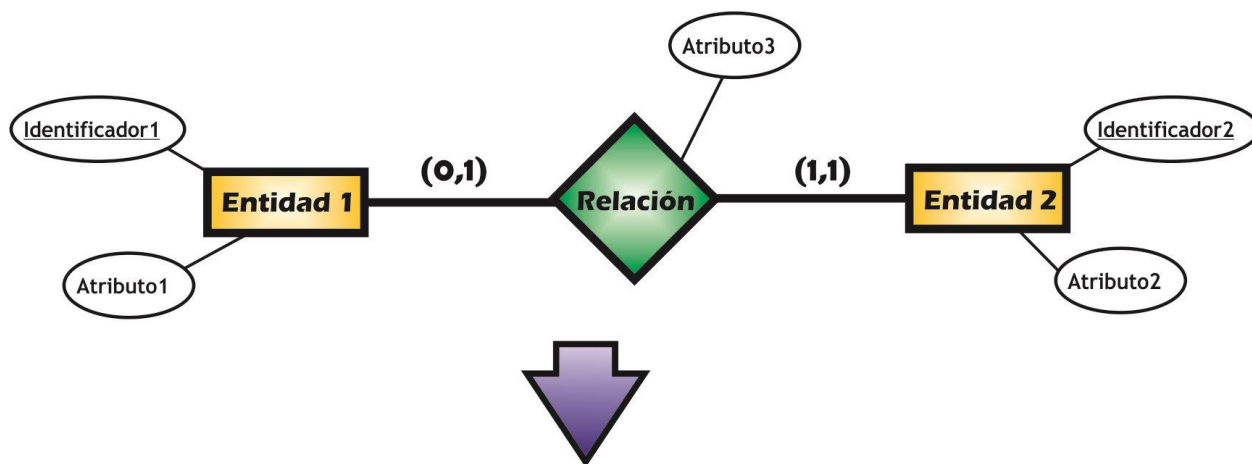
Solución 1: Entidad1(Identificador1,Atributo1, Identificador2, Atributo2,Atributo3)
 Solución 2: Entidad2(Identificador2, Atributo2,Atributo3,Identificador1,Atributo1)

Ilustración 39, Las dos soluciones válidas a la cardinalidad 1 a

relaciones cero a uno

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 la solución difiere respecto a la anterior solución. No conviene generar una única tabla ya que habría numerosos valores nulos en la tabla (debido a que hay ejemplares que no se relacionan en las dos tablas).

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):



Entidad1(Identificador1,Atributo1, Identificador2, Atributo3)
 Entidad2(Identificador2,Atributo2)

Ilustración 40, Solución a la relación 0 a 1

relaciones cero a cero

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, marcada en el dibujo con un asterisco).

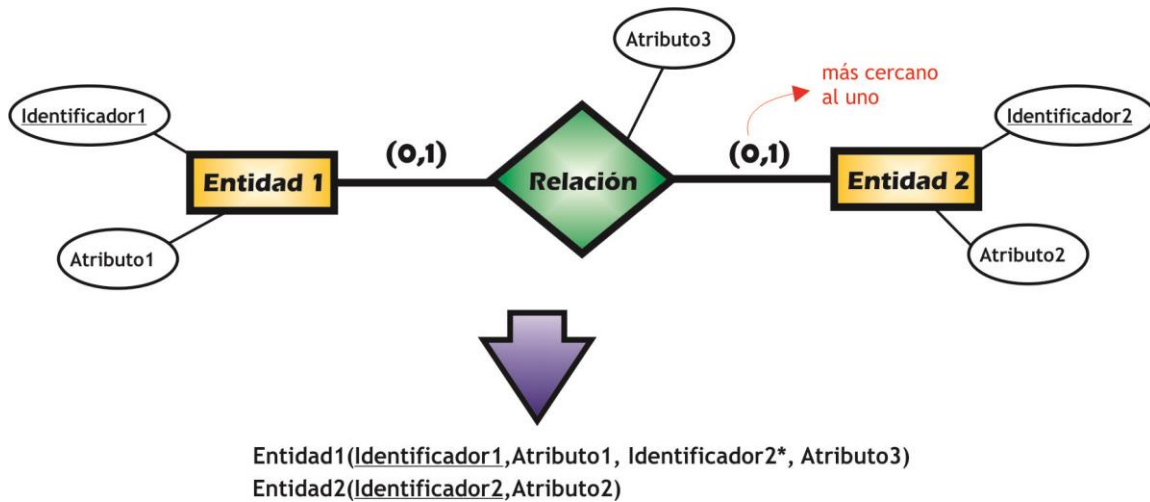


Ilustración 41, Paso a relacional de relaciones cero a cero, suponiendo que son los ejemplares de la entidad 2, los que más aparecen en la relación. El Identificador 2 pasa con restricción de unicidad.

relaciones recursivas

Las relaciones recursivas se tratan de la misma forma que las otras, sólo que hay que imaginar que la tabla se divide en dos, una por cada rol. Teniendo en cuenta eso, la solución es idéntica a lo ya resuelto en los casos anteriores.

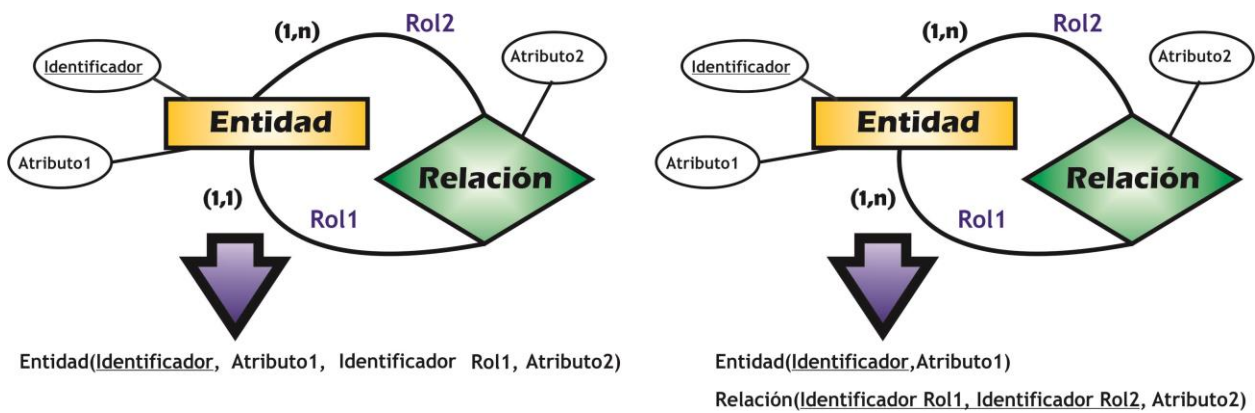


Ilustración 42, Transformación de relaciones recursivas en el modelo relacional

Es decir la forma de imaginarse las relaciones recursivas para resolverlas es:

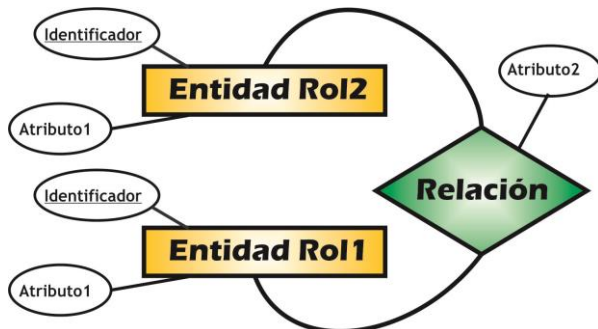


Ilustración 43, Abstracción para facilitar la conversión de relaciones recursivas, nos imaginamos dos entidades en lugar de una.

(2.5.3) entidades débiles

Toda entidad débil incorpora una relación implícita con una entidad fuerte. Esta relación no necesita incorporarse como tabla en el modelo relacional (al tratarse de una relación n a 1), bastará con añadir como atributo y clave foránea en la entidad débil, el identificador de la entidad fuerte.

En ocasiones el identificador de la entidad débil tiene como parte de su identificador al identificador de la entidad fuerte (por ejemplo si para identificar líneas de factura utilizamos el *número de línea* y *el número de factura*, clave de la entidad *factura*). En esos casos no hace falta añadir de nuevo como clave externa el identificador de la entidad fuerte (imagen de la derecha)

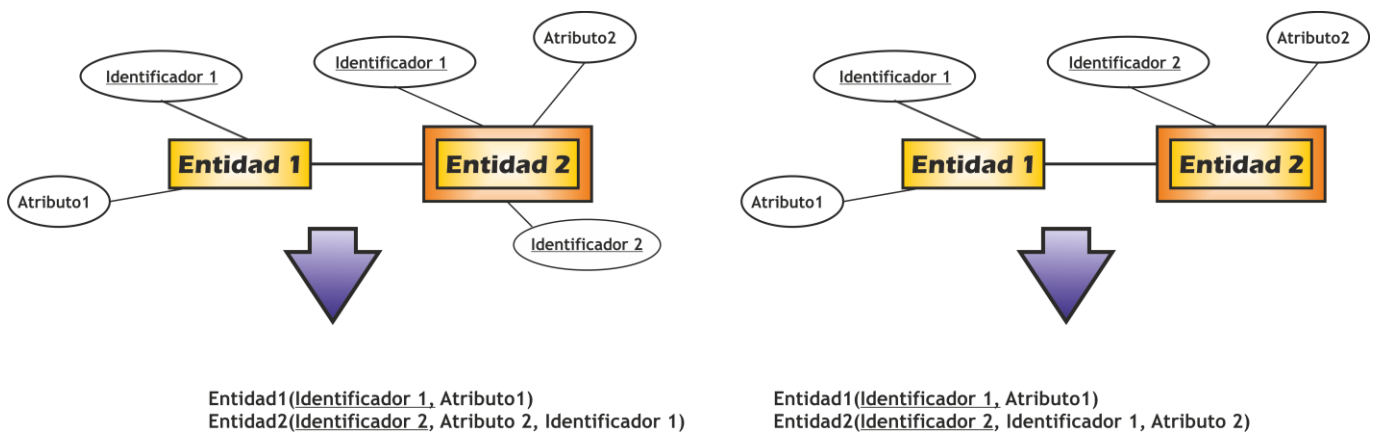


Ilustración 44, transformación de entidades débiles en el modelo relacional

(2.5.4) relaciones ISA

En el caso de las relaciones ISA, se siguen estas normas:

- (1) Tanto las superentidades como las subentidades generarán tablas en el modelo relacional (en el caso de que la ISA sea de tipo total, se podría incluso no hacer la superentidad y pasar todos sus atributos a las subentidades, pero no es recomendable porque puede complicar enormemente el esquema interno).
- (2) Los atributos se colocan en la tabla a la que se refiere a la entidad correspondiente.
- (3) Si las subentidades no tienen clave propia, se colocará como clave, la clave de su superentidad. Además esta clave heredada será clave secundaria, además de clave principal.
- (4) En el caso de que las subentidades tengan clave principal propia. Se colocará en las subentidades el identificador de la superentidad como clave secundaria, además será clave alternativa.

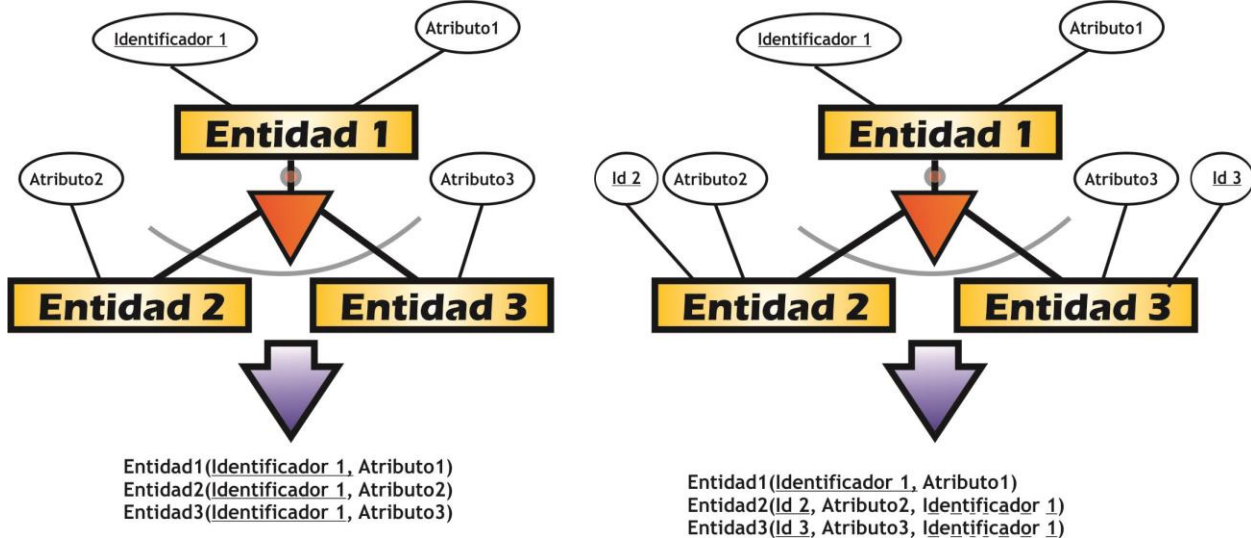


Ilustración 45, Proceso de transformación de relaciones ISA. A la izquierda cuando el identificador es heredado por las subentidades; a la derecha las subentidades tienen identificador propio. No varía el resultado por el tipo de ISA (parcial o total), aunque en el caso de las exclusivas, se suele indicar el arco en el resultado.

- (5) Si la ISA es exclusiva o no, no cambia el esquema relacional, pero sí habrá que tenerlo en cuenta para las restricciones futuras en el esquema interno (casi siempre se realizan mediante triggers), ya que en las exclusivas no puede haber repetición de la clave de la superentidad en ninguna subentidad.
- (6) No varía el resultado por ser total o parcial la relación ISA, el modelo relacional no puede marcar esa posibilidad. Pero es interesante tenerlo en cuenta (se suele añadir un comentario al diseño relacional para posibles restricciones a tener en cuenta)

(2.5.5) 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,.... Especial cuidado hay que tener con las relaciones ISA. 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.

(2.6) representación de esquemas de bases de datos relacionales

Ya se ha comentado como son los esquemas de datos relacionales. La manera formal es:

- PIEZA(Tipo, Modelo, Nombre, Apellido1, Apellido2)
- EMPRESA(CIF, Cod_Empresa, Nombre, Dirección)
- SUMINISTROS(Tipo, Modelo, Cod_Empresa, Precio)

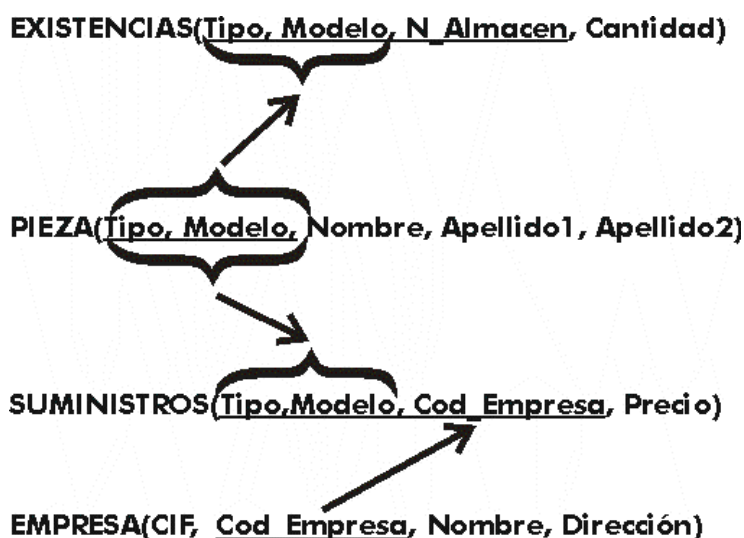
EXISTENCIAS(Tipo, Modelo, N_Almacen, Cantidad)

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.

