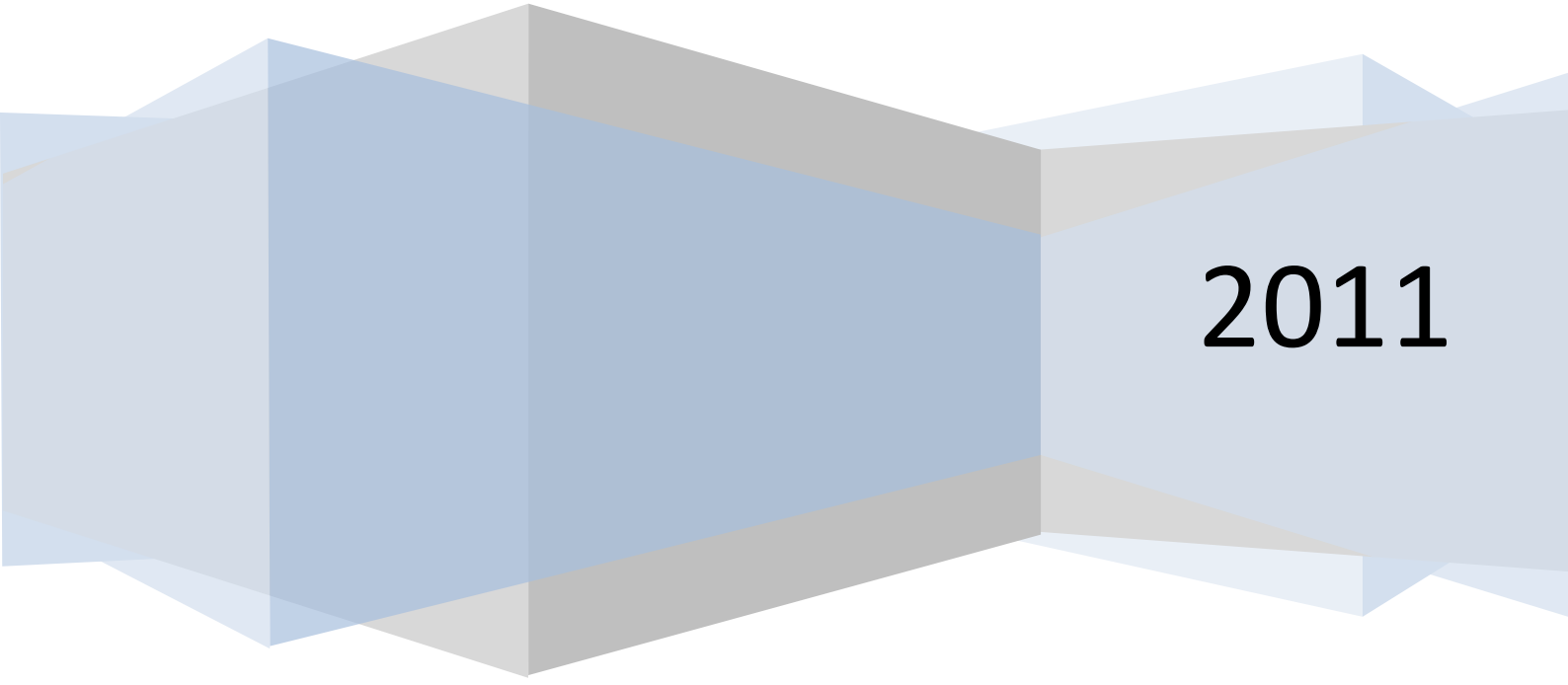
TEMA 1

BASES DE DATOS

Desarrollo de Aplicaciones Web

José Luis Comesaña



2011

TEMA 1

ÍNDICE

1.- Introducción. 1
2.- Los ficheros de información. 2
2.1.- ¿Qué es un fichero? 2
2.2.- Tipos de ficheros. 3
2.3.- Los soportes de información. 3
2.4.- Métodos de acceso. 4
2.5.- Ficheros secuenciales. 4
2.6.- Ficheros de acceso directo. 5
2.7.- Ficheros indexados. 6
2.8.- Otros (secuenciales indexados, hash.). 7
a. Ficheros Secuenciales Indexados 7
b. Ficheros de Acceso Calculado o Hash 7
➔ Módulo 8
➔ Extracción 8
2.9.- Parámetros de utilización. 8
3.- Bases de datos. 9
3.1.- Conceptos. 9
Base de datos 9
3.2.- Usos. 10
¿Quién utiliza las bases de datos? 10
¿Para qué se utilizan las bases de datos? 11
3.3.- Ubicación de la información. 11
4.- Modelos de bases de datos. 13
4.1.- Modelo jerárquico. 13
4.2.- Modelo en red. 14
4.3- Modelo relacional. 14
4.4.- Modelo orientado a objetos. 15
4.5.- Otros modelos. 16
Modelo Objeto-Relacional 16
Modelo de bases de datos deductivas 16
Bases de datos multidimensionales 25
Bases de datos transaccionales 25
5.- Tipos de bases de datos. 26
6 - Sistemas gestores de bases de datos 30
6.1.- Funciones. 31
6.2.- Componentes. 32
6.3.- Arquitectura. 33
6.4.- Tipos. 34
7.- SGBD comerciales. 35
8.- SGBD libres. 37
9.- Bases de datos centralizadas. 38
10.- Bases de datos distribuidas. 39
10.1.- Fragmentación. 40
11.- Primeros pasos en Oracle Database 10g Express Edition. 42
¿Qué es Oracle Database 10g Express Edition? 42
¿Por dónde empezamos? 42
¿Cómo se realiza la instalación? 42
Instalación de Oracle Database 10g Express Edition bajo Windows 7 42
Instalación de Oracle Database 10g Express Edition bajo Ubuntu Linux. 43

Gestión básica de datos en Oracle Database 10g Express Edition 43
Administración simple de usuarios en Oracle Database 10g Express Edition 43

Almacenamiento de la información

Caso práctico

__Ada__ sabe bien que BK Programación deberá hacer frente a retos importantes que requerirán del dominio adecuado de múltiples disciplinas. Tiene claro que el desarrollo de sus proyectos ha de estar apoyado sobre unas bases firmes, y una de ellas será la gestión adecuada de los datos.
Considera que __Juan__ y __María__ deben conocer la evolución que han experimentado las técnicas de almacenamiento de información, destacando que el dominio de las bases de datos es fundamental para garantizar un funcionamiento óptimo de las aplicaciones que BK Programación va a tener que desarrollar.

1.- Introducción.

¿Te has preguntado alguna vez dónde y de qué manera se almacenan y gestionan los datos que utilizamos diariamente? Si pensamos en cualquier acción de nuestra vida cotidiana, o si analizamos la mayoría de los ámbitos de actividad, nos encontramos que la utilización de las bases de datos está ampliamente extendida. Éstas, y los datos contenidos en ellas, serán imprescindibles para llevar a cabo multitud de acciones.

¿Crees que no es para tanto? Piensa en las siguientes situaciones:

- ✓ Cuando seleccionamos nuestro canal favorito en la TDT.
- ✓ Al utilizar la agenda del móvil para realizar una llamada telefónica.
- ✓ Cuando operamos en el cajero automático.
- ✓ Al solicitar un certificado en un organismo público.
- ✓ Cuando acudimos a la consulta del médico.
- ✓ Al inscribirnos en un curso, plataforma OnLine, etc.
- ✓ Si utilizas un GPS.
- ✓ Cuando reservamos unas localidades para un evento deportivo o espectáculo.
- ✓ Si consumimos ocio digital.
- ✓ Cuando consultamos cualquier información en Internet. (Bibliotecas, enciclopedias, museos, etc.)
- ✓ Al registrarte en una página de juegos OnLine, redes sociales o foros.
- ✓ Incluso, si tienes coche, puede ser que éste incorpore alguna base de datos.

Suponemos que no es necesario que continuemos más para darnos cuenta de que casi todo lo que nos rodea, en alguna medida, está relacionado con los datos, su almacenamiento y su gestión. El gran volumen de datos que actualmente manejamos y sus innumerables posibilidades requieren de la existencia de técnicos perfectamente formados y capaces de trabajar con ellos.

Este módulo profesional se centra en el estudio de las **Bases de Datos** y su uso en el desarrollo de aplicaciones. En esta primera unidad comenzaremos conociendo los primeros sistemas basados en ficheros para el almacenamiento y gestión de la información. Seguidamente, se desarrollarán los conceptos y definiciones básicas relacionadas con las bases de datos, posteriormente analizaremos sus modelos y tipos, un poco más adelante, podremos conocer las características y capacidades de los sistemas gestores de bases de datos y finalmente, identificaremos las herramientas reales con las que llevar a cabo la gestión dichas bases.

2.- Los ficheros de información.

Caso práctico

Juan le cuenta a María que hace poco visitó un museo en el que había una exposición sobre historia de la informática y que pudo ver soportes antiguos para almacenamiento de información: tarjetas perforadas, cintas magnéticas, tambores magnéticos, discos de diferentes tamaños y otros dispositivos de la época.

-Todo ha evolucionado muchísimo, la cantidad de datos y archivos que hoy podemos transportar en los modernos sistemas de almacenamiento y la velocidad a la que podemos acceder a ellos es sorprendente -comenta María.

Ada, mientras, prepara un DVD para realizar una copia de seguridad de los archivos de su portátil, destaca que gracias a las mejoras en el modo de organización de ficheros y soportes de información, se ha abierto un sin fin de posibilidades para la aplicación de las TIC en cualquier ámbito.

2.1.- ¿Qué es un fichero?

En la década de los setenta, los procesos básicos que se llevaban a cabo en una empresa se centraban en cuestiones relacionadas con contabilidad y facturación. Las necesidades de almacenamiento y gestión de información podían satisfacerse utilizando un número relativamente reducido de archivos en papel agrupados y ordenados, los típicos ficheros clásicos.

Al llevar a cabo una primera informatización, se pasó de tener los datos en formato papel a poder acceder a ellos de manera mucho más rápida a través del ordenador. En ese momento, 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,...

La información debía ser trasladada desde el papel al formato digital y por lo general, era necesario almacenarla para su posterior recuperación, consulta y procesamiento. De este modo, para llevar a cabo un tratamiento eficiente de ésta era necesario establecer métodos adecuados para su almacenamiento. El elemento que permitió llevar a cabo el almacenamiento de datos de forma permanente en dispositivos de memoria masiva fue el fichero o archivo.

Fichero o archivo: conjunto de información relacionada, tratada como un todo y organizada de forma estructurada. Es una secuencia de dígitos binarios que organiza información relacionada con un mismo aspecto.

Los ficheros están formados por registros lógicos que contienen datos relativos a un mismo elemento u objeto (por ejemplo, los datos de usuarios de una plataforma educativa). A su vez, los registros están divididos en campos que contienen cada una de las informaciones elementales que forman un registro (por ejemplo, el nombre del usuario o su dirección de correo electrónico).

Hemos de resaltar que los datos están almacenados de tal forma que se puedan añadir, suprimir, actualizar o consultar individualmente en cualquier momento.

Como los ficheros suelen ser muy voluminosos, solo se pueden llevar a la memoria principal partes de ellos para poder procesarlos. La cantidad de información que es transferida entre el soporte en el que se almacena el fichero, y la memoria principal del ordenador, en una sola operación de lectura/grabación, recibe el nombre de registro físico o bloque.

Normalmente en cada operación de lectura/grabación se transfieren varios registros del fichero, es decir un bloque suele contener varios registros lógicos. Al número de registros que entran en un bloque se le conoce con el nombre de factor de blocaje, y a esta operación de agrupar varios registros en un bloque se le llama bloqueo de registros.

2.2.- Tipos de ficheros.

Según la función que vaya a desempeñar los ficheros, éstos pueden ser clasificados de varias maneras. En la siguiente imagen puedes observar una posible clasificación.

a. **Ficheros permanentes:** contienen información relevante para una aplicación. Es decir, los datos necesarios para el funcionamiento de ésta. Tienen un periodo de permanencia en el sistema amplio. Estos se subdividen en:
   - ✓ **Ficheros maestros:** contienen el estado actual de los datos que pueden modificarse desde la aplicación. Es la parte central de la aplicación, su núcleo. Podría ser un archivo con los datos de los usuarios de una plataforma educativa.
   - ✓ **Ficheros constantes:** son aquellos que incluyen datos fijos para la aplicación. No suelen ser modificados y se accede a ellos para realización de consultas. Podría ser un archivo con códigos postales.
   - ✓ **Ficheros históricos:** contienen datos que fueron considerados como actuales en un periodo o situación anterior. Se utilizan para la reconstrucción de situaciones. Podría ser un archivo con los usuarios que han sido dados de baja en la plataforma educativa.

b. **Ficheros temporales:** Se utilizan para almacenar información útil para una parte de la aplicación, no para toda ella. Son generados a partir de datos de ficheros permanentes. Tienen un corto periodo de existencia. Estos se subdividen en:
   - ✓ **Ficheros intermedios:** almacenan resultados de una aplicación que serán utilizados por otra.
   - ✓ **Ficheros de maniobras:** almacenan datos de una aplicación que no pueden ser mantenidos en memoria principal por falta de espacio.
   - ✓ **Ficheros de resultados:** almacenan datos que van a ser transferidos a un dispositivo de salida.

Supongamos una aplicación informática para gestionar una biblioteca, existirá un fichero con el catálogo de libros disponibles, otro con las editoriales, otro con información sobre libros que se han quedado obsoletos, etc. ¿A cuál de los siguientes tipos correspondería el fichero que almacena las editoriales?

- ☐ Fichero maestro.
- ◉ **Fichero constante.**
- ☐ Fichero intermedio.

2.3.- Los soportes de información.

Los ficheros se almacenan en soportes de información manejados por dispositivos periféricos del ordenador, que permiten leer y grabar datos en el soporte. Los soportes más utilizados para almacenar los ficheros son las cintas magnéticas y los discos (magnéticos, ópticos, o magneto-ópticos). Dentro de estos dos tipos de soporte existen en el mercado una gran variedad de modelos.
Inicialmente, los primeros sistemas de almacenamiento físico eran tambores de cinta magnética. Tenían unas dimensiones parecidas a los discos de vinilo. Estos tambores funcionaban de manera similar a los antiguos casetes, pero sus mayores dimensiones les permitían almacenar gran cantidad de datos en formato digital, es decir en ceros y unos, en orden secuencial.

Posteriormente, los sistemas de almacenamiento de información comenzaron a cambiar de la mano de los avances en el hardware, en concreto con la aparición del disquete y del disco duro. Eran

dispositivos de acceso aleatorio, no siendo necesario en ellos pasar por todos los datos desde el inicio hasta la zona donde se encuentra la información que nos interesa.
Por tanto, se distinguen dos tipos de soportes para el almacenamiento de datos:

- ✓ **Soportes de Acceso Directo a los datos** (Por ejemplo: discos). Son los más empleados y el acceso a los datos puede hacerse de forma directa, pudiendo colocarnos en la posición que nos interesa y leer a partir de ella.
- ✓ **Soportes de Acceso Secuencial** (Por ejemplo: cintas magnéticas). Se suelen usar en copias de seguridad y si deseamos leer un dato que está en la mitad de la cinta, tendremos que leer todo lo que hay hasta llegar a esa posición.

Conoce más sobre las características de cintas y discos a través de los enlaces que te proponemos:

http://es.wikipedia.org/wiki/Cinta_magn%C3%A9tica_de_almacenamiento_de_datos
http://www.infodata.es/tour/disco-duro.html
http://es.wikipedia.org/wiki/Disco_%C3%B3ptico
http://www.consumer.es/web/es/tecnologia/hardware/2006/01/26/148862.php
http://www.infodata.es/tour/magneto-opticos.html

2.4.- Métodos de acceso.

A medida que la tecnología ha ido evolucionando, atendiendo principalmente a los avances hardware, el acceso a la información contenida en los diferentes tipos de ficheros ha variado mucho.
Los objetivos fundamentales de estas modificaciones pueden resumirse en los siguientes puntos:

- ✓ Proporcionar un acceso rápido a los registros.
- ✓ Conseguir economizar el almacenamiento.
- ✓ Facilitar la actualización de los registros.
- ✓ Permitir que la estructura refleje la organización real de la información.

Las distintas formas de organizar un fichero en un soporte de memoria o, lo que se conoce también por métodos de acceso a los ficheros se detallan en el siguiente gráfico.

Las organizaciones secuenciales, de acceso aleatorio o directo y de acceso indexado son las más comunes. En los siguientes epígrafes se detallarán las características de cada uno de los métodos de acceso a los ficheros.

Relaciona los diferentes métodos de acceso a los ficheros.

Método de acceso	Relación	Tipo de acceso
Encadenado.	**2**	1. Directo.
Indexado.	**3**	2. Secuencial.
Calculado o Hash.	**1**	3. Directo o secuencial.

Como ves el método de acceso Indexado existe en los dos tipos de acceso: directo y secuencial.

2.5.- Ficheros secuenciales.

Un fichero con organización secuencial se caracteriza porque sus registros están almacenados de forma contigua, de manera, que la única forma de acceder a él, es leyendo un registro tras otro desde el principio hasta el final. En los ficheros secuenciales suele haber una marca indicativa del fin

del fichero, que suele denominarse **EOF** (End of File). Para detectar el final del fichero sólo es necesario encontrar la marca EOF.

Este tipo de ficheros pueden utilizar dispositivos o soportes no direccionables o de acceso secuencial, como son las cintas magnéticas de almacenamiento de datos. También se utiliza en los CD de audio y los DVD de vídeo, en los que la música o las imágenes se almacenan a lo largo de una espiral continua.

Los registros almacenados se identifican por medio de una información ubicada en uno de sus campos, a este campo se le denomina **clave o llave**. Si se ordena un archivo secuencial por su clave, es más rápido realizar cualquier operación de lectura o escritura.