(2.6.1) 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:



(2.6.2) esquemas relacionales derivados del modelo entidad/relación

Hay quien los llama **esquemas entidad/relación relacionales**, pero en general se consideran variantes del modelo entidad/relación. Sin embargo la mayoría no son capaces de representar lo mismo que el modelo entidad/relación de Chen, la razón estriba en que parten de que el modelo lógico es el relacional y por lo tanto están muy orientados a ese modelo.

modelo de patas de gallo

Quizá el modelo más famoso sea el llamado modelo de **pata de gallo** (*crow's foot* en inglés) utilizado en la metodología **Information Engineering** (*Ingeniería de la Información*) que tiene bastante popularidad; de hecho es el tipo de notación habitual en la mayoría de herramientas CASE. En él las cardinales se representan con símbolos en lugar de con números. El modelo de pata de gallo correspondiente al ejemplo anterior sería:

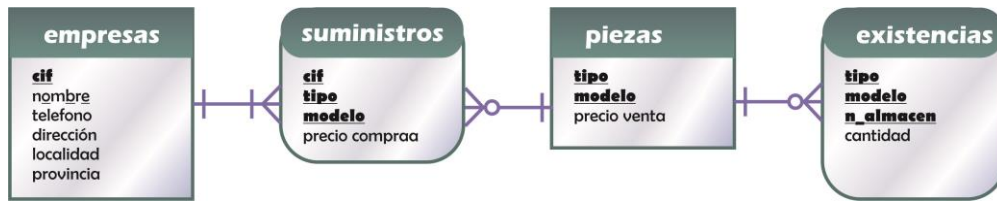


Ilustración 46, Diagrama estilo pata de gallo

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.

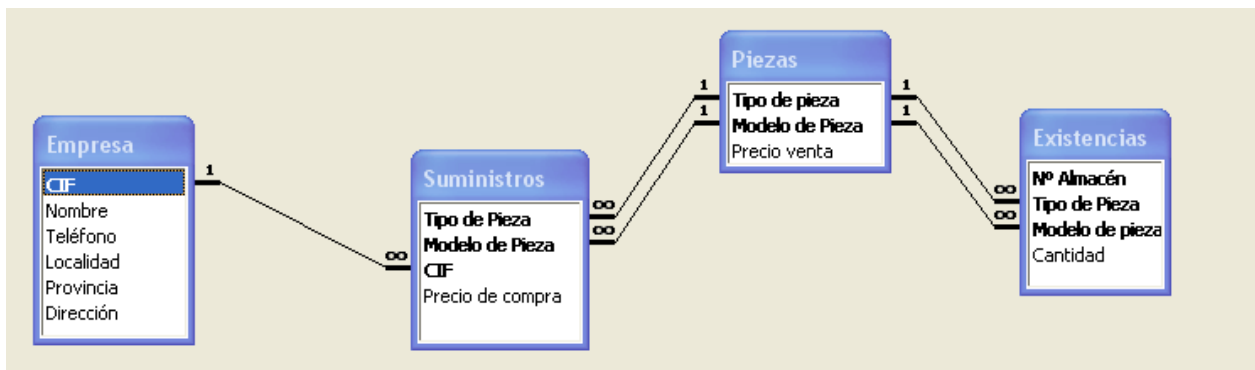
En el diagrama anterior (Ilustración 46) se puede examinar un modelo sencillo estilo pata de gallo. En estos diagramas la cardinalidad máxima *n* se dibuja con las famosas patas de gallo (↖), la cardinalidad mínima de tipo *ceros* con un círculo y la cardinalidad de tipo *uno* con una barra vertical. El hecho de que *suministros* y *existencias* tengan las esquinas redondeadas es para remarcar que representan relaciones entre entidades.

En cualquier caso tampoco hay un estándar unánimemente aceptado para este tipo de notación.

estilo Access

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

Ejemplo:



Es otra forma muy clara de representar relaciones y cardinalidades (aunque tiene problemas para representar relaciones de dos o más atributos).

completando esquemas

Sin duda los esquemas más completos son los que reflejan no sólo las cardinalidades sino también todas las restricciones (e incluso los tipos de datos, aunque esto ya es una competencia del esquema interno). Véase el esquema de la Ilustración 47. En ese esquema los símbolos funcionan de esta forma:

Símbolo	Ejemplo	Significado
Subrayado	<u>DNI</u>	Clave principal
Subrayado discontinuo	C <u>lave2</u>	Clave alternativa

°	Nombre °	No admite valores nulos (restricción NOT NULL)
*	Nombre *	No admite duplicados (restricción UNIQUE)

Además los campos que están el final de una flecha son claves secundarias.



Ilustración 47, Esquema relacional completo de la base de datos de un Video Club

En el esquema anterior las flechas representan cardinalidades *n* y los círculos cardinalidades de tipo *cero*. Las de tipo *uno* no tienen ningún símbolo asignado. A este tipo de diagramas (los de flechas y ceros) se les llama diagramas en **notación Bachman**.

En muchas **herramientas CASE**, los diagramas relacionales suelen representar las restricciones con letras. De hecho esta notación es la más habitual actualmente puesto que son las más completa (aunque a veces visualmente son un poco más difíciles de interpretar que la forma representada en la Ilustración 47).

Ejemplo de notación en pata de gallo con las restricciones usando abreviaturas:

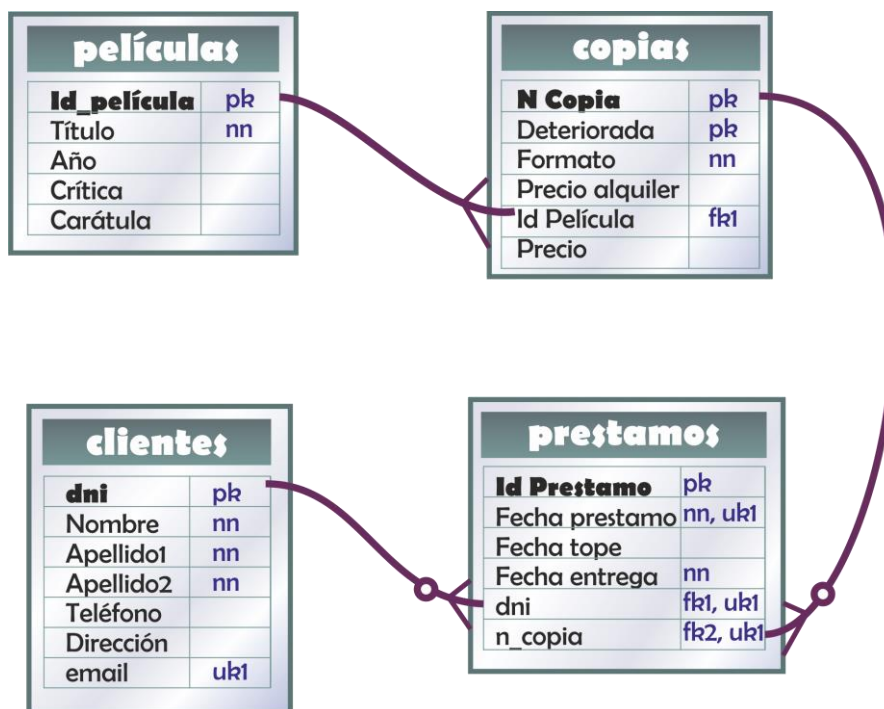


Ilustración 48, Esquema relacional usando notación en pata de gallo con las restricciones usando abreviaturas

En este caso los símbolos **PK** significan *Primary Key* (clave principal), **FK** es *Foreign Key* (clave secundaria) **UK** (o simplemente U) es *Unique* (unicidad) y **CK** es restricción de validación (*check*). Los números sirven para aclarar los atributos que forman parte de la restricción. Así sabemos que en la tabla de *prestamos*, *dni* forma una clave foránea y *n_copia* otra distinta; y que *fecha_prestamo*, *dni* y *n_copia* forman juntos una restricción de unicidad.

(2.7) normalización

(2.7.1) 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 elemento al que representan. Los datos de cada fila podrían referirse a más de un ejemplar de esa tabla 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.

(2.7.2) 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.

(2.7.3) 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 una 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 está en primera forma normal.

(2.7.4) 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**; es decir para un DNI concreto sólo hay un nombre posible. En la tabla del 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 $X \twoheadrightarrow Y$

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 ($Y \not\rightarrow X$). Entonces ocurre que X produce una dependencia funcional transitiva sobre Z .

Esto se denota como: ($X \twoheadrightarrow Z$)

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 $Y \not\rightarrow X$ (el código de la clase no depende funcionalmente del código tutor, un código tutor se puede corresponder con varios códigos de clase). Entonces $X \twoheadrightarrow Z$ (el código del departamento depende transitivamente del código de la clase).

(2.7.5) 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				
<u>DNI</u>	<u>Cod Curso</u>	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

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		
<u>DNI</u>	Nombre	Apellido1
12121219A	Pedro	Valiente
3457775G	Ana	Fernández
5674378J	Sara	Crespo

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

(2.7.6) 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				
<u>DNI</u>	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			
<u>DNI</u>	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	
<u>Cod Provincia</u>	Provincia
34	Palencia
47	Valladolid
08	Barcelona

(2.7.7) 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:

ORGANIZACIÓN		
<u>Trabajador</u>	<u>Departamento</u>	<u>Responsable</u>
Alex	Producción	Felipa
Arturo	Producción	Martín
Carlos	Ventas	Julio
Carlos	Producción	Felipa
Gabriela	Producción	Higinio
Luisa	Ventas	Eva
Luisa	Producción	Martín
Manuela	Ventas	Julio
Pedro	Ventas	Eva

La cuestión es que un trabajador o trabajadora puede trabajar en varios departamentos. En dicho departamento hay varios responsables, pero cada trabajador sólo tiene asignado uno. El detalle importante que no se ha tenido en cuenta, es que el o la responsable sólo puede ser responsable en un departamento.

Este detalle último produce una dependencia funcional ya que:

Responsable → Departamento

Por lo tanto hemos encontrado un determinante que no es clave candidata. No está por tanto en FNBC. En este caso la redundancia ocurre por mala selección de clave. La redundancia del departamento es completamente evitable. La solución sería:

PERSONAL	
<u>Trabajador</u>	<u>Responsable</u>
Alex	Felipa
Arturo	Martín
Carlos	Julio
Carlos	Felipa
Gabriela	Higinio
Luisa	Eva
Luisa	Martín
Manuela	Julio
Pedro	Eva

RESPONSABLES	
Responsables	Departamento
Felipa	Producción
Martín	Producción
Julio	Ventas
Higinio	Producción
Eva	Ventas

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

(2.7.8) 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 Y (es decir $X \twoheadrightarrow Y$), siendo X e Y 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 de forma multivaluada del curso y no del profesor en una dependencia multivaluada (no hay dependencia funcional ya que los posibles valores son varios). Para el par *Nº 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 no trivial es una dependencia funcional. Son triviales aquellas dependencias multivaluadas en las que el conjunto formado por el determinante y el implicado no forman la clave primaria de la tabla y además el implicado no forma parte del determinante: es decir si $X \twoheadrightarrow Y$ y además $Y \not\subset X$ y X, Y no es la clave de la tabla, tenemos una dependencia multivaluada no trivial (como ocurre en el ejemplo anterior).

Para la tabla anterior la solución serían dos tablas:

<u>Nº Curso</u>	<u>Material</u>
17	1
17	2
25	1
25	2
25	3

<u>Nº Curso</u>	<u>Profesor</u>
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

(2.7.9) 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**). Es decir una tabla formada por unas cuantas columnas de la tabla original.

La operación **JOIN** procedente también del álgebra relacional, consiste en formar una tabla con la unión de dos tablas. La tabla resultante estará formada por la combinación de todas las columnas y filas de ambas, excepto las columnas y filas repetidas.

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

quinta forma normal o forma normal de proyección-uni3n

Ocurre cuando una tabla est1 en 4FN y cada dependencia de uni3n (JOIN) en ella es implicada por las claves candidatas.

Es la más compleja y polémica de todas. Polémica pues no está claro en muchas ocasiones está muy claro que el paso a 5FN mejore la base de datos. Fue definida también por Fagin.

Es raro encontrarse este tipo de problemas cuando la normalización llega a 4FN. Se deben a restricciones semánticas especiales aplicadas sobre la tabla. Ejemplo:

<u>Proveedor</u>	<u>Material</u>	<u>Proyecto</u>
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. Así vista la tabla, no permite ninguna proyección en la que no perdamos datos.

Pero 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 anterior nos suministre también ese material para el 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. Si un nuevo proyecto necesitara el material 1, entonces habrá que pedirlo a los proveedores 1 y 2 (ya que en otros proyectos los hemos utilizado)

La dependencia de reunión que produce esta restricción es muy difícil de ver ya que es lejana. Para esa restricción esta proyección de tablas sería válida:

<u>Proveedor</u>	<u>Material</u>
1	1
1	2
2	1

<u>Material</u>	<u>Proyecto</u>
1	2
2	1
1	1

<u>Proveedor</u>	<u>Proyecto</u>
1	2
1	1
2	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 y esa descomposición no tenga como clave la clave original de la tabla.

(2.7.10) forma normal de dominio clave (FNDC)

Se la conoce más con sus siglas en inglés **DKNF**. Se trata de una forma normal enunciada también por **Fagin** en 1981 al darse cuenta de los problemas de redundancia que ocurrían con algunos dominios.

En este caso no se basó en dependencias entre los datos, sino que se basó en **restricciones de dominio** y **restricciones de clave**.

- **Restricciones de dominio.** Se trata de la restricción que hace que un determinado atributo obtenga sólo ciertos valores, los que estén de acuerdo a la definición de un determinado dominio.
- **Restricción de clave.** Es la restricción que permite que un atributo o un conjunto de atributos forme una clave candidata.

Fagin dice que una tabla está en FNDC si toda restricción sobre la tabla es consecuencia lógica de aplicar las restricciones de dominio y clave sobre la misma. Fagin demostró que si esto ocurría la tabla incluso estaba en 5FN.

Ejemplo:

Alumno	Nivel logrado	Nota
Antonio	Muy alto y eficiente	9
Marisa	Muy alto aunque sin trabajo constante	9
Mario	Alto con trabajo habitual	7
Luisa	Medio aunque con trabajo	5

Observando los datos de la tabla se observa que cuando la nota es superior a 9, en el nivel aparece la palabra alto, cuando es un 7 o un 8 medio, y un 5 o 6 sería medio. Es decir tenemos restricciones que no son ni de dominio ni de clave en esa tabla. Lo lógico sería:

Alumno	Nivel de trabajo	Nota
Antonio	Eficiente	9
Marisa	Trabajo constante	9
Mario	Trabajo habitual	7
Luisa	Trabajo medio	5

Nivel académico	Nota mínima	Nota máxima
Muy alto	9	10
Alto	7	8,99
Medio	5	6,99
Bajo	0	4,99

No se pierde información al diseñar las tablas de esta forma y de hecho es más eficiente para la base de datos.

(3)

SQL (I).

DDL y DML

(3.1) notas previas

(3.1.1) versión de SQL

Aunque estos apuntes sirven como guía de uso de SQL estándar, la base de datos que se utiliza como referencia fundamental es la base de datos Oracle. Normalmente se indican siempre las instrucciones para Oracle y para el SQL estándar. En las partes donde no se indique explícitamente diferencia, significará que Oracle coincide con el estándar.

Sin embargo hay que avisar que todos los ejemplos han sido probados para Oracle, mientras que no se puede decir lo mismo para SQL estándar. En el caso de SQL estándar, el software PostgreSQL se ha tenido muy en cuenta ya que parece el más respetuoso actualmente con el estándar.

No obstante debo disculparme porque es posible que muchos apartados se refieran sólo a Oracle y sobre todo los detalles de funcionamiento y resultados no han tenido en cuenta el estándar, sino sólo el funcionamiento de Oracle.

La razón de utilizar Oracle como base de trabajo se debe a su respeto por SQL estándar (aunque desde luego no tan estricto como PostgreSQL), es decir que no es excesivamente diferente; pero fundamentalmente por ser el SGBD de referencia más importante desde hace ya muchos años. De hecho lo que Oracle aporta de novedoso en cada versión, acaba formando parte del estándar futuro.

(3.1.2) 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 naranja claro con el reborde de color marrón anaranjado.

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 naranja claro, pero sin el reborde. Ejemplo:

```
SELECT nombre FROM cliente;
```

Los ejemplos sirven para escenificar una instrucción concreta, la sintaxis se utiliza para indicar las 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:

- **PALABRA** Cuando en la sintaxis se utiliza una palabra en negrita, significa que es una palabra que hay que escribir literalmente (aunque sin importar si en mayúsculas o minúsculas).
- texto. El texto que aparece en color normal sirve para indicar que no hay que escribirle 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.

- **texto en negrita.** Sirve para indicar texto o símbolos que hay que escribir de forma literal, pero que no son palabras reservadas del lenguaje.
- **[] (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. Las palabras separadas con este signo indican que se debe elegir una de entre todas las palabras.
- **... (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. Es sin duda el lenguaje fundamental para manejar una base de datos relacional.

SQL es un **lenguaje declarativo** en lo que lo importante es definir **qué** se desea hacer, por encima de **cómo** hacerlo (que es la forma de trabajar de los lenguajes de programación de aplicaciones como C o Java). Con este lenguaje se pretendía que las instrucciones se pudieran escribir como si fueran órdenes humanas; es decir, utilizar un lenguaje lo más natural posible. De ahí que se le considere un lenguaje de cuarta generación.

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.

(3.2.2) 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 pronuncia **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 como lenguaje de bases de datos relacionales.

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 2011 (**SQL2011**)

(3.2.3) funcionamiento

componentes de un entorno de ejecución SQL

Según la normativa ANSI/ISO cuando se ejecuta SQL, existen los siguientes elementos a tener en cuenta en todo el entorno involucrado en la ejecución de instrucciones SQL:

- Un **agente SQL**. Entendido como cualquier elemento que cause la ejecución de instrucciones SQL que serán recibidas por un cliente SQL
- Una **implementación SQL**. Se trata de un procesador software capaz de ejecutar las instrucciones pedidas por el agente SQL. Una implementación está compuesta por:
 - Un **cliente SQL**. Software conectado al agente que funciona como interfaz entre el agente SQL y el servidor SQL. Sirve para establecer conexiones entre sí mismo y el servidor SQL.
 - Un **servidor SQL** (puede haber varios). El software encargado de manejar los datos a los que la instrucción SQL lanzada por el agente hace referencia. Es el software que realmente realiza la instrucción, los datos los devuelve al cliente.

posibles agentes SQL. posibles modos de ejecución SQL

ejecución directa. SQL interactivo

Las instrucciones SQL se introducen a través de un cliente que está directamente conectado al servidor SQL; por lo que las instrucciones se traducen sin intermediarios y los resultados se muestran en el cliente.

Normalmente es un modo de trabajo incómodo, pero permite tener acceso a todas las capacidades del lenguaje SQL de la base de datos a la que estamos conectados.

ejecución incrustada o embebida

Las instrucciones SQL se colocan como parte del código de otro lenguaje que se considera anfitrión (C, Java, Pascal, Visual Basic,...). Al compilar el código se utiliza un precompilador de la propia base de datos para traducir el SQL y conectar la aplicación resultado con la base de datos a través de un software adaptador (**driver**) como **JDBC** u **ODBC** por ejemplo.

ejecución a través de clientes gráficos

Se trata de software que permite conectar a la base de datos a través de un cliente. El software permite manejar de forma gráfica la base de datos y las acciones realizadas son traducidas a SQL y enviadas al servidor. Los resultados recibidos vuelven a ser traducidos de forma gráfica para un manejo más cómodo

ejecución dinámica

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

(3.2.4) 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.

(3.3) elementos del lenguaje SQL

(3.3.1) 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 (DTL).** Administran las modificaciones creadas por las instrucciones DML. Lo forman las instrucciones **ROLLBACK y COMMIT.** Se las considera parte del **DML.**
- **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(),...**)
- **Literales.** Valores concretos para las consultas: números, textos, caracteres, ... Ejemplos: 2, 12.34, 'Avda Cardenal Cisneros'
- **Metadatos.** Obtenidos de la propia base de datos

(3.3.2) normas de escritura

- En SQL no se distingue entre mayúsculas y minúsculas.
- Las instrucciones finalizan con el signo de punto y coma
- Cualquier comando SQL (**SELECT**, **INSERT**,...) puede 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) DDL

(3.4.1) introducción

El DDL es la parte del lenguaje SQL que realiza la función de definición de datos del SGBD. Fundamentalmente se encarga de la creación, modificación y eliminación de los objetos de la base de datos (es decir de los **metadatos**). Por supuesto es el encargado de la creación de las 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 otras 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 generan acciones que no se pueden deshacer (salvo que dispongamos de alguna copia de seguridad).

(3.4.2) 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**) (**SYSDBA** en el caso de Oracle).

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 (parámetros de Oracle):

```
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;
```

(3.4.3) 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.

(3.4.4) creación de tablas

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 con comillas dobles. 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”**)

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 [DEFAULT valor] [restricciones]
[, ...]);
```

Ejemplo:

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

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

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 VARCHAR(25));
```

Se puede indicar un valor por defecto para el atributo mediante la cláusula **DEFAULT**:

```
CREATE TABLE Proveedores (
nombre VARCHAR(25),
localidad VARCHAR(30) DEFAULT 'Palencia');
```

De este modo si añadimos un proveedor y no indicamos localidad, se tomará Palencia como localidad de dicho Proveedor.

(3.4.5) tipos de datos

A la hora de crear tablas, hay que indicar el tipo de datos de cada campo. Necesitamos pues conocer los distintos tipos de datos. Estos son:

Descripción	Tipos Estándar SQL	Oracle SQL
Texto		
Texto de anchura fija	CHARACTER(<i>n</i>) CHAR(<i>n</i>)	CHAR(<i>n</i>)
Texto de anchura variable	CHARACTER VARYING(<i>n</i>) VARCHAR (<i>n</i>)	VARCHAR2(<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>)

Descripción	Tipos Estándar SQL	Oracle SQL
Texto de anchura variable para caracteres nacionales	NATIONAL CHARACTER VARYING(<i>n</i>) NATIONAL CHAR VARYING(<i>n</i>) NCHAR VARYING(<i>n</i>)	NVARCHAR2(<i>n</i>)
Números		
Enteros pequeños (2 bytes)	SMALLINT	
Enteros normales (4 bytes)	INTEGER INT	
Enteros largos (8 bytes)	BIGINT (en realidad no es estándar, pero es muy utilizado en muchas bases de datos)	
Enteros precisión decimal		NUMBER(<i>n</i>)
Decimal de coma variable	FLOAT DOUBLE DOUBLE PRECISION REAL	NUMBER
Decimal de coma fija	NUMERIC(<i>m,d</i>) DECIMAL(<i>m,d</i>)	NUMBER(<i>m,d</i>)
Fechas		
Fechas	DATE	DATE
Fecha y hora	TIMESTAMP	TIMESTAMP
Intervalos	INTERVAL	INTERVAL
Booleanos y binarios		
Lógicos	BOOLEAN BOOL	
Binarios	BIT BIT VARYING(<i>n</i>) VARBIT(<i>n</i>)	
Datos de gran tamaño		
Texto gran longitud	CHARACTER LARGE OBJECT CLOB	LONG (en desuso) CLOB
Binario de gran longitud	BINARY LARGE OBJECT BLOB	RAW (en desuso) LONG RAW (en desuso) BLOB

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

textos

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

- **VARCHAR** . Para textos de longitud variable. Su tamaño depende de la base de datos (en Oracle es de 4000). En Oracle se llama **VARCHAR2**, pero es posible seguir utilizando VARCHAR.
- **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 Oracle se llama **NVARCHAR2**.

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 VARCHAR2, 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.

números

En este capítulo se explican los tipos numéricos para el sistema Oracle; para SQL estándar consultar la tabla de tipos de datos.

En Oracle, el tipo **NUMBER** es un formato versátil que permite representar todo tipo de números. Su rango recoge números de entre 10^{-130} y $9,99999999999 * 10^{128}$. 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	345255.35
NUMBER(7)	345255.345	Da error de precisión
NUMBER(9,-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.

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. Las fechas no se pueden manejar directamente, normalmente se usan funciones de conversión. En el caso de Oracle se suele usar **TO_DATE** (que se detallará en el tema siguiente). Ejemplo:

```
TO_DATE('3/5/2007')
```

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..

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

```
/* 123 años y seis meses */  
INTERVAL '123-6' YEAR TO MONTH  
/* 123 años */  
INTERVAL '123' YEAR TO MONTH  
/* 6 meses */  
INTERVAL '6' MONTH TO MONTH
```

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**.

Ejemplos:

```
/* 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;
```

datos de gran tamaño

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

CLOB

Utilizado para almacenar datos de texto de gran tamaño (hasta 4 GB de texto)

BLOB

Utilizado para guardar datos binarios de hasta 4 GB de tamaño

(3.4.6) dominios

En Oracle se echa de menos una instrucción que forma parte del estándar SQL y que permite crear dominios. Sin embargo en SQL estándar sí hay esa posibilidad y de hecho es muy interesante. Se trata de **CREATE DOMAIN**:

```
CREATE DOMAIN name [AS] data_type  
[ DEFAULT expression ]  
[ restricciones [ ... ] ]
```

Ejemplo:

```
CREATE DOMAIN Tdireccion AS VARCHAR(3);
```

Gracias a esa instrucción podemos crear la siguiente tabla:

```
CREATE TABLE personal(  
    cod_pers SMALLINT,  
    nombre VARCHAR(30),
```

```
    direccion Tdireccion
```

```
);
```

Como se observa en la sintaxis, se puede indicar un valor por defecto al dominio e incluso establecer algunas restricciones (más adelante se explica cómo poner restricciones).

En el caso de Oracle se puede utilizar la instrucción **CREATE TYPE**, aunque no es sinónimo de ésta. De hecho **CREATE TYPE** es una instrucción objeto-relacional y permite crear tipos avanzados de datos (que no es lo mismo que un dominio).

(3.4.7) consultar las tablas del usuario

consultar el diccionario de datos

Todas las bases de datos disponen de posibilidades para consultar el diccionario de datos. Siguiendo las reglas de Codd, la forma de consultar los metadatos es la misma que en el resto de tablas. Es decir existen tablas (en realidad **vistas**) que en lugar de contener datos, contienen los metadatos. En el caso de SQL estándar, el diccionario de datos es accesible mediante el esquema de información (**INFORMATION_SCHEMA**), un esquema especial que contiene el conjunto de vistas con el que se pueden consultar ñps metadatos de la base de datos. En concreto la vista **INFORMATION_SCHEMA.TABLES** obtiene una vista de las tablas creadas. Es decir:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES
```

Esa instrucción muestra una tabla con diversas columnas, entre ellas la columna **TABLE_CATALOG** indica el catálogo en el que está la tabla, **TABLE_SCHEMA** el esquema en el que está la tabla y **TABLE_NAME** el nombre de la tabla.

Muchos SGBD respetan el estándar, pero en el caso de **Oracle** no. Oracle utiliza diversas vistas para mostrar las tablas de la base de datos. En concreto **USER_TABLES** y que contiene una lista de las tablas del usuario actual (o del esquema actual). Así para sacar la lista de tablas del usuario actual, se haría:

```
SELECT * FROM USER_TABLES;
```

Esta vista obtiene numerosas columnas, en concreto la columna **TABLES_NAME** muestra el nombre de cada tabla.

Otra vista es **ALL_TABLES** mostrará una lista de todas las tablas de la base de datos (no solo del usuario actual), aunque oculta las que el usuario no tiene derecho a ver. Finalmente **DBA_TABLES** es una tabla que contiene absolutamente todas las tablas del

sistema; esto es accesible sólo por el usuario administrador (**DBA**). En el caso de **ALL_TABLES** y de **DBA_TABLES**, la columna **OWNER** indica el nombre del propietario de la tabla.

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. Un ejemplo del 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)

obtener la lista de las columnas de las tablas

Otra posibilidad para poder consultar los datos referentes a las columnas de una tabla, es utilizar el diccionario de datos.

Oracle posee una vista llamada **USER_TAB_COLUMNS** que permite consultar todas las columnas de las tablas del esquema actual. Las vistas **ALL_TAB_COLUMNS** y **DBA_TAB_COLUMNS** muestran los datos del resto de tablas (la primera sólo de las tablas accesibles por el usuario).

En el caso de SQL estándar las columnas son accesibles mediante la vista **INFORMATION_SCHEMA.COLUMNS**

(3.4.8) 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.

(3.4.9) modificar tablas

cambiar de nombre a una tabla

De forma estándar (SQL estándar) se hace:

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

En Oracle, además de con la orden anterior, se realiza mediante la orden **RENAME** (que permite el cambio de nombre de cualquier objeto). Sintaxis:

```
RENAME nombreViejo TO nombreNuevo;
```

Pero por coherencia es mejor hacerlo de la primera forma (la del estándar).

borrar contenido de tablas

Oracle dispone de una orden no estándar para eliminar definitivamente los datos de una tabla; es la orden **TRUNCATE TABLE** seguida del nombre de la tabla a borrar. Hace que se elimine el contenido de la tabla, pero no la estructura de la tabla en sí. Incluso borra del archivo de datos el espacio ocupado por la 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 (hay que recordar que el orden de las columnas no importa). Ejemplo:

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

Muchas bases de datos (pero no Oracle) requieren escribir la palabra **COLUMN** tras la palabra **ADD**. Normalmente suele ser opcional

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 única columna de una tabla que sólo tiene esa columna (habrá que usar **DROP TABLE**).

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

Al igual que en el caso anterior, en SQL estándar se puede escribir el texto **COLUMN** tras la palabra **DROP**.

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
- Cualquier otro cambio sólo es posible si la tabla está vacía

Ejemplo:

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

En el caso de SQL estándar en lugar de **MODIFY** se emplea **ALTER** (que además opcionalmente puede ir seguida de **COLUMN**). Por ejemplo:

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

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**)

(3.4.10) 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 (algo poco recomendable) 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 tipoDeDatos [propiedades]  
[[CONSTRAINT nombreRestricción ]] tipoRestricción (columnas)  
[,siguienteCampo...]  
[,CONSTRAINT nombreRestricción tipoRestricción (columnas) ...])
```

Las restricciones tienen un nombre, se puede hacer que sea la base de datos la que les ponga nombre, pero entonces sería críptico. Por eso es mejor ponerle un nombre nosotros para que sea más fácil de recordar.

Los nombres de restricción no se pueden repetir para el mismo esquema, debemos de buscar nombres únicos. 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).