Otras características relevantes de los ficheros secuenciales son:

- ✓ La lectura siempre se realiza hacia delante.
- ✓ Son ficheros monousuario, no permiten el acceso simultáneo de varios usuarios.
- ✓ Tienen una estructura rígida de campos. Todos los registros deben aparecer en orden, es decir, la posición de los campos de cada registro siempre ha de ser la misma.
- ✓ El modo de apertura del fichero, condiciona la lectura o escritura.
- ✓ Aprovechan al máximo el soporte de almacenamiento, al no dejar huecos vacíos.
- ✓ Se pueden grabar en cualquier tipo de soporte, tanto en secuenciales como direccionables.
- ✓ Todos los lenguajes de programación disponen de instrucciones para trabajar con este tipo de ficheros.
- ✓ No se pueden insertar registros entre los que ya están grabados.

En el siguiente gráfico se observa la estructura de un fichero secuencial.

2.6.- Ficheros de acceso directo.

En este tipo de ficheros se puede acceder a un registro indicando la posición relativa del mismo dentro del archivo o, más comúnmente, a través de una clave que forma parte del registro como un campo más. Estos archivos deben almacenarse en dispositivos de memoria masiva de acceso directo, como son los discos magnéticos.

Campo clave: campo que permite identificar y localizar un registro de manera ágil y organizada.
Cada uno de los registros se guarda en una posición física, que dependerá del espacio disponible en memoria masiva, de ahí que la distribución de los registros sea aleatoria dentro del soporte de almacenamiento. Para acceder a la posición física de un registro se utiliza una dirección o índice, no siendo necesario recorrer todo el fichero para encontrar un determinado registro.
A través de una transformación específica aplicada a la clave, se obtendrá la dirección física en la que se encuentra el registro. Según la forma de realizar esta transformación, existen diferentes modos de acceso:

En el acceso directo la clave coincide con la dirección, debiendo ser numérica y comprendida dentro del rango de valores de las direcciones. Es el método más rápido.

La medida básica de posicionamiento del puntero en el fichero es el byte, dependiendo del tipo de codificación de caracteres que empleemos (Unicode, ANSI) se utilizarán 1 o 2 bytes por carácter respectivamente. Teniendo esto en cuenta, el puntero avanzará de

uno en uno o de dos en dos bytes para poder leer o escribir cada carácter.

Otras características fundamentales de los ficheros de acceso directo o aleatorio son:
- ✓ Posicionamiento inmediato.
- ✓ Registros de longitud fija.
- ✓ Apertura del fichero en modo mixto, para lectura y escritura.
- ✓ Permiten múltiples usuarios utilizándolos.
- ✓ Los registros se borran colocando un cero en la posición que ocupan.
- ✓ Permiten la utilización de algoritmos de compactación de huecos.
- ✓ Los archivos se crean con un tamaño definido, es decir, con un máximo de registros establecido durante la creación.
- ✓ Esta organización sólo es posible en soportes direccionables.
- ✓ Se usan cuando el acceso a los datos de un registro se hace siempre empleando la misma clave y la velocidad de acceso a un registro es lo que más nos importa.
- ✓ Permiten la actualización de los registros en el mismo fichero, sin necesidad de copiar el fichero.
- ✓ Permiten realizar procesos de actualización en tiempo real.

En los ficheros de acceso directo los registros siempre se encuentran en posiciones contiguas dentro del soporte de almacenamiento.

Verdadero. ☐ Falso ◉

2.7.- Ficheros indexados.

Se basan en la utilización de **índices**, que permiten el acceso a un registro del fichero de forma directa, sin tener que leer los anteriores. Estos índices son similares a los de los libros. Si nos interesa leer un capítulo concreto podemos recurrir al índice que nos dice en que página comienza, y abrimos el libro por esa página, sin tener que mirar en todas las páginas anteriores para localizarlo.

Por tanto, existirá una **zona de registros** en la que se encuentran los datos del archivo y una **zona de índices,** que contiene una tabla con las claves de los registros y las posiciones donde se encuentran los mismos. La tabla de índices estará ordenada por el campo clave.

La tabla de índices será cargada en memoria principal para realizar en ella la búsqueda de la fila correspondiente a la clave del registro a encontrar, obteniéndose así la dirección donde se encuentra el registro. Una vez localizada la dirección, sólo hay que acceder a la zona de registros en el soporte de almacenamiento y posicionarnos en la dirección indicada. Puesto que la tabla debe prever la inclusión de todas las direcciones posibles del archivo, su principal inconveniente resulta determinar su tamaño y mantenerla ordenada por los valores de la clave.

Las características más relevantes de un fichero indexado, son las siguientes:
- ✓ El diseño del registro tiene que tener un campo, o combinación de campos, que permita identificar cada registro de forma única, es decir, que no pueda haber dos registros que tengan la misma información en él. A este campo se le llama **campo clave** y es el que va a servir de índice. Un mismo fichero puede tener mas de un campo clave, pero al menos uno de ellos no admitirá valores duplicados y se le llama clave primaria. A las restantes se les llama claves alternativas.
- ✓ Permiten utilizar el modo de **acceso secuencial** y el modo de **acceso directo** para leer la información guardada en sus registros.
- ✓ Para acceder a este tipo de ficheros utilizando el modo de acceso directo se hace conociendo el contenido del campo clave del registro que queremos localizar. Con esa información el sistema operativo puede consultar el índice y conocer la posición del registro dentro del fichero.

- ✓ Para acceder a este tipo de ficheros utilizando el modo de acceso secuencial los registros son leídos ordenados por el contenido del campo clave, independientemente del orden en que se fueron grabando (el orden lógico no es igual al orden físico), debido a que el acceso a los datos se hace a través del índice, que para hacer más fácil la búsqueda de los registros, permanece siempre ordenado por el campo clave.
- ✓ Solamente se puede grabar en un soporte direccionable.Por ejemplo, un disco magnético. Si esto no fuera así, no podría emplear el acceso directo.

2.8.- Otros (secuenciales indexados, hash.).

Existen otros tipos de organización de ficheros, ficheros secuenciales indexados y ficheros de acceso calculado, a continuación se detallan las características de cada uno de ellos.

a. **Ficheros Secuenciales Indexados:**
También llamados parcialmente indexados, al igual que en los ficheros indexados existe una **zona de índices** y otra **zona de registros de datos**, pero esta última se encuentra dividida en **segmentos** (bloques de registros) ordenados.

En la tabla de índices, cada fila hace referencia a cada uno de los segmentos. La clave corresponde al último registro y el índice apunta al registro inicial. Una vez que se accede al primer registro del segmento, dentro de él se localiza (de forma secuencial) el registro buscado.

Esta organización es muy utilizada, tanto para procesos en los que intervienen pocos registros como para aquellos en los que se maneja el fichero completo.
Las principales características son:

- ➔ Permite el acceso secuencial. Esto es muy interesante cuando la tasa de actividad es alta. En el acceso secuencial, además, los registros se leen ordenados por el campo clave.
- ➔ Permite el acceso directo a los registros. Realmente emula el acceso directo, empleando para ello las tablas de índices. Primero busca la clave en el área de índices y luego va a leer al área de datos en la dirección que le indica la tabla.
- ➔ Se pueden actualizar los registros en el mismo fichero, sin necesidad de crear un fichero nuevo de copia en el proceso de actualización.
- ➔ Ocupa mas espacio en el disco que los ficheros secuenciales, debido al uso del área de índices.
- ➔ Solo se puede utilizar soportes direccionables.
- ➔ Obliga a una inversión económica mayor, por la necesidad de programas y, a veces, hardware mas sofisticado.

b. **Ficheros de Acceso Calculado o Hash:**
Cuando utilizamos ficheros indexados es necesario siempre tener que consultar una tabla para obtener la dirección de almacenamiento a partir de la clave. La técnica del acceso calculado o **hash**, permite accesos más rápidos, ya que en lugar de consultar una tabla, se utiliza una transformación o función matemática (función de hashing) conocida, que a partir de la clave genera la dirección de cada registro del archivo. Si la clave es alfanumérica, deberá previamente ser transformada en un número.

El mayor problema que presenta este tipo de ficheros es que a partir de diferentes claves se obtenga la misma dirección al aplicar la función matemática o transformación. A este problema se le denomina **colisión**, y las claves que generan la misma dirección se conocen por **sinónimos**. Para resolver este problema se aplican diferentes métodos, como tener un bloque de excedentes o zona de sinónimos, o crear un archivo de sinónimos, etc.

Para llevar a cabo la transformación existen multitud de métodos, siendo algunos:

- **Módulo:** La dirección será igual al resto de la división entera entre la clave y el número de registros.
- **Extracción:** La dirección será igual a una parte de las cifras que se extraen de la clave.

Una buena transformación o función de hash, será aquella que produzca el menor número de colisiones. En este caso hay que buscar una función, a ser posible biunívoca, que relacione los posibles valores de la clave con el conjunto de números correlativos de dirección. Esta función consistirá en realizar una serie de cálculos matemáticos con el valor de la clave hasta obtener un número entre 1 y n, siendo n el número de direcciones que tiene el fichero.

En un fichero con acceso calculado:

- ☐ Se utiliza la dirección como clave.
- ☐ Hay una tabla en la que está cada clave con la dirección del registro correspondiente.
- ◉ **La dirección se obtiene a partir de la clave mediante un algoritmo.**

2.9.- Parámetros de utilización.