Desde la empresa Oracle se aconseja la siguiente 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
```

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)

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. 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**

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 VARCHAR(9) PRIMARY KEY,  
nombre VARCHAR(50)) ;
```

O, poniendo un nombre a la restricción:

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

Si la clave está formada por más de un campo:

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

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 (aplicando una restricción de integridad referencial) 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_nal_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 permitírse nos 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 eliminan 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** en ningún caso.

La sintaxis completa para añadir claves foráneas es:

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

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 | ON UPDATE  
    [SET NULL | CASCADE | DEFAULT]  
);
```

Ejemplo:

```
CREATE TABLE alquiler(dni VARCHAR(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,  
    CONSTRAINT pelicula_fk FOREIGN KEY (cod_pelicula)  
    REFERENCES peliculas(cod)  
    ON DELETE CASCADE  
);
```

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)  
);
```

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.

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 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
```

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. En SQL estándar sólo se pone **CASCADE** y no **CASCADE CONSTRAINTS**.

desactivar restricciones

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

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

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

activar restricciones

Anula la desactivación. Formato (Oracle):

```
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.

cambiar de nombre a las restricciones

Para hacerlo se utiliza este comando (Oracle):

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

mostrar restricciones

SQL estándar

En SQL estándar hay dos vistas del diccionario de datos que permiten visualizar la información sobre las restricciones aplicadas en la base de datos. Son:

- INFORMATION_SCHEMA.TABLE_CONSTRAINTS
- INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE
- INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE.

La primera permite analizar las restricciones colocada. Devuelve una tabla con la siguiente estructura:

Columna	Tipo de datos	Descripción
TABLE_CATALOG	texto	Muestra el nombre del catálogo al que pertenece la tabla a la que se puso la restricción
TABLE_SCHEMA	texto	Muestra el nombre del esquema al que pertenece la tabla a la que se puso la restricción
TABLE_NAME	texto	Muestra el nombre de la tabla a la que se puso la restricción
CONSTRAINT_CATALOG	texto	Catálogo en el que está almacenada la restricción
CONSTRAINT_SCHEMA	texto	Esquema al que pertenece la restricción
CONSTRAINT_NAME	texto	Nombre de la restricción
CONSTRAINT_TYPE	carácter	Indica el tipo de restricción, puede ser: CHECK (C), FOREIGN KEY (F), PRIMARY KEY (P) o UNIQUE (U)

Por su parte **INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE** obtiene información sobre las columnas a las que afecta la restricción. La tabla que obtiene es:

Columna	Tipo de datos	Descripción
TABLE_CATALOG	<i>texto</i>	Muestra el nombre del catálogo al que pertenece la tabla a la que se puso la restricción
TABLE_SCHEMA	<i>texto</i>	Muestra el nombre del esquema al que pertenece la tabla a la que se puso la restricción
TABLE_NAME	<i>texto</i>	Muestra el nombre de la tabla a la que se puso la restricción
CONSTRAINT_CATALOG	<i>texto</i>	Catálogo en el que está almacenada la restricción
CONSTRAINT_SCHEMA	<i>texto</i>	Esquema al que pertenece la restricción
CONSTRAINT_NAME	<i>texto</i>	Nombre de la restricción
COLUMN_NAME	<i>texto</i>	Nombre de cada columna a la que afecta la restricción.

En el caso de **INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE** simplemente nos dice el nombre de las restricciones y de las tablas a las que afecta.

Oracle

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:

Columna	Tipo de datos	Descripción
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: <ul style="list-style-type: none"> ➤ C. De tipo CHECK o NOT NULL ➤ P. PRIMARY KEY ➤ R. FOREIGN KEY ➤ 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_COL_CONSTRAINTS**.

(3.5) DML

(3.5.1) 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.

(3.5.2) 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:

```
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 apartados anteriores.

(3.5.3) 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='Orense';
```

```
UPDATE productos SET precio=precio*1.16;
```

El primer dato actualiza la provincia de los clientes de Orense 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 (en el ejemplo se utilizan funciones de fecha para conseguir que los partidos que se jugaban hoy, pasen a jugarse el martes):

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

En la condición se pueden utilizar cualquiera de los siguientes operadores de comparación:

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

Además se puede utilizar:

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

(3.5.4) 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. Ejemplo:

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

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.

(3.6) 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.

(3.6.2) 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.

(3.6.3) 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.

(3.6.4) 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.

(3.7) otras instrucciones DDL

(3.7.1) índices

Los índices son objetos que forman parte del esquema que hacen que las bases de datos aceleren las operaciones de consulta y ordenación sobre los campos a los que el índice hace referencia.

Se almacenan aparte de la tabla a la que hace referencia, lo que permite crearles y borrarles en cualquier momento.

Lo que realizan es una lista ordenada por la que Oracle puede acceder para facilitar la búsqueda de los datos. cada vez que se añade un nuevo registro, los índices involucrados se actualizan a fin de que su información esté al día. De ahí que cuantos más índices haya, más le cuesta a Oracle añadir registros, pero más rápidas se realizan las instrucciones de consulta.

La mayoría de los índices se crean de manera implícita, como consecuencia de las restricciones **PRIMARY KEY**, **UNIQUE** y **FOREIGN KEY**. Estos son índices obligatorios, por lo que les crea el propio SGBD.

creación de índices

Aparte de los índices obligatorios comentados anteriormente, se pueden crear índices de forma explícita. Éstos se crean para aquellos campos sobre los cuales se realizarán búsquedas e instrucciones de ordenación frecuente.

Sintaxis:

```
CREATE INDEX nombre  
ON tabla (columna1 [,columna2...])
```

Ejemplo:

```
CREATE INDEX nombre_completo  
ON clientes (apellido1, apellido2, nombre);
```

El ejemplo crea un índice para los campos apellido1, apellido2 y nombre. Esto no es lo mismo que crear un índice para cada campo, este índice es efectivo cuando se buscan u ordenan clientes usando los tres campos (*apellido1, apellido2, nombre*) a la vez.

Se aconseja crear índices en campos que:

- Contengan una gran cantidad de valores
- Contengan una gran cantidad de nulos
- Sean parte habitual de cláusulas **WHERE**, **GROUP BY** u **ORDER BY**
- Seann parte de listados de consultas de grandes tablas sobre las que casi siempre se muestran como mucho un 4% de su contenido.

No se aconseja en campos que:

- Pertenezcan a tablas pequeñas
- No se usen a menudo en las consultas
- Pertenecen a tablas cuyas consultas muestran menos de un 4% del total de registros
- Pertenecen a tablas que se actualizan frecuentemente
- Se utilizan en expresiones

Los índices se pueden crear utilizando expresiones complejas:

```
CREATE INDEX nombre_complejo  
ON clientes (UPPER(nombre));
```

Esos índices tienen sentido si en las consultas se utilizan exactamente esas expresiones.

lista de índices

Para ver la lista de índices en Oracle se utiliza la vista **USER_INDEXES**. Mientras que la vista **USER_IND_COLUMNS** muestra la lista de columnas que son utilizadas por índices.

borrar índices

La instrucción **DROP INDEX** seguida del nombre del índice permite eliminar el índice en cuestión.

(3.7.2) 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.

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;
```

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

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 (Oracle):

```
SELECT numeroPlanta.NEXTVAL FROM DUAL;
```

En SQL estándar:

```
SELECT nextval('numeroPlanta');
```

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** (en Oracle):

```
INSERT INTO plantas(num, uso)
VALUES(numeroPlanta.NEXTVAL, 'Suites');
```

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}}]
```

borrar secuencias

Lo hace el comando **DROP SEQUENCE** seguido del nombre de la secuencia a borrar.

lista de secuencias

En SQL estándar, a través de **INFORMATION_SCHEMA.SEQUENCES** podemos acceder a la información sobre todas las secuencias creadas. En Oracle se hace mediante la vista **USER_SEQUENCES** permite observar la lista de secuencias del usuario.

(3.7.3) sinónimos

En Oracle, 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.

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).

borrado

```
DROP SYNONYM nombre
```

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 de todos los esquemas a los que tenemos acceso.

(4)

SQL (II).

Consultas

(4.1) consultas de datos con SQL. DQL

(4.1.1) 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

(4.1.2) 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
- *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;
```

(4.2) cálculos

(4.2.1) 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 una 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 es el propio valor **NULL**.

Se puede utilizar cualquiera de los operadores aritméticos: suma (+), resta (-), multiplicación (*), división (/). Como es habitual, la multiplicación y la división tienen preferencia sobre la suma y la resta en el orden de ejecución de la instrucción; dicho orden se puede alterar mediante el uso de los paréntesis.

(4.2.2) 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 sería:

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	22	BI-22
BI	24	BI-24

En la mayoría de bases de datos, la función **CONCAT** (se describe más adelante) realiza la misma función.

(4.3) 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;
```

(4.3.1) 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

Operador	Significado
!=	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 (en especial al ordenar o comparar con el signo de mayor o menor, ya que la el orden ASCII no respeta el orden de cada alfabeto nacional). No obstante es un problema que tiende a arreglarse en la actualidad en todos los SGBD (en Oracle no existe problema alguno) especialmente si son compatibles con Unicode.

(4.3.2) 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
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, apellido1, apellido2 FROM personas
WHERE edad >= 25 AND edad <= 50;
/* Obtiene a la gente de más de 60 años o de menos de 20 */
SELECT nombre, apellido1, apellido2 FROM personas
WHERE edad > 60 OR edad < 20;
/* Obtiene a la gente de con primer apellido entre la A y la O */
SELECT nombre, apellido1, apellido2 FROM personas
WHERE apellido1 > 'A' AND apellido2 < 'Z';
```

(4.3.3) 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).

(4.3.4) IN

Permite obtener registros cuyos valores estén en una lista de valores:

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

Obtiene piezas cuyos precios sean 3, 5 u 8 (no valen ni el precio 4 ni el 6, por ejemplo).

(4.3.5) 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:

Símbolo	Significado
%	Una serie cualquiera de caracteres
_	Un carácter cualquiera

Ejemplos:

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

(4.3.6) IS NULL

Devuelve verdadero si el valor que examina es 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 en el caso contrario, cuando la expresión no es nula.

(4.3.7) 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

(4.4) 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** (para una sola tabla):

```
SELECT { * | [DISTINCT] {columna | expresión} [[AS] alias], ... }  
FROM tabla  
[WHERE condición]  
[ORDER BY expresión1 [,expresión2,...][{ASC|DESC}];
```

(4.5) funciones

(4.5.1) 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 devuelven un resultado que procede de un determinado cálculo. La mayoría de funciones precisan que se les envíe datos de entrada (**parámetros** o **argumentos**) que son necesarios para realizar el cálculo de la función. Este resultado, lógicamente depende de los parámetros enviados. Dichos parámetros se pasan entre paréntesis. De tal manera que la forma de invocar a una función es:

```
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**).

En este apartado se tratan las funciones del primer tipo (más adelante se comentan las de agrupación).

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.

En los siguientes apartados se describen algunas de las funciones más interesantes, las más importantes son las remarcadas con un fondo naranja más intenso.

(4.5.2) funciones numéricas

redondeos

Función	Descripción
ROUND (n,decimales)	Redondea el número al siguiente número con el número de decimales indicado más cercano. ROUND (8.239,2) devuelve 8.24
TRUNC (n,decimales)	Los decimales del número se cortan para que sólo aparezca el número de decimales indicado

matemáticas

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

(4.5.3) funciones de caracteres

conversión del texto a mayúsculas y minúsculas

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

funciones de transformación

Función	Descripción
RTRIM (texto)	Elimina los espacios a la derecha del texto
LTRIM (texto)	Elimina los espacios a la izquierda que posea el texto
TRIM (texto)	Elimina los espacios en blanco a la izquierda y la derecha del texto y los espacios dobles del interior.
TRIM (caracteres FROM texto)	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 (texto,n[,m])	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 (texto)	Obtiene el tamaño del texto
INSTR (texto, textoBuscado [,posInicial [, nAparición]])	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 (texto, textoABuscar, [textoReemplazo])	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

Función	Descripción
<p>TRANSLATE(texto, caracteresACambiar, caracteresSustitutivos)</p>	<p>Potentísima función que permite transformar caracteres. Los <i>caracteresACambiar</i> son los caracteres que se van a cambiar, los <i>caracteresSustitutivos</i> son los caracteres que reemplazan a los anteriores. De tal modo que el primer carácter a cambiar se cambia por el primer carácter sustitutivo, el segundo por el segundo y así sucesivamente. Ejemplo:</p> <pre>SELECT TRANSLATE('prueba','ue','wx') FROM DUAL;</pre> <p>El resultado sería el texto <i>prwxba</i>, de tal forma que la <i>u</i> se cambia por la <i>w</i> y la <i>e</i> por la <i>x</i>.</p> <p>Si la segunda cadena es más corta, los caracteres de la primera que no encuentran sustituto, se eliminan. Ejemplo:</p> <pre>SELECT TRANSLATE('prueba','ue','w') FROM DUAL;</pre> <p>Da como resultado <i>prwba</i></p>
<p>LPAD(texto, anchuraMáxima, [caracterDeRelleno]) RPAD(texto, anchuraMáxima, [caracterDeRelleno])</p>	<p>Rellena el texto a la izquierda (LPAD) o a la derecha (RPAD) con el carácter indicado para ocupar la anchura indicada.</p> <p>Si el texto es más grande que la anchura indicada, el texto se recorta.</p> <p>Si no se indica carácter de relleno se rellenará el espacio marcado con espacios en blanco.</p> <p>Ejemplo:</p> <pre>LPAD('Hola',10,'-')</pre> <p>da como resultado</p> <p><i>-----Hola</i></p>
<p>REVERSE(texto)</p>	<p>Invierte el texto (le da la vuelta)</p>

otras funciones de caracteres

Función	Descripción
<p>ASCII(carácter)</p>	<p>Devuelve el código ASCII del carácter indicado</p>
<p>CHR(número)</p>	<p>Devuelve el carácter correspondiente al código ASCII indicado</p>

Función	Descripción
<p>SOUNDEX(texto)</p>	<p>Devuelve el valor fonético del texto. Es una función muy interesante para buscar textos de los que se no se sabe con exactitud su escritura. Por ejemplo:</p> <pre data-bbox="715 477 1380 555">SELECT * FROM personas WHERE SOUNDEX(apellido1)=SOUNDEX('Smith')</pre> <p>En el ejemplo se busca a las personas cuyo primer apellido suena como <i>Smith</i></p>

(4.5.4) funciones de trabajo con nulos

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

Función	Descripción
<p>COALESCE(listaExpresiones)</p>	<p>Devuelve la primera de las expresiones que no es nula. Ejemplo²:</p> <pre data-bbox="715 1032 1380 1473">CREATE TABLE test (col1 VARCHAR2(1), col2 VARCHAR2(1), col3 VARCHAR2(1)); INSERT INTO test VALUES (NULL, 'B', 'C'); INSERT INTO test VALUES ('A', NULL, 'C'); INSERT INTO test VALUES (NULL, NULL, 'C'); INSERT INTO test VALUES ('A', 'B', 'C'); SELECT COALESCE(col1, col2, col3) FROM test;</pre> <p>El resultado es:</p> <pre data-bbox="715 1552 742 1682">B A C A</pre>
<p>NVL(valor,sustituto)</p>	<p>Si el <i>valor</i> es NULL, devuelve el valor <i>sustituto</i>; de otro modo, devuelve valor</p>

² Ejemplo tomado de http://www.psoug.org/reference/string_func.html

Función	Descripción
NVL2 (valor,sustituto1,sustituto2)	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>
NULLIF (valor1,valor2)	Devuelve nulo si <i>valor1</i> es igual a <i>valor2</i> . De otro modo devuelve <i>valor1</i>

(4.5.5) funciones de expresiones regulares

expresiones regulares

Las expresiones regulares son una parte cada vez más importante de los lenguajes de programación. Han existido desde que aparecieron los lenguajes de programación de alto nivel y cada vez tienen más importancia, hasta el punto de que lenguajes como **Perl** no tendrían cabida sin las expresiones regulares.

En el caso de las bases de datos, también tienen una enorme importancia, en especial para crear restricciones de validación (**CHECK**) que imponen una forma muy concreta que debe de cumplir el texto., Es lo que comúnmente se conoce como patrón o máscara de entrada.

Por ejemplo si nuestra base de datos almacena números de identidad españoles, éstos se forman por ocho números y una letra. Si no queremos permitir otra posibilidad (por ejemplo aceptar que se pueda introducir como *dni* al texto *abcdefghijkl*), entonces lo ideal es indicarlo mediante un check con una expresión regular, utilizando la función **REGEXP_LIKE** por ejemplo.

Una expresión regular está formada por un texto que contiene símbolos especiales que son los que permiten indicar claramente la expresión. Tradicionalmente hay dos normas de expresiones regulares:

- **POSIX**. *Portable Operating System Interface*, es el formato que procede del sistema **Unix** (ahí la equis en sus siglas) y que ha sido aceptado por el organismo **IEEE** (*Institute of Electrical and Electronics Engineers*) de estándares. Oracle es totalmente compatible con el estándar POSIX correspondiente y exactamente al el borrador estándar 1003.2/D11.2.
- **PCRE** (*Perl Compatible Regular Expressions*). Proceden del lenguaje Perl (precisamente famoso por su uso de las expresiones regulares) y es un formato en claro ascenso. Oracle admite muchas de las expresiones correspondiente a esta especificación (las más populares)

Además Oracle ha incluido la especificación que compatibiliza las expresiones regulares con las lenguas incluidas en Unicode, lo que evita problemas habituales con las expresiones regulares en otros lenguajes. A continuación se indican las posibilidades que tiene Oracle para indicar expresiones regulares.

caracteres de uso habitual en las expresiones regulares

símbolo	significado
<code>c</code>	Si <code>c</code> es un carácter cualquiera (por ejemplo <code>a</code> , <code>H</code> , <code>ñ</code> , <i>etc.</i>) indica, donde aparezca dentro de la expresión, que en esa posición debe aparecer dicho carácter para que la expresión sea válida. Es decir si nuestra expresión regular es simplemente <code>'a'</code> será válido para esa expresión cualquier texto que contenga la letra <code>a</code> .
<code>cde</code>	Siendo <code>c</code> , <code>d</code> , y <code>e</code> caracteres, indica que esos caracteres deben aparecer de esa manera en la expresión.
<code>^x</code>	Comenzar por. Indica el texto debe empezar por la expresión <code>x</code> . Por ejemplo: <code>'^Ja'</code> es una expresión que da por válido cualquier texto que empiece por <code>jota</code> y luego lleve una <code>a</code> .
<code>x\$</code>	Finalizar por. Indica que el String texto terminar con la expresión <code>x</code> .
<code>[cde]</code>	Opción, son válidos uno de estos caracteres: <code>c</code> , <code>d</code> ó <code>e</code>
<code>c-d</code>	Cumplen esa expresión los caracteres que, en orden ASCII, vayan del carácter <code>c</code> al carácter <code>d</code> . Por ejemplo <code>a-z</code> representa todas las letras minúsculas del alfabeto inglés.
<code>[^x]</code>	No es válido ninguno de los caracteres que cumplan la expresión <code>x</code> . Por ejemplo <code>[^dft]</code> indica que no son válidos los caracteres <code>d</code> , <code>f</code> ó <code>t</code> .
<code>.</code>	Cualquier carácter. El punto indica que en esa posición puede ir cualquier carácter
<code>x+</code>	La expresión a la izquierda de este símbolo se puede repetir una o más veces
<code>x*</code>	la expresión a la izquierda de este símbolo se puede repetir cero o más veces
<code>x?</code>	El carácter a la izquierda de este símbolo se puede repetir cero o una veces
<code>x{n}</code>	Significa que la expresión <code>x</code> aparecerá <code>n</code> veces, siendo <code>n</code> un número entero positivo.
<code>x{n,}</code>	Significa que la expresión <code>x</code> aparecerá <code>n</code> o más veces
<code>x{m,n}</code>	Significa que la expresión <code>x</code> aparecerá de <code>m</code> a <code>n</code> veces.
<code>(x)</code>	Permite indicar un subpatrón dentro del paréntesis. Ayuda a formar expresiones regulares complejas.
<code>x y</code>	La barra indica que las expresiones <code>x</code> e <code>y</code> son opcionales, se puede cumplir una u otra.

caracteres especiales

Nos permiten indicar caracteres que de otra forma no se pueden escribir (como el salto de línea por ejemplo), ase les llama también secuencias de escape y proceden del lenguaje C.

carácter	significado
\t	Tabulador
\n	Salto de línea
\r	Salto de carro
\v	Tabulador vertical
\\	La propia barra

También se utiliza si necesitamos poner como literales algunos de los símbolos especiales de las expresiones regulares. Por ejemplo el símbolo punto (.) significa cualquier carácter, pero si indicamos \. entonces es el propio carácter punto. Por ejemplo si nuestra expresión verifica que un texto contiene una dirección de correo electrónico, pondríamos: `.*@\.*\.*` y eso daría válido cualquier texto que empiece con lo que sea (.*), que luego tenga el símbolo @, luego lo que sea (.*), luego obligatoriamente, un punto (\.) y luego lo que sea.

Lo mismo para otros símbolos: `\$, \[, \], ...`

símbolos procedentes del modelo POSIX

Deben utilizarse siempre dentro de los corchetes, así por ejemplo si nuestra expresión se refiere a una letra, se pondría `[[:alpha:]]` y si es cualquier carácter excepto letra, se usa `[^[:alpha:]]` y si fuera una letra o el número5, sería: `[[:alpha:]]5`

clase	significado
<code>[[:alpha:]]</code>	Letra
<code>[[:alnum:]]</code>	Letra o número
<code>[[:blank:]]</code>	Espacio o tabulador
<code>[[:cntrl:]]</code>	Carácter de control
<code>[[:digit:]]</code>	Número
<code>[[:graph:]]</code>	Carácter visible (no valen ni espacios ni caracteres de control)
<code>[[:lower:]]</code>	Letra minúscula
<code>[[:print:]]</code>	Carácter visible o el espacio en blanco
<code>[[:punct:]]</code>	Caracteres de puntuación
<code>[[:space:]]</code>	Espacios en blanco y símbolos que añaden espacios (como el salto de línea, tabulador,...)
<code>[[:upper:]]</code>	Letra mayúscula
<code>[[:xdigit:]]</code>	Dígito hexadecimal

símbolos procedentes del modelo Perl de expresión regular (y válidos en Oracle)

clases	significado
<code>\d</code>	Dígito, vale cualquier dígito numérico
<code>\D</code>	Todo menos dígito
<code>\s</code>	Espacio en blanco
<code>\S</code>	Cualquier carácter salvo el espacio en blanco
<code>\w</code>	<i>Word</i> , carácter válido dentro de los que PHP considera para identificar variables. Es decir, letras, números o el guion bajo.
<code>\W</code>	Todo lo que no sea un carácter de tipo <i>Word</i> .

ejemplos de expresiones regulares

DNI (ocho números y una letra).

```
'^[0-9]{8}[A-Z]$'
```

Se delimita entre `^` y `$` para que exactamente el contenido del texto que cumpla la expresión sea así. De otro modo este texto cumpliría la expresión:

```
holhola17672612Cholahola → No es válido, gracias a ^y $
```

Nº de serie con una o dos letras mayúsculas, tres números, seguido de una letra un guión y dos números:

```
'^[A-Z]{1,2}[0-9]{3}[A-Z]-[0-9]{2}$'
```

Código postal, números del 01000 al 52999 (se hacen tres grupos: del 10000 al 19999; del 01000 al 09000; y del 51000 al 52999; de otro modo dejaríamos algún caso de más)

```
'^([1-4][0-9]{4}) | (0[1-9][0-9]{3}) | (5[1-2][0-9]{3})$'
```

Correo electrónico (al menos una letra con posibilidad de punto (y si lo hay habría letra a sus dos lados y al menos un punto tras el símbolo de arroba, al final un punto y nombre de dominio de dos o tres letras; `\w` significa *palabra*)

```
'^[\\w]+(\\.\\w+)*@[\\w]+(\\.\\w+)*\\.([a-z]{2,3})$'
```

funciones de Oracle para uso con expresiones regulares

Función	Descripción
---------	-------------

Función	Descripción
<p>REGEXP_LIKE(texto, expresionRegular [,flag])</p>	<p>Devuelve verdadero si el texto cumple la expresión regular.</p> <p>El flag que se puede indicar como tercer parámetro es una de estas letras (entrecomilladas):</p> <ul style="list-style-type: none"> • i. No distingue entre mayúsculas y minúsculas • c. Distingue entre mayúsculas y minúsculas • n. Hace que el punto represente a todos los caracteres excepto al salto de línea (\n). • x. Ignora los espacios en blanco
<p>REGEXP_COUNT(texto, expresionRegular [,posición [,flag]])</p>	<p>Cuenta las veces que aparece la expresión regular en el texto. Se puede especificar la posición en el texto desde la que empezamos a buscar y el parámetro flag comentado anteriormente.</p>
<p>REGEXP_INSTR(texto, expresionRegular [,posInicial [,nAparición [,modoPos [,flag [,subexpr]]]]])</p>	<p>Es una versión de la función INSTR, pero con más potencia. Busca la expresión regular en el texto y devuelve su posición (o cero si no la encuentra).</p> <p>Se puede empezar a buscar desde una posición (indicando un número en el parámetro posInicial) e incluso indicar que buscamos no la primera aparición (que es como funciona por defecto) sino que buscamos el número de aparición que indiquemos en el cuarto parámetro.</p> <p>El quinto parámetro vale cero (valor por defecto) si la posición devuelta indica el primer carácter en el que aparece la expresión; y vale uno si la posición devuelta es la del siguiente carácter respecto al texto que cumple la expresión regular.</p> <p>flag es el parámetro comentado en la función REGEXP_LIKE.</p>
<p>REGEXP_REPLACE(texto, expresionRegular [, nuevoTexto [,posInicial [,nAparición [,flag]]]])</p>	<p>Busca la expresión regular en el texto y reemplaza todo el texto que la cumple por el nuevo texto indicado.</p> <p>Si no hay nuevoTexto, entonces se eliminan los caracteres que cumplan la expresión regular</p>

Función	Descripción
REGEXP_SUBSTR (texto, expresionRegular [,posInicial [, nAparición [,flag [,subexpr]]]])	Busca la expresión regular en el texto y extrae los caracteres que cumplen la expresión. Es, por lo tanto, una versión más potente de SUBSTR .

(4.5.6) funciones de fecha y manejo de fechas e intervalos

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.

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** para hacer cálculos. Gracias a ello se permiten sumar horas o minutos por ejemplo a los datos de tipo **TIMESTAMP**.

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

calcular fechas

Función	Descripción
ADD_MONTHS (fecha,n)	Añade a la fecha el número de meses indicado por <i>n</i>
MONTHS_BETWEEN (fecha1, fecha2)	Obtiene la diferencia en meses entre las dos fechas (puede ser decimal)
NEXT_DAY (fecha,día)	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 (fecha)	Obtiene el último día del mes al que pertenece la fecha. Devuelve un valor DATE
EXTRACT (valor FROM fecha)	Extrae un valor de una fecha concreta. El valor puede ser day (día), month (mes), year (año), etc.
GREATEST (fecha1, fecha2,..)	Devuelve la fecha más moderna la lista
LEAST (fecha1, fecha2,..)	Devuelve la fecha más antigua la lista

Función	Descripción
ROUND (fecha [, 'formato'])	Redondea la fecha al valor de aplicar el formato a la fecha. El formato puede ser: '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 (fecha [formato])	Igual que el anterior pero trunca la fecha en lugar de redondearla.

(4.5.7) 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.

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).

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):

Símbolo	Significado
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
D	Día de la semana (del 1 al 7)

Símbolo	Significado
DD	Día del mes en formato de dos cifras (del 1 al 31)
DDD	Día del año
Q	Semestre
WW	Semana 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 */
```

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
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)

TO_NUMBER

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

TO_DATE

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

CAST

Función muy versátil que permite convertir el resultado a un tipo concreto. Sintaxis:

```
CAST(expresión AS tipoDatos)
```

Ejemplo:

```
SELECT CAST(2.34567 AS NUMBER(7,6)) FROM DUAL;
```

Lo interesante es que puede convertir de un tipo a otro. Por ejemplo imaginemos que tenemos una columna en una tabla mal planteada en la que el precio de las cosas se ha escrito en Euros. Los datos son (se muestra sólo la columna precio):

precio
25.2 €
2.8 €
123.65 €
.78 €
.123 €
20 €

Imaginemos que queremos doblar el precio, no podremos porque la columna es de tipo texto, por ello debemos tomar sólo la parte numérica y convertirla a número, después podremos mostrar los precios multiplicados por dos:

```
SELECT 2 * CAST(SUBSTR(precio,1,INSTR(precio,'€')-2) AS NUMBER)  
FROM precios;
```

La combinación de SUBSTR e INSTR es para obtener sólo los números. Incluso es posible que haya que utilizar REPLACE para cambiar los puntos por comas (para utilizar el separador decimal del idioma español).

(4.5.8) 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.

(4.5.9) funciones XML

Oracle tiene la posibilidad de conseguir resultados compatibles con el lenguaje XML (que después podríamos utilizar para formatear, analizar, consultar,... utilizando herramientas propias de XML).

Aprender XML va más allá de los objetivos de estos apuntes, si se desea aprender más al respecto desde www.jorgesanchez.net hay varios manuales de XML (y hay muchos más y mejores escritos por otros autores tanto libremente en Internet como en libros).

XMLElement

Es una de las funciones más utilizadas para mostrar información en forma XML. Su sintaxis es:

```
XMLElement(nombreElemento[,cláusulaDeAtributos][,valor1[,...]])
```

Es una función muy poderosa. En su forma más fácil simplemente se indica el elemento:

```
SELECT XMLElement("prueba") FROM DUAL;
```

Esa consulta (sobre la tabla estándar de pruebas DUAL) muestra:

```
<prueba></prueba>
```

Es decir un elemento XML sin contenido. Sin embargo esta otra:

```
SELECT XMLElement("nombre",nombre) as c1 FROM personas;
```

Muestra un resultado como, por ejemplo:

c1
<nombre>Vanesa</nombre>
<nombre>Luis</nombre>
<nombre>Luisa</nombre>
<nombre>Mercedes</nombre>
<nombre>Laura</nombre>
<nombre>Matías</nombre>

```
<nombre>Esperanza</nombre>
```

Para cada persona aparece una fila y en ella el nombre dentro de un elemento llamado *nombre*.

Otro ejemplo más complejo (con subelementos):