En función del uso que se vaya a dar al fichero, serán adecuados unos tipos u otros de organización. Mediante la utilización de **parámetros de referencia**, podremos determinar el uso de un fichero. Estos parámetros son:

a. **Capacidad o volumen:** es el espacio, en caracteres, que ocupa el fichero. La capacidad podrá calcularse multiplicando el número previsto de registros por la longitud media de cada registro.
b. **Actividad:** permite conocer la cantidad de consultas y modificaciones que se realizan en el fichero. Para poder especificar la actividad se deben tener en cuenta:
   - **Tasa de consulta o modificación:** que es el porcentaje de registros consultados o modificados en cada tratamiento del fichero, respecto al número total de registros contenidos en él.
   - **Frecuencia de consulta o modificación:** número de veces que se accede al fichero para hacer una consulta o modificación en un periodo de tiempo fijo.
c. **Volatilidad:** mide la cantidad de inserciones y borrados que se efectúan en un fichero. Para determinar la volatilidad es necesario conocer:
   - **Tasa de renovación:** es el tanto por ciento de registros renovados en cada tratamiento del fichero, respecto al número total de registros contenidos en él.
   - **Frecuencia de renovación:** es el número de veces que se accede al fichero para renovarlo en un periodo de tiempo fijo.
d. **Crecimiento:** es la variación de la capacidad del fichero y se mide con la tasa de crecimiento, que es el porcentaje de registros en que aumenta el fichero en cada tratamiento.

La volatilidad de un fichero es un parámetro que indica:

- ☐ La variación del volumen del fichero.
- ☐ La cantidad de veces que se abre o cierra el fichero.
- ◉ **El peso de los procesos de inserción y borrado en dicho fichero (frecuencia de renovación).**

3.- Bases de datos.

Caso práctico

__Ada__, __Juan y María__, se han reunido para aclarar ideas sobre qué sistema de gestión de información van a utilizar.
-Bases de datos, está claro. Pero, hay de varios tipos ¿no? -pregunta __Juan__.
__Ada,__ asiente con la cabeza y confirma a sus dos compañeros que la práctica totalidad de los sistemas de información actuales utilizan acceso a bases de datos.

Continúa __Ada__: -Sé que todos conocemos lo que son las bases de datos, pero es necesario afianzar y aclarar muchos conceptos fundamentales que nos van hacer falta para plantear, diseñar y construir las bases de datos que nuestras aplicaciones utilizarán.

Si BK Programación va a desarrollar aplicaciones para diferentes ámbitos, deberá documentarse adecuadamente para poder seleccionar qué sistema de base de datos debe utilizar en cada situación. Para ello, todos sus miembros tendrán que recordar, actualizar o aprender gran cantidad de interesantes conocimientos relacionados con este campo de la informática.

Como hemos visto anteriormente, los ficheros permiten organizar y memorizar conjuntos de datos del mismo tipo o naturaleza con una determinada estructura, siendo un medio para el almacenamiento de los datos o resultados de una aplicación específica. Pero si las aplicaciones, al ser diseñadas, deben depender directamente de sus ficheros o archivos, se pierde independencia y surgen serios inconvenientes: como información duplicada, incoherencia de datos, fallos de seguridad, etc.

Estos problemas debían ser solucionados, es cuando aparece el concepto de base de datos. Una base de datos permitirá reunir toda la información relacionada en un único sistema de almacenamiento, pudiendo cualquier aplicación utilizarla de manera independiente y ofreciendo una mejora en el tratamiento de la información, así como una evolución para el desarrollo de aplicaciones.

La gestión de las bases de datos ha experimentado gran cantidad de cambios, partiendo de aplicaciones especializadas hasta llegar a convertirse en el núcleo de los entornos informáticos modernos. Con la llegada de Internet en los noventa, el número de usuarios de bases de datos creció exponencialmente, y aunque muchos de ellos no sean conscientes de ello, el acceso a dichas bases forma parte de la vida cotidiana de muchos de nosotros.

Conocer los sistemas que gestionan las bases de datos, sus conceptos fundamentales, el diseño, lenguajes y la implementación de éstas, podemos considerarlo imprescindible para alguien que se está formando en el campo de la informática.

3.1.- Conceptos.

A finales de los setenta, la aparición de nuevas tecnologías de manejo de datos a través de los sistemas de bases de datos supuso un considerable cambio. Los sistemas basados en ficheros separados dieron paso a la utilización de sistemas gestores de bases de datos, que son sistemas software centralizados o distribuidos que ofrecen facilidades para la definición de bases de datos, selección de estructuras de datos y búsqueda de forma interactiva o mediante lenguajes de programación.
Llegados a este punto, te preguntarás... ¿Qué es una base de datos?

Base de datos: Es una colección de datos relacionados lógicamente entre sí, con una definición y descripción comunes y que están estructurados de una determinada manera. Es un conjunto estructurado de datos que representa entidades y sus interrelaciones, almacenados con la mínima

redundancia y posibilitando el acceso a ellos eficientemente por parte de varias aplicaciones y usuarios.

La base de datos no sólo contiene los datos de la organización, también almacena una descripción de dichos datos. Esta descripción es lo que se denomina **metadatos**, se almacena en el **diccionario de datos o catálogo** y es lo que permite que exista **independencia de datos** lógica-física.

Una base de datos constará de los siguientes elementos:

- ✓ **Entidades:** objeto real o abstracto con características diferenciadoras de otros, del que se almacena información en la base de datos. En una base de datos de una clínica veterinaria, posibles entidades podrían ser: ejemplar, doctor, consulta, etc.
- ✓ **Atributos:** son los datos que se almacenan de la entidad. Cualquier propiedad o característica de una entidad puede ser atributo. Continuando con nuestro ejemplo, podrían ser atributos: raza, color, nombre, número de identificación, etc.
- ✓ **Registros:** donde se almacena la información de cada entidad. Es un conjunto de atributos que contienen los datos que pertenecen a una misma repetición de entidad. En nuestro ejemplo, un registro podría ser: 2123056, Sultán, Podenco, Gris, 23/03/2009.
- ✓ **Campos:** donde se almacenan los atributos de cada registro. Teniendo en cuenta el ejemplo anterior, un campo podría ser el valor Podenco.

Una base de datos es:

- ○ Un programa para gestionar archivos muy grandes.
- ○ El conjunto de datos de los usuarios almacenados en un único disco duro.
- ◉ Conjunto de datos de distinto tipo relacionados entre sí, junto con un programa de gestión de dichos datos.

3.2.- Usos.

Ya sabemos lo que es una base de datos y sus características principales, pero es necesario conocer quien las usa y para qué.

¿Quién utiliza las bases de datos?

Existen cuatro tipos de personas que pueden hacer uso de una base de datos: el administrador, los diseñadores de la base de datos, los programadores de aplicaciones y los usuarios finales.

¿Quién utiliza las bases de datos?	
Tipo	**Funciones y características**
El administrador	Es la persona encargada de la creación o implementación física de la base de datos. Es quien escoge los tipos de ficheros, los índices que hay que crear, la ubicación de éstos, etc. En general, es quien toma las decisiones relacionadas con el funcionamiento físico del almacenamiento de información. Siempre teniendo en cuenta las posibilidades del sistema de información con el que trabaje. Junto a estas tareas, el administrador establecerá la política de seguridad y de acceso para garantizar el menor número de problemas.
Los diseñadores	Son las personas encargadas de diseñar cómo será la base de datos. Llevarán a cabo la identificación de los datos, las relaciones entre ellos, sus restricciones, etc. Para ello han de conocer a fondo los datos y procesos a representar en la base de datos. Si estamos hablando de una empresa, será necesario que conozcan las reglas de negocio en la que esta se mueve. Para obtener un buen resultado, el diseñador de la base de datos debe implicar en el proceso a todos los usuarios de la base de datos, tan pronto como sea

	posible.
Los programadores de aplicaciones	Una vez diseñada y construida la base de datos, los programadores se encargarán de implementar los programas de aplicación que servirán a los usuarios finales. Estos programas de aplicación ofrecerán la posibilidad de realizar consultas de datos, inserción, actualización o eliminación de los mismos. Para desarrollar estos programas se utilizan lenguajes de tercera o cuarta generación.
Los usuarios finales	Son los clientes finales de la base de datos. Al diseñar, implementar y mantener la base de datos se busca cumplir los requisitos establecidos por el cliente para la gestión de su información.

¿Para qué se utilizan las bases de datos?

Enumerar todos y cada uno de los campos donde se utilizan las bases de datos es complejo, aunque seguro que quedarán muchos en el tintero, a continuación se recopilan algunos de los ámbitos donde se aplican.

- ✓ Banca: información de clientes, cuentas, transacciones, préstamos, etc.
- ✓ Líneas aéreas: información de clientes, horarios, vuelos, destinos, etc.
- ✓ Universidades: información de estudiantes, carreras, horarios, materias, etc.
- ✓ Transacciones de tarjeta de crédito: para comprar con tarjetas de crédito y la generación de los extractos mensuales.
- ✓ Telecomunicaciones: para guardar registros de llamadas realizadas, generar facturas mensuales, mantener el saldo de las tarjetas telefónicas de prepago y almacenar información sobre las redes.
- ✓ Medicina: información hospitalaria, biomedicina, genética, etc.
- ✓ Justicia y Seguridad: delincuentes, casos, sentencias, investigaciones, etc.
- ✓ Legislación: normativa, registros, etc.
- ✓ Organismos públicos: datos ciudadanos, certificados, etc.
- ✓ Sistemas de posicionamiento geográfico.
- ✓ Hostelería y turismo: reservas de hotel, vuelos, excursiones, etc.
- ✓ Ocio digital: juegos online, apuestas, etc.
- ✓ Cultura: gestión de bibliotecas, museos virtuales, etc.
- ✓ Etc.

3.3.- Ubicación de la información.

Utilizamos a diario las bases de datos, pero ¿Dónde se encuentra realmente almacenada la información?. Las bases de datos pueden tener un tamaño muy reducido (1 MegaByte o menos) o bien, ser muy voluminosas y complejas (del orden de Terabytes). Sin embargo todas las bases de datos normalmente se almacenan y localizan en discos duros y otros dispositivos de almacenamiento, a los que se accede a través de un ordenador. Una gran base de datos puede necesitar servidores en lugares diferentes, y viceversa, pequeñas bases de datos pueden existir como archivos en el disco duro de un único equipo.

A continuación, se exponen los sistemas de almacenamiento de información más utilizados para el despliegue de bases de datos, comenzaremos por aquellos en los que pueden alojarse bases de datos de tamaño pequeño y mediano, para después analizar los sistemas de alta disponibilidad de grandes servidores.

- ✓ **Discos SATA:**Es una interfaz de transferencia de datos entre la placa base y algunos dispositivos de almacenamiento, como puede ser el disco duro, lectores y regrabadores de CD/DVD/BD, Unidades de Estado Sólido u otros dispositivos. SATA proporciona mayores velocidades, mejor aprovechamiento cuando hay varias unidades, mayor longitud del cable de transmisión de datos y capacidad para conectar unidades al instante, es decir, insertar el dispositivo sin tener que apagar el ordenador. La primera generación especifica en transferencias de 150 Megabytes por

segundo, también conocida por SATA 150 MB/s o Serial ATA-150. Actualmente se comercializan dispositivos SATA II, a 300 MB/s, también conocida como Serial ATA-300 y los SATA III con tasas de transferencias de hasta 600 MB/s.

- ✓ **Discos SCSI:** Son interfaces preparadas para discos duros de gran capacidad de almacenamiento y velocidad de rotación. Se presentan bajo tres especificaciones: SCSI Estándar (Standard SCSI), SCSI Rápido (Fast SCSI) y SCSI Ancho-Rápido (Fast-Wide SCSI). Su tiempo medio de acceso puede llegar a 7 milisegundos y su velocidad de transmisión secuencial de información puede alcanzar teóricamente los 5 MB/s en los discos SCSI Estándares, los 10 MBps en los discos SCSI Rápidos y los 20 MBps en los discos SCSI Anchos-Rápidos (SCSI-2). Un controlador SCSI puede manejar hasta 7 discos duros SCSI.
- ✓ **RAID:** acrónimo de Redundant Array of Independent Disks o matriz de discos independientes, es un contenedor de almacenamiento redundante. S**e basa en el montaje en conjunto de dos o más discos duros,** formando un bloque de trabajo, para obtener desde una ampliación de capacidad a mejoras en velocidad y seguridad de almacenamiento. Según las características que queramos primar, se establecen distintos sistemas de RAID.
- ✓ **Sistemas NAS:** Es el acrónimo de Network Attached Storage ó sistema de almacenamiento masivo en red. Estos sistemas de almacenamiento permiten compartir la capacidad de almacenamiento de un computador (Servidor) con ordenadores personales o servidores clientes a través de una red, haciendo uso de un sistema operativo optimizado para dar acceso a los datos a través de protocolos de comunicación específicos. Suelen ser dispositivos para almacenamiento masivo de datos con capacidades muy altas, de varios Terabytes, generalmente superiores a los discos duros externos y además se diferencian de estos al conectar por red.
- ✓ **Sistemas SAN:** Acrónimo de Storage Area Network o red de área de almacenamiento.Se trata de una red concebida para conectar servidores, matrices (arrays) de discos y librerías de soporte. La arquitectura de este tipo de sistemas permite que los recursos de almacenamiento estén disponibles para varios servidores en una red de área local o amplia. Debido a que la información almacenada no reside directamente en ninguno de los servidores de la red, se optimiza el poder de procesamiento para aplicaciones comerciales y la capacidad de almacenamiento se puede proporcionar en el servidor donde más se necesite.

Puedes ampliar más información sobre algunos de los sistemas de almacenamiento vistos, además de tendencias y curiosidades en almacenamiento, a través de los siguientes enlaces:

http://es.wikipedia.org/wiki/RAID
http://www.ideasgeek.net/2010/05/12/nas-dispositivo-de-almacenamiento-externo-masivo-conectado-por-red/
https://tihuilo.wordpress.com/2010/05/27/sistemas-de-almacenamiento/
http://www.oracle.com/technetwork/server-storage/general/3d-demos-333955.html
http://databaseandtech.wordpress.com/2009/06/22/fusion-tables-google-trae-la-base-de-datos-a-la-nube-del-internet/
http://www.cosasquecontar.com/2011/04/bigtable-como-google-almacena-los-datos/

Rellena los huecos con los conceptos adecuados.

Un tipo de red donde se optimiza el poder de procesamiento para aplicaciones comerciales, pudiendo proporcionarse la capacidad de almacenamiento en el servidor donde más se necesite, se denomina sistema SAN.

En efecto, se trata de una red de área de almacenamiento. Este tipo de tecnología permite conectividad de alta velocidad, de servidor a almacenamiento, almacenamiento a almacenamiento, o servidor a servidor. Este método usa una infraestructura de red por separado, evitando así cualquier problema asociado con la conectividad de las redes existentes.

4.- Modelos de bases de datos.

Caso práctico

Juan *tiene ya experiencia con bases de datos: -Registros, tablas, relaciones, claves,... tiene su teoría, pero dame un problema a resolver y casi puedo construir la base de datos en un abrir y cerrar de ojos.*

María *ve como* ***Ada,*** *algo escéptica al respecto, aclara a Juan algunas ideas:* ***-Juan****, la experiencia es un grado como siempre hemos destacado, pero es imprescindible conocer y dominar los conceptos más importantes sobre bases de datos. Al igual que comenzar a programar directamente codificando, implementar una base de datos directamente sin detenerse a realizar un análisis previo y emplear las herramientas adecuadas, puede provocar muchos quebraderos de cabeza.*

Ada *indica a* ***María:*** *-Las bases de datos no siempre han sido como las conocemos ahora, han habido diferentes modelos para su construcción y es bueno conocer la evolución de éstos para comprender por qué utilizaremos el modelo de bases de datos relacional.*

La clasificación tradicional de las bases de datos establece tres modelos de bases de datos: jerárquico, en red y relacional. En la actualidad el modelo de bases de datos más extendido es el relacional. Aunque, hay que tener en cuenta que dos de sus variantes (modelo de bases de datos distribuidas y orientadas a objetos) son las que se más se están utilizando en los últimos tiempos.

En los siguientes epígrafes analizaremos cada uno de ellos, así como otros modelos de bases de datos existentes.

Conoce las características generales y graba en tu memoria fotográfica los gráficos que representan a cada uno de los modelos expuestos en el siguiente artículo:

http://es.kioskea.net/contents/bdd/bddtypes.php3

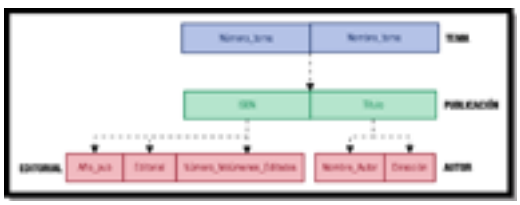
4.1.- Modelo jerárquico.

Cuando IBM creó su Sistema Administrador de Información o IMS, se establecieron las bases para que la gran mayoría de sistemas de gestión de información de los años setenta utilizaran el modelo jerárquico. También recibe el nombre de modelo en árbol, ya que utiliza una estructura en árbol invertido para la organización de los datos.

La información se organiza con una jerarquía en la que la relación entre las entidades de este modelo siempre es del tipo padre/hijo. De tal manera que existen nodos que contienen atributos o campos y que se relacionarán con sus nodos hijos, pudiendo tener cada nodo más de un hijo, pero un nodo siempre tendrá un sólo 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.

Hoy en día, debido a sus limitaciones, el modelo jerárquico está en desuso. En el siguiente gráfico puedes observar la estructura de almacenamiento del modelo jerárquico.



Si deseas completar tus conocimientos acerca de este modelo, te proponemos los siguientes enlaces:

http://es.wikipedia.org/wiki/Modelo_jer%C3%A1rquico

http://ddd.uab.cat/pub/elies/elies_a2000v9/4-2-1.htm

Rellena los huecos con los conceptos adecuados.
El modelo Jerárquico es un modelo muy rígido en el que las diferentes entidades se organizan en niveles múltiples, de acuerdo a una estricta relación **padre**/**hijo**, de manera que un **padre** puede tener más de un **hijo**, todos ellos localizados en el mismo nivel, y un **hijo** únicamente puede tener un **padre** situado en el nivel inmediatamente superior al suyo.

Como puedes ver en el gráfico anterior, la estructura representa las relaciones padre/hijo y las despliega en forma de árbol invertido. De esta manera un padre tiene un hijo o varios, y un hijo sólo podrá tener un padre.