```
SELECT XMLElement(
    "persona",
    XMLElement("nombre",nombre),
    XMLElement("apellido1",apellido1),
    XMLElement("apellido2",apellido2)
) AS datos
FROM alumnos;
```

Resultado:

datos
<persona><nombre>Vanesa</nombre><apellido1>Megino</apellido1><apellido2>Sánchez</apellido2></persona>
<persona><nombre>Luis</nombre><apellido1>Abadía</apellido1><apellido2>Castrourdiales</apellido2></persona>
<persona><nombre>Luisa</nombre><apellido1>Calvo</apellido1><apellido2>Calvo</apellido2></persona>
<persona><nombre>Mercedes</nombre><apellido1>Aragón</apellido1><apellido2>Cantor</apellido2></persona>
<persona><nombre>Laura</nombre><apellido1>Albero</apellido1><apellido2>Siniestra</apellido2></persona>
<persona><nombre>Matías</nombre><apellido1>Hierro</apellido1><apellido2>Amo</apellido2></persona>
<persona><nombre>Esperanza</nombre><apellido1>Castro</apellido1><apellido2>González</apellido2></persona>

Podemos también usar cláusulas de atributos, eso es posible con **XMLAttributes**, otra función que es capaz de usar atributos. Su sintaxis es:

```
XMLAttributes(valor1 AS alias1 [,valor2 AS alias2 [,...]])
```

El valor es el contenido del atributo y los alias el nombre del atributo. Sólo puede haber una cláusula XMLAttributes dentro de una XMLElement. Ejemplo:

```
SELECT XMLElement(
    "persona",
    XMLAttributes(sexo AS "sexo",dni AS "id"),
    XMLElement("nombre",nombre),
    XMLElement("apellido1",apellido1),
    XMLElement("apellido2",apellido2)
) AS datos FROM personas;
```

Resultado:

datos
<persona sexo="m" id="12030430 "> <nombre>Vanesa</nombre><apellido1>Megino</apellido1><apellido2>Sánchez</apellido2></persona>
<persona sexo="h" id="12049558 "> <nombre>Luis</nombre><apellido1>Abadía</apellido1><apellido2>Castrourdiales</apellido2></persona>
<persona sexo="m" id="12080980 "> <nombre>Luisa</nombre><apellido1>Calvo</apellido1><apellido2>Calvo</apellido2></persona>
<persona sexo="m" id="12162777 "> <nombre>Mercedes</nombre><apellido1>Aragón</apellido1><apellido2>Cantor</apellido2></persona>
<persona sexo="m" id="12178979 "> ><nombre>Laura</nombre><apellido1>Albero</apellido1><apellido2>Siniestra</apellido2></persona>
<persona sexo="h" id="12213987 "> <nombre>Matías</nombre><apellido1>Hierro</apellido1><apellido2>Amo</apellido2></persona>
<persona sexo="m" id="12239347 "> <nombre>Esperanza</nombre><apellido1>Castro</apellido1><apellido2>González</apellido2></persona>

XMLForest

Se trata de una función similar a **XMLElement**, sólo que de uso más cómodo. Muestra elementos XML a partir de datos de las tablas. Su sintaxis es:

XMLForest(nombre1 [AS alias1] [,nombre2 [AS alias2]][,...])

Se maneja igual que **XMLAttribute** pero ahora se refiere a elementos. Ejemplo:

SELECT XMLForest(nombre,apellido1 AS ap1) FROM alumnos;

datos
<NOMBRE>Vanesa</NOMBRE><AP1>Megino</AP1>
<NOMBRE>Luis</NOMBRE><AP1>Abadía</AP1>
<NOMBRE>Luisa</NOMBRE><AP1>Calvo</AP1>
<NOMBRE>Mercedes</NOMBRE><AP1>Aragón</AP1>
<NOMBRE>Laura</NOMBRE><AP1>Albero</AP1>
<NOMBRE>Matías</NOMBRE><AP1>Hierro</AP1>
<NOMBRE>Esperanza</NOMBRE><AP1>Castro</AP1>

El alias se ha usado para la segunda etiqueta que es **AP1** en lugar de **APELLIDO1**.

XMLAgg

Es una función muy poderosa que permite realizar agrupaciones al estilo de la cláusula **GROUP BY**. Es la función que permite obtener resultados muy espectaculares, como si realmente al final tuviéramos un archivo XML con los datos. Sintaxis:

XMLAgg(resultadoDeFuncionXML [cláusula de orden])

La cláusula de orden permite el uso de la cláusula ORDER BY dentro de la función. Con **XMLAgg** conseguimos que todos los resultados se puedan asociar a un solo elemento. Ejemplo:

```
SELECT XMLElement(  
  "listado",  
  XMLAgg(XMLElement(  
    "persona",  
    XMLForest(  
      nombre as "nombre",  
      apellido1 as "apellido1",  
      apellido2 as "apellido2"  
    )  
  ) ORDER BY dni  
)  
FROM personas  
WHERE edad>60;
```

Se utilizan comillas dobles y alias en **XMLForest** para asegurar que los elementos XML salen en minúsculas (que es lo más recomendable para XML). Aunque el resultado saldría seguido y en una sola fila, se muestra a continuación en una forma más fácil de leer en XML.

```
<listado>  
  <persona>  
    <nombre>Luisa</nombre>  
    <apellido1>Calvo</apellido1>  
    <apellido2>Calvo</apellido2>  
  </persona>  
  <persona>  
    <nombre>Antonio</nombre>  
    <apellido1>Sánchez</apellido1>  
    <apellido2>Fernández</apellido2>  
  </persona>  
  <persona>  
    <nombre>Pablo</nombre>  
    <apellido1>Varilla</apellido1>  
    <apellido2>Sánchez</apellido2>  
  </persona>  
  <persona>  
    <nombre>Jorge Luis</nombre>  
    <apellido1>Andérez</apellido1>  
    <apellido2>Sánchez</apellido2>  
  </persona>
```

```
<persona>
  <nombre>Johann</nombre>
  <apellido1>Mazukelle</apellido1>
  <apellido2>Nburu</apellido2>
</persona>
</listado>
```

XMLAgg ha conseguido meter todos los elementos dentro del grupo creado, listado. Un ejemplo más complejo sería (combinando todas las funciones anteriores y usando varias tablas, cuya técnica se explica en el apartado (4.6)):

```
SELECT XMLElement(
  "listado",
  XMLAgg(XMLElement(
    "empresa",
    XMLForest(e.nombre as "nombre"),
    (
      SELECT XMLElement(
        "trabajador",
        XMLAgg(
          XMLForest(
            nombre as "nombre",
            apellido1 as "apellido1",
            apellido2 as "apellido2")
          )
        ) FROM alumnos a
        WHERE a.cif=e.cif
      )
    ) ORDER BY e.nombre
  )
)
FROM empresas e;
```

El resultado (mostrado de una forma más legible, porque en realidad se mostraría todo seguido):

```
<listado>
  <empresa>
    <nombre>Class</nombre>
    <trabajador>
      <nombre>Matías</nombre>
      <apellido1>Hierro</apellido1>
      <apellido2>Amo</apellido2>
      <nombre>Esperanza</nombre>
      <apellido1>Castro</apellido1>
```

```
<apellido2>González</apellido2>
</trabajador>
</empresa>
<empresa>
  <nombre>FRT S. A.</nombre>
  <trabajador>
    <nombre>Luis</nombre>
    <apellido1>Abadía</apellido1>
    <apellido2>Castrourdiales</apellido2>
    <nombre>Mercedes</nombre>
    <apellido1>Aragón</apellido1>
    <apellido2>Cantor</apellido2>
  </trabajador>
</empresa>
<empresa>
  <nombre>Vidrios Carmelo</nombre>
  <trabajador>
    <nombre>Luisa</nombre>
    <apellido1>Calvo</apellido1>
    <apellido2>Calvo</apellido2>
  </trabajador>
</empresa>
</listado>
```

(4.6) 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.

(4.6.1) 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

(4.6.2) asociando tablas

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

```
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;
```

(4.6.3) 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;
```

(4.6.4) 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.columa=tabla2.columa)]
```

Se describen sus posibilidades en los siguientes apartados.

CROSS JOIN

Utilizando la opción **CROSS JOIN** se realiza un producto cruzado entre las tablas indicadas. Eso significa que cada tupla de la primera tabla se combina con cada tupla de la segunda tabla. Es decir si la primera tabla tiene 10 filas y la segunda otras 10, como resultado se obtienen 100 filas, resultado de combinar todas entre sí. Ejemplo:

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

No es una operación muy utilizada, aunque posibilita resolver consultas extremadamente complicadas.

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.

Hay que asegurarse de que sólo son las claves principales y secundarias de las tablas relacionadas, las columnas en las que el nombre coincide, de otro modo fallaría la asociación y la consulta no funcionaría.

JOIN USING

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

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

Las columnas deben de tener exactamente el mismo nombre en ambas tablas-

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);
```

relaciones externas

Utilizando las formas anteriores de relacionar tablas (incluidas las explicadas con la sintaxis SQL 92), sólo aparecen en el resultado de la consulta filas presentes en las tablas relacionadas. Es decir en esta consulta:

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

```
piezas.modelo=existencias.modelo);
```

Sólo aparecen piezas presentes en la tabla de existencias. Si hay piezas que no están en existencias, éstas no aparecen (y si hay existencias que no están en la tabla de piezas, tampoco salen).

Por ello se permite utilizar relaciones laterales o externas (*outer join*). Su sintaxis es:

```
...  
{LEFT | RIGHT | FULL} OUTER JOIN tabla  
{ON(condición) | USING (expresion)}  
...
```

Así esta consulta:

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

Obtiene los datos de las piezas estén o no relacionadas con datos de la tabla de existencias (la tabla **LEFT** sería piezas por que es la que está a la izquierda del **JOIN**).

En la consulta anterior, 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 (piezas sin existencias relacionadas y viceversa).

consultas de no coincidentes

Un caso típico de uso de las relaciones externas es el uso de consultas de no coincidentes. Estas consultas sustituyen de forma más eficiente al operador **NOT IN** (se explica más adelante) ya que permiten comprobar filas de una tabla que no están presentes en una segunda tabla que se relaciona con la primera a través de alguna clave secundaria.

Por ejemplo supongamos que tenemos una base de datos que posee una tabla de empresas y otra de trabajadores de las empresas. Ambas tablas supongamos que se relacionan a través del CIF de la empresa. Si existen empresas de las que no tenemos ningún trabajador podremos saberlo mediante esta consulta:

```
SELECT e.nombre FROM empresas e  
LEFT OUTER JOIN trabajadores t ON (t.cif=e.cif)  
WHERE t.dni IS NULL;
```

En la consulta anterior gracias a utilizar el operador **LEFT OUTER**, obtenemos la lista completa de empresas (tengan o no trabajadores). Si eliminamos aquellas filas con el **dni** de los trabajadores nulo, entonces salen las empresas sin trabajadores relacionados (sólo las empresas sin trabajadores mostrarán valores nulos en los datos de los trabajadores).

(4.7) 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 condicionesDeGrupo]
[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:

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**

(4.7.1) 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 (expresión)	Suma los valores de la expresión
AVG (expresión)	Calcula la media aritmética sobre la expresión indicada
MIN (expresión)	Mínimo valor que toma la expresión indicada
MAX (expresión)	Máximo valor que toma la expresión indicada
STDDEV (expresión)	Calcula la desviación estándar
VARIANCE (expresión)	Calcula la varianza

Todas las funciones de la tabla anterior 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

(4.7.2) 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:

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

(4.8) subconsultas

(4.8.1) uso de subconsultas simples

Se trata de una técnica que permite utilizar el resultado de una tabla SELECT en otra consulta SELECT. Permite solucionar consultas que requieren para funcionar el resultado previo de otra consulta.

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');
```

En realidad lo primero que hace la base de datos es calcular el resultado de la subconsulta:

```
SELECT nombre_empleado, paga
FROM empleados
WHERE paga < ← 2500
    (SELECT paga FROM empleados
     WHERE nombre_empleado='Martina')
AND paga > ← 1870
    (SELECT paga FROM empleados
     WHERE nombre_empleado='Luis');
```

La última consulta obtiene los empleados cuyas pagas estén entre lo que gana Luís (1870 euros) y lo que gana Martina (2500) .

Las subconsultas siempre se deben encerrar entre paréntesis y se debería colocar a la derecha del operador relacional. Una subconsulta que utilice los valores >,<,>=,... tiene que devolver un único valor, **de otro modo ocurre un error**. Además tienen que tener el mismo tipo de columna para relacionar la subconsulta con la consulta que la utiliza (no puede ocurrir que la subconsulta tenga dos columnas y ese resultado se compare usando una sola columna en la consulta general).

(4.8.2) uso de subconsultas de múltiples filas

En el apartado anterior se comentaba que las subconsultas sólo pueden devolver una fila. Pero a veces se necesitan 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 **todos** los sueldos del departamento de ventas. Pero no podremos utilizar un operador de comparación directamente ya que esa subconsulta devuelve más de una fila. La solución a esto es utilizar instrucciones especiales entre el operador y la consulta, que permiten el uso de subconsultas de varias filas.

Esas instrucciones son:

Instrucción	Significado
ANY	Compara con cualquier registro de la subconsulta. 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);
```

La consulta anterior 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);
```

(4.8.3) consultas EXISTS

Este operador devuelve verdadero si la consulta que le sigue devuelve algún valor. Si no, devuelve falso. Se utiliza sobre todo en consultas correlacionadas. Ejemplo:

```
SELECT tipo,modelo, precio_venta
FROM piezas p
WHERE EXISTS (
    SELECT tipo,modelo FROM existencias
    WHERE tipo=p.tipo AND modelo=p.modelo);
```

Esta consulta devuelve las piezas que se encuentran en la tabla de existencias (es igual al ejemplo comentado en el apartado subconsultas sobre múltiples valores). La consulta contraria es :

```
SELECT tipo,modelo, precio_venta  
FROM piezas p  
WHERE NOT EXISTS (  
    SELECT tipo,modelo FROM existencias  
    WHERE tipo=p.tipo AND modelo=p.modelo);
```

Normalmente las consultas EXISTS se pueden realizar de alguna otra forma con los operadores ya comentados.

(4.9) combinaciones especiales

(4.9.1) 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).

(4.9.2) 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
```

(4.9.3) 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 */
```

(4.10) consultas avanzadas

(4.10.1) consultas con ROWNUM

La función **ROWNUM** devuelve el número de la fila de una consulta. Por ejemplo en:

```
SELECT ROWNUM, edad, nombre FROM clientes
```

Aparece el número de cada fila en la posición de la tabla. Esa función actualiza sus valores usando subconsultas de modo que la consulta:

```
SELECT ROWNUM AS ranking, edad, nombre FROM clientes
FROM (SELECT edad, nombre FROM clientes ORDER BY edad DESC)
```

Puesto que la consulta `SELECT edad, nombre FROM clientes ORDER BY edad DESC`, obtiene una lista de los clientes ordenada por edad, el `SELECT` superior obtendrá esa lista pero mostrando el orden de las filas en esa consulta. Eso permite hacer consultas el tipo top-n, (los n más...).

Por ejemplo para sacar el top-10 de la edad de los clientes (los 10 clientes más mayores):

```
SELECT ROWNUM AS ranking, edad, nombre FROM clientes
FROM (SELECT edad, nombre FROM clientes ORDER BY edad DESC)
WHERE ROWNUM<=10
```

(4.10.2) consultas sobre estructuras jerárquicas

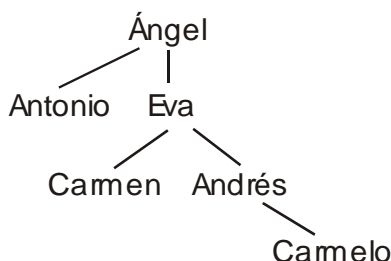
Imaginemos una tabla de empleados definida por un código de empleado, nombre del mismo y el código del jefe. Este último código está relacionado con el código de empleado que posee el jefe en cuestión. Así definido, una consulta que muestre el nombre de un empleado y el nombre de su jefe directo, sería:

```
SELECT e.nombre AS empleado, j.nombre AS jefe
FROM empleados e
JOIN empleados j ON (e.cod_jefe=j.cod_empleado);
```

Saldría por ejemplo:

EMPLEADO	JEFE
Antonio	Ángel
Ángel	
Eva	Ángel
Carmen	Eva
Andrés	Eva
Carmelo	Andrés

En el ejemplo se observa como un jefe puede tener otro jefe, generando una estructura jerárquica:



En este tipo de estructuras, a veces se requieren consultas que muestren todos los empleados de un jefe, mostrando los mandos intermedios. Se trata de una consulta que recorre ese árbol. Este tipo de consultas posee esta sintaxis:

```
SELECT [LEVEL,] listaDeColumnasYExpresiones
FROM tabla(s)...
[WHERE condiciones...]
[START WITH condiciones]
CONNECT BY [NOCYCLE] [PRIOR] expresion1=[PRIOR] expresion2
```

El apartado **CONNECT** permite indicar qué relación hay que seguir para recorrer el árbol. La palabra **PRIOR** indica hacia dónde se dirige el recorrido. Finalmente el apartado **START** indica la condición de inicio del recorrido (normalmente la condición

que permita buscar el nodo del árbol por el que comenzamos el recorrido, es decir sirve para indicar desde donde comenzamos. Ejemplo:

```
SELECT nombre FROM empleados
START WITH nombre='Andrés'
CONNECT BY PRIOR n_jefe=n_empleado;
```

Resultado:

NOMBRE
Andrés
Eva
Ángel

Sin embargo:

```
SELECT nombre FROM empleados
START WITH nombre='Andrés'
CONNECT BY n_jefe= PRIOR n_empleado;
```

Devuelve:

NOMBRE
Andrés
Carmelo

Si en lugar de Andrés en esa consulta buscáramos desde Ángel, saldría:

NOMBRE
Ángel
Antonio
Eva
Carmen
Andrés
Carmelo

El modificador LEVEL permite mostrar el nivel en el árbol jerárquico de cada elemento:

```
SELECT LEVEL, nombre FROM empleados
START WITH nombre='Ángel'
CONNECT BY n_jefe= PRIOR n_empleado;
```

Resultado:

LEVEL	NOMBRE
1	Ángel
2	Antonio
2	Eva
3	Carmen
3	Andrés
4	Carmelo

Para eliminar recorridos, se utilizan condiciones en WHERE o en el propio CONNECT. De modo que :

```
SELECT LEVEL, nombre FROM empleados  
WHERE nombre!='Eva'  
START WITH nombre='Ángel'  
CONNECT BY n_jefe= PRIOR n_employado;
```

En ese ejemplo, Eva no sale en los resultados. En este otro:

```
SELECT LEVEL, nombre FROM empleados  
START WITH nombre='Ángel'  
CONNECT BY n_jefe= PRIOR n_employado AND nombre!='Eva';
```

No sale ni Eva ni sus empleados (se corta la rama entera).

Cuando los datos forman un árbol jerárquico (como ocurre con los ejemplos) no suele haber ningún problema. Pero hay diseños en los que hay padres de hijos que a su vez pueden ser sus padres. Con los padres e hijos no ocurre esta situación evidentemente. Pero por ejemplo si tuviéramos un diseño de redes sociales donde se apunta el nombre del usuario y el nombre de sus amigos, entonces resulta que yo estaré apuntado como amigo de una persona que, a su vez, es mi amigo.

En ese caso resultaría un bucle infinito cuando se hace esta consulta:

```
SELECT amigo FROM contactos  
START WITH nombre='Ángel'  
CONNECT BY PRIOR amigo=nombre;  
-- Ocurre un bucle imposible de solucionar
```

Para esos casos disponemos de la cláusula **NOCYCLE** que controla los resultados repetidos y evita esos caminos. En cualquier caso consultas en datos no jerarquizados en forma de árbol sino en forma de grafo (como el comentado ejemplo de las redes sociales), pueden ocupar muchísimo tiempo a Oracle por lo que no es mala idea pensar en otra solución con ayuda de PL/SQL (lenguaje que se comenta en estos mismos apuntes).

```
SELECT DISTINCT amigo FROM contactos  
START WITH nombre='Ángel'  
CONNECT BY NOCYCLE PRIOR amigo=nombre; -- Ahora sí
```

Por cierto, el DISTINCT impide que aparezcan muchas veces las mismas personas. Ya que pueden ser amigos de muchos de mis amigos (e incluso de muchos de los amigos de mis amigos).

(4.11) consultas de agrupación avanzada

(4.11.1) ROLLUP

Esta expresión en una consulta de agrupación (GROUP BY) permite obtener los totales de la función utilizada para calcular en esa consulta. Ejemplo:

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

Esta consulta suma las cantidades de la tabla existencias por cada tipo y modelo distinto. Si añadimos:

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

Entonces nos añade un registro para cada tipo en el que aparece la suma del total para ese tipo. Al final mostrará un registro con el total absoluto. Es decir el resultado de esa consulta es:

TI	MODELO	SUM(CANTIDAD)
AR	6	10530
AR	9	4928
AR	15	5667
AR	20	43
AR		21168
BI	10	363
BI	38	1740
BI	57	1638
BI		3741
CL	12	7000
CL	15	3068
CL	18	6612
CL		16680
EM	21	257
EM	42	534
EM		791

TI	MODELO	SUM(CANTIDAD)
PU	5	12420
PU	9	7682
PU		20102
TO	6	464
TO	9	756
TO	10	987
TO	12	7740
TO	16	356
TO		10303
TU	6	277
TU	9	876
TU	10	1023
TU	12	234
TU	16	654
TU		3064
		75849

Se pueden unir varias columnas entre paréntesis para tratarlas como si fueran una unidad:

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

La diferencia respecto a la anterior es que el total mostado por ROLLUP se referirá al tipo y al modelo.

(4.11.2) CUBE

Es muy similar al anterior, sólo que este calcula todos los subtotales relativos a la consulta. Ejemplo:

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

Resulta:

TI	MODELO	SUM(CANTIDAD)
		75849
	5	12420
	6	11271

TI	MODELO	SUM(CANTIDAD)
	9	14242
	10	2373
	12	14974
	15	8735
	16	1010
	18	6612
	20	43
	21	257
	38	1740
	42	534
	57	1638
AR		21168
AR	6	10530
AR	9	4928
AR	15	5667
AR	20	43
BI		3741
BI	10	363
BI	38	1740
BI	57	1638
CL		16680
CL	12	7000
CL	15	3068
CL	18	6612
EM		791
EM	21	257
EM	42	534
PU		20102
PU	5	12420
PU	9	7682
TO		10303
TO	6	464
TO	9	756
TO	10	987
TO	12	7740
TO	16	356

TI	MODELO	SUM(CANTIDAD)
TU		3064
TU	6	277
TU	9	876
TU	10	1023
TU	12	234
TU	16	

Es decir, calcula totales por tipo, por modelo y el total absoluto.

(4.11.3) GROUPING

Se trata de una función que funciona con **ROLLUP** y **CUBE** y que recibe uno o más campos e indica si la fila muestra un subtotal referido a los campos en cuestión. Si la fila es un subtotal de esos campos pone 1, sino lo marca con 0. Ejemplo:

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

Sale:

TIPO	MODELO	SUM(CANTIDAD)	GROUPING(TIPO)	GROUPING(MODELO)
		75849	1	1
	5	12420	1	0
	6	11271	1	0
	9	14242	1	0
	10	2373	1	0
	12	14974	1	0
	15	8735	1	0
	16	1010	1	0
	18	6612	1	0
	20	43	1	0
	21	257	1	0
	38	1740	1	0
	42	534	1	0
	57	1638	1	0
AR		21168	0	1
AR	6	10530	0	0
AR	9	4928	0	0

TIPO	MODELO	SUM(CANTIDAD)	GROUPING(TIPO)	GROUPING(MODELO)
AR	15	5667	0	0
AR	20	43	0	0
BI		3741	0	1
BI	10	363	0	0
BI	38	1740	0	0
BI	57	1638	0	0
CL		16680	0	1
CL	12	7000	0	0
CL	15	3068	0	0
CL	18	6612	0	0
EM		791	0	1
EM	21	257	0	0
EM	42	534	0	0
PU		20102	0	1
PU	5	12420	0	0
PU	9	7682	0	0
TO		10303	0	1
TO	6	464	0	0
TO	9	756	0	0
TO	10	987	0	0
TO	12	7740	0	0
TO	16	356	0	0
TU		3064	0	1
TU	6	277	0	0
TU	9	876	0	0
TU	10	1023	0	0
TU	12	234	0	0
TU	16	654	0	0

Se utiliza sobre todo para preparar un consulta para la creación de informes.

(4.11.4) GROUPING SETS

Se trata de una mejora de Oracle 9i que permite realizar varias agrupaciones para la misma consulta. Sintaxis:

```
SELECT...  
...  
GROUP BY GROUPING SETS (listaDeCampos1) [,(lista2)...]
```

Las listas indican los campos por los que se realiza la agrupación. Ejemplo:

```
SELECT tipo, modelo, n_almacen, SUM(cantidad)  
FROM existencias  
GROUP BY GROUPING SETS ((tipo,modelo), (n_almacen));
```

De esa consulta se obtiene:

TI	MODELO	N_ALMACEN	SUM(CANTIDAD)
AR	6		10530
AR	9		4928
AR	15		5667
AR	20		43
BI	10		363
BI	38		1740
BI	57		1638
CL	12		7000
CL	15		3068
CL	18		6612
EM	21		257
EM	42		534
PU	5		12420
PU	9		7682
TO	6		464
TO	9		756
TO	10		987
TO	12		7740
TO	16		356
TU	6		277
TU	9		876
TU	10		1023

TI	MODELO	N_ALMACEN	SUM(CANTIDAD)
TU	12		234
TU	16		654
		1	30256
		2	40112
		3	5481

Se trata de dos consultas de totales unidades

(4.11.5) conjuntos de agrupaciones combinadas

Se pueden combinar agrupaciones de diversas formas creando consultas como:

```
SELECT tipo, modelo, n_almacen, SUM(cantidad)
FROM existencias
GROUP BY tipo, ROLLUP(modelo), CUBE(n_almacen)
```

Que mostraría un informe espectacular sobre las tablas anteriores. Así como:

```
SELECT tipo, modelo, n_almacen, SUM(cantidad)
FROM existencias
GROUP BY GROUPING SETS(tipo, modelo), GROUPING
SETS(tipo, n_almacen)
```

(4.12) DQL en instrucciones DML

A pesar del poco ilustrativo título de este apartado, la idea es sencilla. Se trata de cómo utilizar instrucciones SELECT dentro de las instrucciones DML (INSERT, DELETE o UPDATE), ello permite dar más potencia a dichas instrucciones.

(4.12.1) 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. Se hace mediante la instrucción INSERT y, en definitiva, permite copiar datos de una consulta a otra.

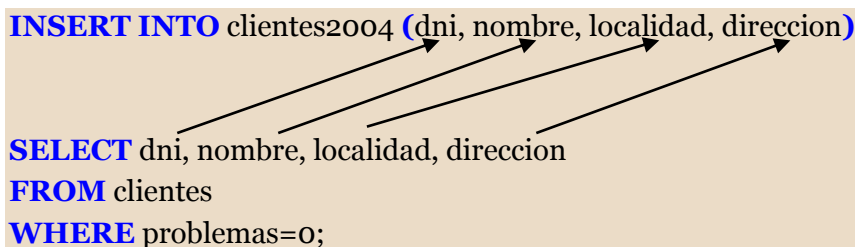
Ese relleno se basa en una consulta SELECT que poseerá los datos a añadir. Lógicamente el orden de esos campos debe de coincidir con la lista de campos indicada en la instrucción INSERT.

Sintaxis:

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

Ejemplo:

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



Lógicamente las columnas del SELECT se tienen que corresponder con las columnas a rellenar mediante INSERT (observar las flechas).

(4.12.2) subconsultas en la instrucción UPDATE

La instrucción **UPDATE** permite modificar filas. Es muy habitual el uso de la cláusula **WHERE** para indicar las filas que se modificarán. Esta cláusula se puede utilizar con las mismas posibilidades que en el caso del **SELECT**, por lo que es posible utilizar subconsultas. Por ejemplo:

```
UPDATE empleados  
SET sueldo=sueldo*1.10  
WHERE id_seccion =(SELECT id_seccion FROM secciones  
WHERE nom_seccion='Producción');
```

Esta instrucción aumenta un 10% el sueldo de los empleados de la sección llamada *Producción*. También podemos utilizar subconsultas en la cláusula SET de la instrucción UPDATE.

Ejemplo:

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

Esta instrucción 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.

(4.12.3) subconsultas en la instrucción DELETE

Al igual que en el caso de las instrucciones INSERT o SELECT, DELETE dispone de cláusula WHERE y en dicha cláusulas podemos utilizar subconsultas. Por ejemplo:

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

En este caso se trata de una subconsulta creada con el operador IN, se eliminarán los empleados cuyo identificador esté dentro de la tabla de *errores graves*.

(4.13) vistas

(4.13.1) 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.

(4.13.2) 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.n_provincia, p.nombre, p.superficie, l2.nombre,  
    c.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 vista pasa a usarse como una tabla normal*/
```

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 o utilizarse como subconsulta en los comandos UPDATE o DELETE.

(4.13.3) 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).

(4.13.4) borrar vistas

Se utiliza el comando DROP VIEW:

```
DROP VIEW nombreDeVista;
```


(5) PL/SQL

(5.1) introducción al SQL procedimental

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,...). La razón es que hay diversas acciones en la base de datos para las que **SQL** no es suficiente.

Por ello todas las bases de datos incorporan algún lenguaje de tipo procedimental (de tercera generación) que permite manipular de forma más avanzada los datos de la base de datos.

PL/SQL es el lenguaje procedimental que es implementado 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

(5.1.2) funciones que pueden realizar los programas PL/SQL

Las más destacadas son:

- Facilitar la realización de tareas administrativas sobre la base de datos (copia de valores antiguos, auditorías, control de usuarios,...)
- Validación y verificación avanzada de usuarios
- Consultas muy avanzadas
- Tareas imposibles de realizar con **SQL**

(5.1.3) conceptos básicos

bloque PL/SQL

Se trata de un trozo de código que puede ser interpretado por Oracle. Se encuentra inmerso dentro de las palabras **BEGIN** y **END**.

programa PL/SQL

Conjunto de bloques que realizan una determinada labor.

procedimiento

Programa PL/SQL almacenado en la base de datos y que puede ser ejecutado si se desea con solo saber su nombre (y teniendo permiso para su acceso).

función

Programa PL/SQL que a partir de unos datos de entrada obtiene un resultado (datos de salida). Una función puede ser utilizada desde cualquier otro programa PL/SQL e incluso desde una instrucción SQL.

trigger (*disparador*)

Programa PL/SQL que se ejecuta automáticamente cuando ocurre un determinado suceso a un objeto de la base de datos.

paquete

Colección de procedimientos y funciones agrupados dentro de la misma estructura. Similar a las bibliotecas y librerías de los lenguajes convencionales.