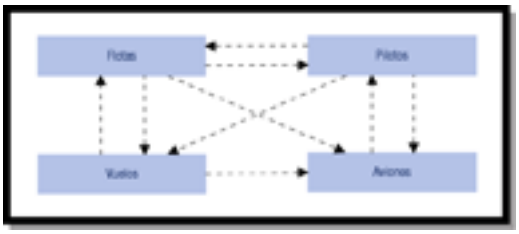
4.2.- Modelo en red.

El modelo de datos en red aparece a mediados de los sesenta como respuesta a limitaciones del modelo jerárquico en cuanto a representación de relaciones más complejas. Podemos considerar a IDS (Integrated Data Store) de Bachman como el primer sistema de base de datos en red. Tras él se intentó crear un estándar de modelo de red por parte de CODASYL, siendo un modelo que tubo gran aceptación a principios de los setenta.

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, pero hace muy complicado su manejo. Al no tener que duplicar la información se ahorra espacio de almacenamiento.

El sistema de gestión de información basado en el modelo en red más popular es el sistema IDMS.



Si deseas completar tus conocimientos acerca de este modelo, te proponemos los siguientes enlaces:

http://es.wikipedia.org/wiki/Modelo_de_red
http://ddd.uab.cat/pub/elies/elies_a2000v9/4-2-2.htm

4.3- Modelo relacional.

Este modelo es posterior a los dos anteriores y fue desarrollado por Codd en 1970. Hoy en día las bases de datos relacionales son las más utilizadas.

En el modelo relacional la base de datos es percibida por el usuario como un conjunto de tablas. Esta percepción es sólo a nivel lógico, ya que a nivel físico puede estar implementada mediante distintas estructuras de almacenamiento.

El modelo relacional utiliza **tablas bidimensionales** (relaciones) para la representación lógica de los datos y las relaciones entre ellos. Cada relación (tabla) posee un nombre que es único y contiene un conjunto de columnas.

Se llamará **registro, entidad o tupla** a cada fila de la tabla y **campo o atributo** a cada columna de la tabla.

A los conjuntos de valores que puede tomar un determinado atributo, se le denomina **dominio**.

Una **clave** será un atributo o conjunto de atributos que identifique de forma única a una tupla.

Las tablas deben cumplir una serie de requisitos:
- ✓ Todos los registros son del mismo tipo.
- ✓ La tabla sólo puede tener un tipo de registro.
- ✓ No existen campos o atributos repetidos.
- ✓ No existen registros duplicados.
- ✓ No existe orden en el almacenamiento de los registros.
- ✓ Cada registro o tupla es identificada por una clave que puede estar formada por uno o varios campos o atributos.

A continuación puedes observar cómo es una relación con sus tuplas y atributos en el modelo relacional.

El lenguaje habitual para construir las consultas a bases de datos relacionales es SQL, Structured Query Language o Lenguaje Estructurado de Consultas, un estándar implementado por los principales motores o sistemas de gestión de bases de datos relacionales.

Durante su diseño, una base de datos relacional pasa por un proceso al que se conoce como normalización de una base de datos.

Si deseas completar tus conocimientos acerca de este modelo, te proponemos el siguiente enlace:

http://es.wikipedia.org/wiki/Modelo_relacional

Rellena los huecos con los conceptos adecuados.

La **normalización** de bases de datos relacional consiste en definir las reglas que determinan las dependencias entre los datos de una base de datos relacional. Si definimos esta relación o dependencia entre los elementos de una determinada base de datos de la manera más sencilla posible, conseguiremos que la cantidad de espacio necesario para guardar los datos sea el menor posible y la facilidad para actualizar la relación sea la mayor posible. Es decir, optimizaremos su funcionamiento.

El proceso de normalización tiene una gran relevancia en el diseño de bases de datos. En futuras unidades de trabajo se abordarán las técnicas para llevar a cabo esta optimización.

4.4.- Modelo orientado a objetos.

El modelo orientado a objetos define una base de datos en términos de **objetos**, sus propiedades y sus operaciones. Los objetos con la misma estructura y comportamiento pertenecen a una **clase**, y las clases se organizan en jerarquías. Las operaciones de cada clase se especifican en términos de procedimientos predefinidos denominados **métodos**. Algunos sistemas existentes en el mercado, basados en el modelo relacional, han sufrido evoluciones incorporando conceptos orientados a objetos. A estos modelos se les conoce como sistemas **objeto-relacionales**.

El objetivo del modelo orientado a objetos es cubrir las limitaciones del modelo relacional. Gracias a este modelo se incorporan mejoras como la herencia entre tablas, los tipos definidos por el usuario, disparadores almacenables en la base de datos (triggers), soporte multimedia, etc.

Los conceptos más importantes del paradigma de objetos que el modelo orientado a objetos incorpora son:

- ✓ **Encapsulación** - Propiedad que permite ocultar la información al resto de los objetos, impidiendo así accesos incorrectos o conflictos.
- ✓ **Herencia** - Propiedad a través de la cual los objetos heredan comportamiento dentro de una jerarquía de clases.
- ✓ **Polimorfismo** - Propiedad de una operación mediante la cual puede ser aplicada a distintos tipos de 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. Este modelo es considerado como el fundamento de las bases de datos de tercera generación, siendo consideradas las bases de datos en red como la primera y las bases de datos relacionales como la segunda generación. Aunque no han reemplazado a las bases de datos relacionales, si son el tipo de base de datos que más está creciendo en los últimos años.

Si deseas completar tus conocimientos acerca de este modelo, te proponemos el siguiente enlace:

http://es.wikipedia.org/wiki/Base_de_datos_orientada_a_objetos

4.5.- Otros modelos.

Además de los modelos clásicos vistos hasta el momento, vamos a detallar a continuación las particularidades de otros modelos de bases de datos existentes y que, en algunos casos, son una evolución de los clásicos.

Modelo Objeto-Relacional

Las bases de datos pertenecientes a este modelo, son un híbrido entre las bases del modelo relacional y el orientado a objetos. El mayor inconveniente de las bases de datos orientadas a objetos radica en los costes de la conversión de las bases de datos relacionales a bases de datos orientadas a objetos.

En una base de datos objeto-relacional (BDOR) siempre se busca obtener lo mejor del modelo relacional, incorporando las mejoras ofrecidas por la orientación a objetos. En este modelo se siguen almacenando tuplas, aunque la estructura de las tuplas no está restringida sino que las relaciones pueden ser definidas en función de otras, que es lo que denominamos herencia directa.

El estándar en el que se basa este modelo es SQL99. Este estándar ofrece la posibilidad de añadir a las bases de datos relacionales procedimientos almacenados de usuario, triggers, tipos definidos por el usuario, consultas recursivas, bases de datos OLAP, tipos LOB, ...

Otra característica a destacar es la capacidad para incorporar funciones que tengan un código en algún lenguaje de programación como por ejemplo: SQL, Java, C, etc.

La gran mayoría de las bases de datos relacionales clásicas de gran tamaño, como Oracle, SQL Server, etc., son objeto-relacionales.

Modelo de bases de datos deductivas

En este modelo las bases de datos almacenan la información y permiten realizar deducciones a través de inferencias. Es decir, se derivan nuevas informaciones a partir de las que se han introducido explícitamente en la base de datos por parte del usuario.

Las bases de datos deductivas son también llamadas bases de datos lógicas, al basarse en lógica matemática. Surgieron para contrarrestar las limitaciones del modelo relacional para la respuesta a consultas recursivas y la deducción de relaciones indirectas entre los datos almacenados.

Si deseas completar tus conocimientos sobre las bases de datos deductivas, te proponemos el siguiente enlace:

2. Bases de datos deductivas

Relaciones básicas:

PIEZA (codpieza: D1, desc: D2, peso: D3)
CP = {codpieza}

PROV (codprov: D4, nombre: D5, zona: D6)
CP = {codprov}

PRECIOS (codprov: D4, codpieza: D1, precio: D7)
CP = {codprov, codpieza}
CAj = {codprov} → PROV
CAj = {codpieza} → PIEZA

COMP (pieza1: D1, pieza2: D1)
CP = {pieza1, pieza2}
CAj = {pieza1} → PIEZA
CAj = {pieza2} → PIEZA

Relaciones derivadas:

PRECIOS3 (codprov: D4, codpieza: D1, precio: D7)
CP = {codprov, codpieza}
CAj = {codprov} → PROV
CAj = {codpieza} → PIEZA

PRECIOS_EXT (codprov: D4, nombre: D5, codpieza: D1, desc: D2, precio: D7)
CP = {codprov, codpieza}
CAj = {codprov} → PROV
CAj = {codpieza} → PIEZA

COMPONENTE (pieza1: D1, pieza2: D1)
CP = {pieza1, pieza2}
CAj = {pieza1} → PIEZA
CAj = {pieza2} → PIEZA

Restricciones de integridad:
∀x ∀y (COMPONENTE (x,y) → ¬ COMPONENTE (y,x))