(5.2) escritura de PL/SQL

(5.2.1) 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 ]
BEGIN
  instrucciones ejecutables
[EXCEPTION
  instrucciones de manejo de errores ]
END;
```

A los bloques se les puede poner nombre usando (así se declara un procedimiento):

```
PROCEDURE nombre IS
bloque
```

para una función se hace:

```
FUNCTION nombre
RETURN tipoDedatos IS
bloque
```

Cuando un bloque no se declara como procedimiento o función, se trata de un bloque anónimo.

(5.2.2) escritura de instrucciones PL/SQL

normas básicas

La mayor parte de las normas de escritura en PL/SQL proceden de SQL, por ejemplo:

- Las palabras clave, nombres de tabla y columna, funciones,... no distinguen entre mayúsculas y minúsculas
- Todas las instrucciones finalizan con el signo del punto y coma (;), excepto las encabezan un bloque
- Los bloques comienzan con la palabra BEGIN y terminan con END
- Las instrucciones pueden ocupar varias líneas

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;
```

(5.3) variables

(5.3.1) 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 (para conseguirlo se necesita software auxiliar). La sintaxis de la declaración de variables es:

```
DECLARE
  identificador [CONSTANT] tipoDeDatos [:= valorInicial];
  [siguienteVariable...]
```

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 (**#**). 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 (<i>p</i>) y el número de decimales (<i>s</i>)
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**.

(5.3.2) 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.

(5.3.3) 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`

(5.3.4) 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 4^3 .

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,...), salvo en las instrucciones SQL permitidas.

A estas funciones se añaden diversas procedentes de paquetes de Oracle o creados por los programadores y las funciones GREATEST y LEAST

(5.3.5) paquetes estándar

Oracle incorpora una serie de paquetes para ser utilizados dentro del código PL/SQL. Es el caso del paquete DBMS_OUTPUT que sirve para utilizar funciones y procedimientos de escritura como PUT_LINE. Por ejemplo DBMS_OUTPUT.NEW_LINE() sirve para escribir una línea en blanco en el buffer de datos.

números aleatorios

El paquete DBMS_RANDOM contiene diversas funciones para utilizar número aleatorios. Quizá la más útil es la función DBMS_RANDOM.RANDOM que devuelve un número entero (positivo o negativo) aleatorio (y muy grande). Por ello si deseáramos un número aleatorio entre 1 y 10 se haría con la expresión:

```
MOD(ABS(DBMS_RANDOM.RANDOM),10)+1
```

Entre 20 y 50 sería:

```
MOD(ABS(DBMS_RANDOM.RANDOM),31)+20
```

(5.4) instrucciones SQL permitidas

(5.4.1) 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;
```

(5.4.2) 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.

(5.5) instrucciones de control de flujo

Son las instrucciones que permiten ejecutar un bloque de instrucciones u otro dependiendo de una condición. También permiten repetir un bloque de instrucciones hasta cumplirse la condición (es lo que se conoce como bucles).

La mayoría de estructuras de control PL/SQL son las mismas que las de los lenguajes tradicionales como C o Pascal. En concreto PL/SQL se basa en el lenguaje **Ada**.

(5.5.1) 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

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;
```


(5.5.2) 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;
```

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.

también se pueden escribir 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;
```

(5.5.3) 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,...)

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 utiliza 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.

(5.6) cursores

(5.6.1) 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

(5.6.2) 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**

(5.6.3) 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;
```

(5.6.4) apertura de cursores

```
OPEN cursor;
```

Esta sentencia abre el cursor, lo que significa:

- (4) Reservar memoria suficiente para el cursor
- (5) Ejecutar la sentencia **SELECT** a la que se refiere el cursor
- (6) Colocar el puntero de recorrido de registros en la primera fila

Si la sentencia **SELECT** del cursor no devuelve registros, Oracle no devolverá una excepción. Hasta intentar leer no sabremos si hay resultados o no.

(5.6.5) 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. La lista de variables tiene tener el mismo tipo y número que las columnas representadas en el cursor (por supuesto el orden de las variables se tiene que corresponder con la lista de columnas). Tras esta instrucción el puntero de registros avanza a la siguiente fila (si la hay).

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 a fin de poder leer todas las filas de un cursor:

```
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;
```

(5.6.6) cerrar el cursor

```
CLOSE cursor;
```

Al cerrar el cursor se libera la memoria que ocupa y se impide su procesamiento (no se podría seguir leyendo filas). Tras cerrar el cursor se podría abrir de nuevo.

(5.6.7) 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. Estos atributos facilitan la manipulación del cursor. Se utilizan indicando el nombre del

cursor, el símbolo % e inmediatamente el nombre del atributo a valorar (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.

(5.6.8) variables de registro

introducción

Los registros son una estructura estática de datos presente en casi todos los lenguajes clásicos (**record** en Pascal o **struct** en C). 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. Gracias a esta interpretación, los registros facilitan la manipulación de los cursores ya que podemos entender que un cursor es un conjunto de registros (cada registro sería una fila del cursor).

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* es un registro que constará de los mismos campos y tipos que las columnas de la tabla *personas*.

(5.6.9) 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 183):

```
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** implícito 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 en cada vuelta del bucle almacena el contenido de cada fila en una variable de registro. La variable de registro utilizada en el bucle **FOR** no se debe declarar en la zona **DECLARE**; se crea al inicio del bucle y se elimina cuando éste finaliza.
- (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 comentado en los apartados anteriores):

```
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.

(5.6.10) cursores avanzados

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 ;
```

(5.7) excepciones

(5.7.1) 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.

(5.7.2) 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.

(5.7.3) 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

Nombre de excepción	Número de error	Ocurre cuando..
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 <i>varray</i> 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 <i>varray</i> 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 <i>varray</i> 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;
```

(5.7.4) 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;
```

(5.7.5) 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;
```

(5.7.6) 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;
```

- (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**

(5.8) procedimientos

(5.8.1) 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.

(5.8.2) estructura de un procedimiento

La sintaxis es:

```
CREATE [OR REPLACE] PROCEDURE nombreProcedimiento  
  [(parámetro1 [modelo] tipoDatos  
  [,parámetro2 [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 con el que se crea ahora. Los parámetros son la lista de variables que necesita el procedimiento para realizar su tarea.

Al declarar cada parámetro se indica el tipo de los mismos, pero no su tamaño; es decir sería **VARCHAR2** y no **VARCHAR2(50)**.

El apartado opcional **modelo**, se elige si el parámetro es de tipo **IN**, **OUT** o **IN OUT** (se explica más adelante).

No se utiliza la palabra **DECLARE** para indicar el inicio de las declaraciones. No obstante la sección de declaraciones figura tras las palabras **IS** o **AS** (es decir justo antes del **BEGIN** es donde debemos declarar las variables).

(5.8.3) 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 (como **toad** por ejemplo). El resultado es el llamado **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 incorpore una llamada al procedimiento).

(5.8.4) 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 llamada al procedimiento *actualizar()* se puede hacer simplemente escribiendo *actualizar*, sin 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 IN OUT.

(5.8.5) borrar procedimientos

El comando **DROP PROCEDURE** seguido del nombre del procedimiento que se elimina es el encargado de realizar esta tarea.

(5.9) funciones

(5.9.1) 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.

(5.9.2) sintaxis

```
CREATE [OR REPLACE] FUNCTION nombreFunción  
[(parámetro1 [modelo] tipoDatos  
[,parámetro2 [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.

(5.9.3) 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.

(5.9.4) 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.

(5.9.5) eliminar funciones

Sintaxis:

```
DROP FUNCTION nombreFunción;
```

(5.9.6) 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;
```

(5.9.7) mostrar procedimientos almacenados

La vista `USER_PROCEDURES`, contiene una fila por cada procedimiento o función que tenga almacenado el usuario actual.

(5.10) paquetes

(5.10.1) 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, funciones y variables **públicos** del paquete (los que se podrán invocar desde 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

(5.10.2) 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 paquete1 IS  
    v_cont NUMBER := 0;  
    PROCEDURE reset_cont(v_nuevo_cont NUMBER);  
    FUNCTION devolver_cont  
        RETURN NUMBER;  
END paquete1;
```

De las funciones hay que indicar sus parámetros y el tipo de datos que devuelve.

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 paquete1 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 paquete1;
```

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.

Eso mismo ocurre con las variables, las variables declaradas en la especificación de un paquete, son accesibles desde fuera del paquete (habría que invocarlas escribiendo `paquete.variable`)

(5.11) triggers

(5.11.1) 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 hecho 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.

(5.11.2) 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

- **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.

(5.11.3) 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 asocian el trigger al uso de una instrucción DML. En el caso de la instrucción **UPDATE**, el apartado **OF** hace que el trigger se ejecute sólo cuando se modifique la columna indicada (o columnas si se utiliza una lista de columnas 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]  
[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 el trigger sea de fila, es decir que se repita su ejecución por cada fila afectada en la tabla 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 las filas que cumplan dicha condición.

El apartado **REFERENCING** es el que permite indicar un nombre para los valores antiguos y otro para los nuevos.

(5.11.4) referencias **NEW** y **OLD**

Cuando se ejecutan instrucciones **UPDATE**, hay que tener en cuenta que se modifican valores antiguos (**OLD**) para cambiarlos 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 palabras **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.

(5.11.5) IF INSERTING, IF UPDATING e IF DELETING

Son palabras que se utilizan para determinar la instrucción DML 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 campo1 ON tabla  
FOR EACH ROW  
BEGIN  
    IF DELETING THEN  
        instrucciones que se ejecutan si el trigger saltó por borrar filas  
    ELSIF INSERTING THEN  
        instrucciones que se ejecutan si el trigger saltó por insertar filas  
    ELSE  
        instrucciones que se ejecutan si el trigger saltó por modificar filas  
    END IF  
END;
```

(5.11.6) 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
    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 código 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.

(5.11.7) administración de triggers

eliminar triggers

Sintaxis:

```
DROP TRIGGER nombreTrigger;
```

desactivar un trigger

```
ALTER TRIGGER nombreTrigger DISABLE;
```

activar un trigger

```
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;
```

(5.11.8) 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

(5.11.9) 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:

- (4) Primero disparadores de tipo **BEFORE** de tipo instrucción
- (5) Disparadores de tipo **BEFORE** por cada fila
- (6) Se ejecuta la propia orden que desencadenó al trigger.
- (7) Disparadores de tipo **AFTER** con nivel de fila.
- (8) Disparadores de tipo **AFTER** con nivel de instrucción.

(5.11.10) 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 trigger es la tabla a la que se refiere el disparador en la cláusula **ON**.

Un disparador no puede leer o modificar una tabla mutante (una tabla que está cambiando) ni tampoco leer o modificar una columna que tiene restricciones asociadas a una tabla mutante (sí se podrían 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, los triggers de tabla sí pueden. Por ello la solución al problema suele ser combinar triggers de tabla con triggers de fila (conociendo el orden de ejecución de los triggers).

El truco consiste en que los triggers de tabla 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 o si el paquete es global).

bibliografía

Introducción a los Sistemas de Bases de Datos (7ª Edición)

C. J. Date

Prentice Hall, 2001

Fundamentos y modelos de Bases de Datos

Adoración de Miguel

Mario Piattini

ra-ma, 1999

Database design, know it all

Varios autores

Elsevier, 2009

Database modelling and design

Toby Teory

Sam Lightstone

Tom Nadeau

Morgan Kaufman, 2006

Database modelling and design

Toby Teory

Morgan Kaufman, 1999

Beginning Database Design: From Novice to Professional

Clare Churchel

Apress, 2007

Beginning Database Design

Gavin Powell

Wrox, 2006

Diseño conceptual de Bases de Datos

Carlo Batini

Stefano Ceri

Shamkant Navathe

Addison Wesley, 1994

El modelo relacional de datos

Covadonga Fernández

Ediciones Díaz de Santos, 1987

Sistemas Gestores de Bases de Datos

Gregorio Cabrera Sánchez

Paraninfo, 2001

Sistemas Gestores de Bases de Datos

María Jesús Ramos

Alicia Ramos

Fernando Montero

Mac Graw Hill, 2006

Oracle 9i

Michael Abbey

Mike Corey

Ian Abramson

Oracle Press, Mac Graw Hill, 2002

Oracle 9i, Manual de Referencia

Kevin Loney

George Koch

Oracle Press, Mac Graw Hill, 2003

Oracle 9i, Programación PL/SQL

Scott Urman

Oracle Press, Mac Graw Hill, 2002

Oracle Database 11g. SQL

Jason Price

Oracle Press, Mac Graw Hill, 2009

OCA. Oracle 11g. Oracle SQL Fundamentals. Exam Guide

John Watson

Rupert Ramklass

Oracle Press, Mac Graw Hill, 2008

Oracle 11g. The Complete reference

Kevin Loney

Scott Gosset

Sreekanth Chintala

Oracle Press, Mac Graw Hill, 2009

Oracle 11g RMAN. Backup and Recovery

Robert G Freeman

Mathew Hart

Oracle Press, Mac Graw Hill, 2011

índice de ilustraciones

Ilustración 1, Ejemplo de fichero de acceso directo.....	12
Ilustración 2, ejemplo de fichero secuencial encadenado. Los punteros le recorren por la clave	13
Ilustración 3, ejemplo de fichero secuencial indexado	14
Ilustración 4, Ejemplo de archivo secuencial indexado y encadenado.....	15
Ilustración 5, Sistemas de Información orientados al proceso	17
Ilustración 6, Sistemas de información orientados a datos	18
Ilustración 7, Esquema del funcionamiento y utilidad de un sistema gestor de bases de datos.....	19
Ilustración 8, Modelo de referencia de las facilidades de usuario	23
Ilustración 9, Esquema del funcionamiento de un SGBD	24
Ilustración 10, Relación entre los organismos de estandarización	27
Ilustración 11, Niveles en el modelo ANSI	28
Ilustración 12, Arquitectura ANSI, explicación clásica del funcionamiento.....	29
Ilustración 13, Modelos de datos utilizados en el desarrollo de una BD.....	31
Ilustración 14, Ejemplo de esquema jerárquico.....	33
Ilustración 15, ejemplo de diagrama de estructura de datos Codasyl	33
Ilustración 16, Ejemplos de entidad y conjunto de entidad	36
Ilustración 17, Representación de la entidad persona	36
Ilustración 18, Entidad débil	37
Ilustración 19, ejemplo de relación.....	37
Ilustración 20, Tipos de relaciones	38
Ilustración 21, Cardinalidades.	39
Ilustración 22, Ejemplo de rol.....	39
Ilustración 23, Atributos.....	40
Ilustración 24, Relación ISA.	42
Ilustración 25, Ejemplo de relación ISA	42
Ilustración 26, Relación de herencia, la clave de la superentidad es clave de las subentidades.	43
Ilustración 27, La clave de la superentidad no es clave de las subentidades.....	43
Ilustración 28, Relación ISA con obligatoriedad.....	44
Ilustración 29, Tipos de relaciones ISA	46
Ilustración 30, Relación candidata a entidad débil.....	46
Ilustración 31, Entidad débil relacionada con su entidad fuerte	47
Ilustración 32, Ejemplo de clave secundaria.....	58
Ilustración 33, Transformación de una entidad fuerte al esquema relacional	60

Ilustración 34, Transformación de una relación n a n	61
Ilustración 35, Transformación en el modelo relacional de una entidad n -aria.....	61
Ilustración 36, Transformación de una relación uno a varios.....	62
Ilustración 37, Las dos soluciones válidas a la cardinalidad 1 a.....	63
Ilustración 38, Solución a la relación 0 a 1.....	63
Ilustración 39, Paso a relacional de relaciones cero a cero.....	64
Ilustración 40, Transformación de relaciones recursivas en el modelo relacional.....	64
Ilustración 41, Abstracción para facilitar la conversión de relaciones recursivas.....	65
Ilustración 42, transformación de entidades débiles en el modelo relacional.....	65
Ilustración 43, Proceso de transformación de relaciones ISA.....	66
Ilustración 44, Diagrama estilo <i>pata de gallo</i>	68
Ilustración 45, Esquema relacional completo de la base de datos de un Video Club.....	69
Ilustración 46, Esquema relacional del almacén según el programa Visio de Microsoft.....	70