Esquema

2. Bases de datos deductivas

BDD

PIEZA

codpieza	desc	peso
pz1	tornillo	10
pz2	tuerca	11
pz3	arandela	8

PROV

codprov	nombre	zona
pv1	Juan	1
pv2	Carlos	9
pv3	Luis	9

PRECIOS

codprov	codpieza	precio
pv1	pz2	10
pv1	pz3	20
pv3	pz2	90
pv3	pz1	50

COMP

pieza1	pieza2
pz1	pz2
pz2	pz3

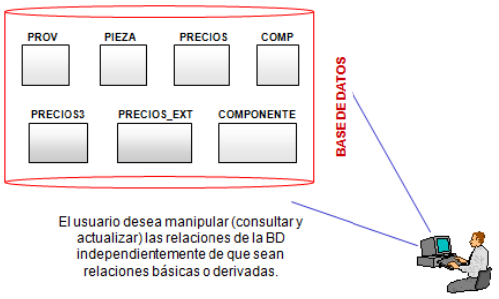
Reglas deductivas:

1 precios3 (x, y,z) ← ∃z (precios (x, y, z) ∧ prov (x, w, 3))

2 componente (x, y) ← ∃z (comp (x, z) ∧ componente (z, y))
3 componente (x, y) ← comp (x, y)

4 precios_ext (x,n,y,d,p) ← ∃z∃z (prov (x, n,z) ∧ pieza (y, d,w) ∧ precios (x, y, p))

2. Bases de datos deductivas



2. Bases de datos deductivas

Mecanismo de **vistas** del modelo relacional ➡ Definición de información implícita

Relación derivada ➡ VISTA

Base de datos deductiva ➡ Base de datos relacional con vistas

2. Bases de datos deductivas

Limitaciones del modelo relacional (SQL92):

Definición de vistas ➡ Limitaciones en la definición de vistas recursivas

Actualización ➡ Limitaciones en la actualización de las vistas

SGBD relacionales ➡ Ausencia de procedimientos para la evaluación de consultas recursivas

2. Bases de datos deductivas.

➡ Los sistemas de gestión de bases de datos deductivas deben superar las limitaciones de los sistemas relacionales

PROBLEMAS:

✓ Formalización
✓ Actualización de la base de datos ➡ **LÓGICA**
✓ Construcción de SGBD deductivos

2. Bases de datos deductivas.

Formalización: Si intentamos representar la información explícita y la información implícita en un mismo lenguaje (lenguaje de 1er orden) obtenemos un programa lógico:

Base de datos deductiva

Hechos:

pieza (pz1, tornillo, 10)
...
prov (pv1, Juan, 1)
...
comp (pz1, pz3)
...

Reglas deductivas:

precios3 (x, y, z) ← ∃w (prov (x, w, 3) ∧ precios (x, y, z))

componente (x, y) ← ∃z (comp (x, z) ∧ componente (z, y))
componente (x, y) ← comp (x, y)

precios_ext (x,n,y,d,p) ← ∃z∃w (prov (x, n, z) ∧ pieza (y, d, w) ∧ precios (x, y, p))

2. Bases de datos deductivas

MARCO FORMAL: Lógica de 1er orden (Programación Lógica)

Esquema de BDD:

(L, RI): - L es un lenguaje de 1er orden
- RI es un conjunto de f.b.f de L (restricciones de integridad)

BDD: (programa lógico)

{A: A es un átomo base} (hechos)
∪
{A ← $L_1 \wedge L_2 \wedge \ldots \wedge L_n$: A es un átomo y L_i es un literal} (reglas)

2. Bases de datos deductivas

Semántica de una BDD

Semántica de una BDD: definir el conocimiento existente en la base de datos.

¿qué es cierto en la BDD?

:

Semántica declarativa: conocimiento en la BDD

Semántica operacional: procedimiento para obtener el conocimiento

2. Bases de datos deductivas

Semántica de una BDD

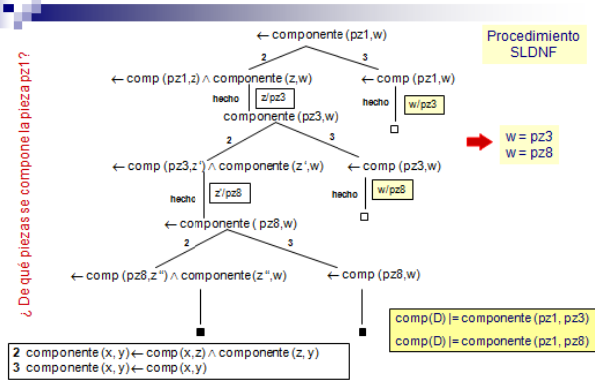
Programación lógica: semántica operacional: SLDNF
semántica declarativa: comp(D)

Semántica operacional: **procedimiento SLDNF**

SLDNF: - procedimiento de refutación
- reglas de inferencia:
  - resolución
  - negación como fallo

Semántica declarativa asociada al SLDNF: **compleción de D**

¿ De qué piezas se compone la pieza pz1?



3. Actualización de base de datos deductivas

Actualización sobre relación derivada → Actualización(es) sobre relación(es) básicas(s)

"Dada una base de datos D (D=BDI ∪ BDE) y un requisito de actualización insertar (A) (resp. borrar (A)) donde A es una tupla de una relación derivada, encontrar una transacción T sobre EDB tal que T(D) satisfaga el requisito de actualización"

Ejemplo: DELETE FROM PRECIOS3 WHERE codprov=pv1

PIEZA

codpieza	desc	peso
pz1	tornillo	10
pz3	tuerca	11
pz8	arandela	8

PRECIOS

codprov	codpieza	precio
pv1	pz3	10
pv1	pz8	20
pv3	pz8	30
pv5	pz1	50

PROV

codprov	nombre	zona
pv1	Juan.....	1
pv5	Carlos	3
pv3	Luis	3

PRECIOS_EXT

codprov	nombre	codpieza	desc	precio
pv1	Juan...	pz3	tuerca	10
pv1	Juan...	pz8	arandela	20
pv3	Luis...	pz8	arandela	30
pv5	Carlos..	pz1	tornillo	50

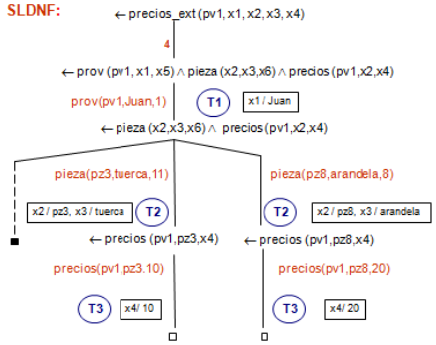
T1={borrar (PROV (pv1,Juan,1))}
T2={borrar (PIEZA (pz3,tuerca,11),
borrar (PIEZA (pz8,arandela,8))}
T3={borrar (PRECIOS (pv1,pz3,10),
borrar (PRECIOS (pv1,pz8,20))}

3. Actualización de bases de datos deductivas

Métodos para la actualización de bases de datos deductivas

Utilización de los procedimientos de evaluación de consultas para determinar los posibles caminos de derivación del conocimiento que se desea a actualizar

SLDNF:



3.1 Actualización

Enunciado general del problema:

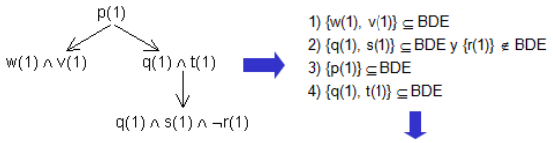
Dados el esquema (L,RI) de una base de datos deductiva, un estado de base de datos D de ese esquema tal que ∀W∈RI se cumple que D *satisface* W, y dado un requisito de actualización U tal que U *no es cierto* en D entonces encontrar una transacción T tal que ∀W∈RI, D' = T(D) *satisface* W y U *es cierto* en D'.

3.1 Actualización

Ejemplo 1

1. p(x) ← q(x) ∧ t(x)
2. p(x) ← w(x) ∧ v(x)
3. t(x) ← s(x) ∧ ¬r(x)

Actualización: U = p(1)



1) {w(1), v(1)} ⊆ BDE
2) {q(1), s(1)} ⊆ BDE y {r(1)} ∉ BDE
3) {p(1)} ⊆ BDE
4) {q(1), t(1)} ⊆ BDE

Obtener transacciones que aseguren una de estas cuatro situaciones

3.1 Actualización

Caracterización del problema:

1) Tiempo de generación de la solución.
2) Variables cuantificadas existencialmente
3) Recursividad
4) Información asumida
5) Tratamiento de restricciones de integridad

3.1 Actualización

1) Tiempo de generación de la solución.

- Tiempo de ejecución: el árbol de derivación para el requisito de actualización se genera cuando la actualización es solicitada.
- Tiempo de definición: el árbol de derivación para un requisito de actualización se estudia cuando se define el esquema de la base de datos, lo que supone una mejora ya que determinadas tareas sólo se realizan una vez.
- Mixto: en este caso una parte de la solución se genera en tiempo de definición del esquema y se completa en tiempo de ejecución.

3.1 Actualización

En el Ejemplo 1:

- un método que obtuviese la solución en tiempo de ejecución estudiaría el árbol de derivación de la actualización p(1) para encontrar una solución.
- un método que trabajase en tiempo de definición del esquema estudiaría el requisito genérico p(x) para obtener soluciones que luego se instanciarían en tiempo de ejecución.

3.1 Actualización

2) Variables existencialmente cuantificadas.

Dada una regla deductiva de una base de datos normal, a las variables que aparecen en el cuerpo de la regla y no aparecen en la cabeza se les denomina variables existencialmente cuantificadas.

$$\forall x_1 \ldots \forall x_i \ldots \forall x_m (A \leftarrow L_1 \wedge \ldots \wedge L_n)$$
$$\equiv \quad (x_i \text{ no aparece en } A)$$
$$\forall x_1 \ldots \forall x_{i-1} \forall x_{i+1} \ldots \forall x_m (A \leftarrow \exists x_i (L_1 \wedge \ldots \wedge L_n))$$

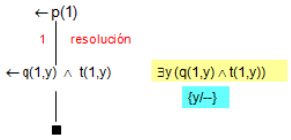
La presencia de variables existencialmente cuantificadas en las reglas deductivas puede provocar la aparición del problema llamado falta de valores durante la generación de las transacciones que resuelven un requisito de actualización.

3.1 Actualización

Ejemplo 2:

BDD: 1. p(x) ← q(x,y) ∧ t(x,y)
....
t(1,2)

Actualización: U = p(1).



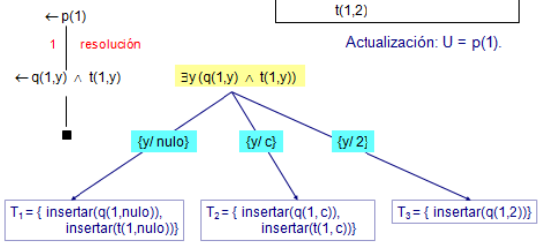
Una solución sencilla a este problema consiste en utilizar el valor nulo para la variable de la que se desconoce el valor o bien un valor cualquiera proporcionado por el usuario o extraído sin criterio de la base de datos. Aunque en ocasiones esta solución es la única posible, en otras se puede elegir un valor tal que la transacción obtenida sea más sencilla.

3.1 Actualización

Ejemplo 2:

BDD: 1. p(x) ← q(x,y) ∧ t(x,y)
....
t(1,2)

Actualización: U = p(1).



3.1 Actualización

3) Recursividad.

La presencia de reglas recursivas en la base de datos puede complicar la generación de la transacción, ya que el árbol de derivación puede ser infinito para un determinado requisito de actualización, lo que supone la existencia de infinitas transacciones posibles para satisfacerlo.

3.1 Actualización

Ejemplo 3:

BDD: 1. $p(x,y) \leftarrow q(x,y)$
2. $p(x,y) \leftarrow q(x,z) \wedge p(z,y)$

Actualización: U = p(1,1).

Para satisfacer este requisito hay infinitas transacciones posibles:

T_1: {insertar(q(1,1))}
T_2: {insertar(q(1,2)), insertar(q(2,1))}
T_3: {insertar(q(1,2)), insertar(q(2,3)), insertar(q(3,1))}
…

3.1 Actualización

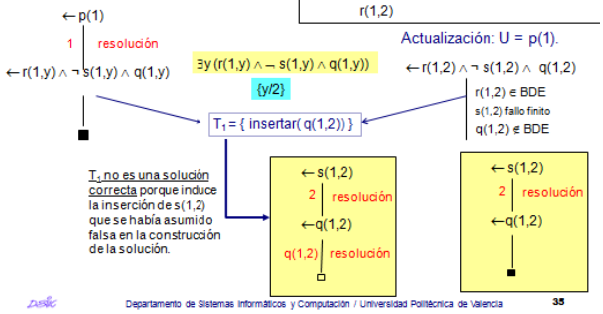
4) Problema de la información asumida.

En presencia de negación en las reglas deductivas, es posible que algunas soluciones que podrían parecer correctas no lo sean, ya que alguna información que se ha supuesto cierta (resp. falsa), durante la construcción de la solución pase a ser falsa (resp. cierta) debido a las actualizaciones propuestas más adelante.

3.1 Actualización

Ejemplo 4:

BDD: 1. $p(x) \leftarrow r(x,y) \wedge \neg s(x,y) \wedge q(x,y)$
2. $s(1,2) \leftarrow q(1,2)$
$r(1,2)$



3.1 Actualización

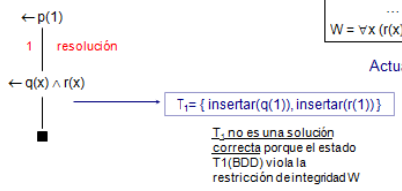
5) Tratamiento de las restricciones de integridad.

La solución propuesta para un requisito de actualización puede suponer la violación de alguna restricción de integridad por lo que es interesante estudiar cómo integra cada método, si lo hace, su estrategia con la comprobación de las restricciones del esquema.

3.1 Actualización

Ejemplo 5:

BDD: 1. $p(x) \leftarrow q(x) \wedge r(x)$
…
$W = \forall x\,(r(x) \rightarrow t(x))$



Un método que integre la generación de las soluciones con la restauración de la integridad podría generar la transacción {insertar(q(1)), insertar(r(1)), insertar(t(1))}.

3.1 Actualización

Estudio de un método de actualización:

1. Semántica asumida: la semántica declarativa determina el conjunto de posibles soluciones al requisito de actualización y la semántica operacional constituye la herramienta para computar esas soluciones.
2. Bases de datos: tipo de bases de datos y de restricciones de integridad para los que está definido el método.
3. Requisitos de actualización: forma sintáctica de los requisitos de actualización permitidos en el método.
4. Transacciones generadas: tipo de soluciones obtenidas y si sólo se obtiene una o todas las soluciones posibles.
5. Descripción del método: estrategia seguida para generar las transacciones.
6. Corrección y completitud del método.
7. Resumen: cómo el método resuelve los problemas antes mencionados.

La Lógica en el desarrollo de las Bases de Datos

1. Lógica y Bases de Datos.
2. Bases de datos deductivas.
3. Actualización de bases de datos deductivas.
   - 3.1 Actualización
   - 3.2 Comprobación de la integridad
   - 3.3 Restauración de la consistencia

3.2 Comprobación de la integridad

Restricción de integridad: propiedad del mundo real que una base de datos debe *satisfacer* en cualquier instante para ser consistente con cierto modelo del mundo real.

Restricciones estáticas: hacen referencia a un único estado de la base de datos. Estas restricciones restringen los estados válidos con independencia de la secuencia de los mismos.

Restricciones dinámicas: hacen referencia a dos o más estados de la base de datos. Estas restricciones restringen las secuencias de estados válidas. Un caso particular de restricciones dinámicas son las restricciones de transición que restringen dos estados consecutivos válidos.

3.2 Comprobación de la integridad

Comprobación de la integridad: la comprobación de la integridad en bases de datos consiste en *comprobar* si el par de estados (D,D') implicados en una transacción T *satisface* las restricciones de transición y si el estado final D' *satisface* las restricciones estáticas.

Método de comprobación de la integridad: es un procedimiento de decisión tal que, dado un estado D y una restricción de integridad estática W, decide con una respuesta binaria si/no si el estado D *satisface/viola* la restricción estática W.

3.2 Comprobación de la integridad

La forma más sencilla de comprobar las restricciones estáticas es evaluar cada una de ellas después de la transacción; sin embargo esta aproximación puede ser muy costosa en bases de datos voluminosas.

La comprobación de la integridad podría simplificarse si consideráramos sólo los "cambios" que la transacción ha producido en la base de datos.

Todos los métodos propuestos para simplificar la comprobación de la integridad suponen que la base de datos era íntegra antes de la transacción. Apoyándose en esta hipótesis, los métodos comprueban sólo instancias de las restricciones generadas a partir de las *actualizaciones* (inserciones y borrados) de la transacción, evitando comprobar instancias que ya se satisfacían antes de la transacción y que además no se ven afectadas por ésta.

3.2 Comprobación de la integridad

Comprobación simplificada de la integridad en bases de datos relacionales:

PRECIOS (codprov: D4, codpieza: D1, precio: D7)
CP = {codprov, codpieza}
CAj = {codprov} → PROV
CAj = {codpieza} → PIEZA

W: ∀x ∀y ∀z ((precios(x, y, z) → ∃w ∃t (prov(x,w,t)))

T = { insertar (PRECIOS (pv11,pz3,100))}

D —— T ——→ D'

Si D es un estado íntegro
entonces
D' satisface W si y sólo si D' satisface ∃w ∃t (prov (pv11,w,t))

3.2 Comprobación de la integridad

Comprobación simplificada de la integridad en bases de datos deductivas:

En bases de datos deductivas las actualizaciones generadas por una transacción no son sólo las explícitamente requeridas por ésta (operaciones que la componen) sino también todas las actualizaciones que se pueden inducir por la presencia de reglas deductivas en la base de datos.

D

COMP	
pieza1	pieza2
pz1	pz3
pz3	pz8

COMPONENTE	
pieza1	pieza2
pz1	pz3
pz3	pz8
pz1	pz8

Reglas deductivas:

COMPONENTE (x, y) ← COMP (x,z) ∧ COMPONENTE (z, y)
COMPONENTE (x, y) ← COMP(x, y)

Transacción: {insertar (COMP(pz8,pz1))}

D'

COMP	
pieza1	pieza2
pz1	pz3
pz3	pz8
pz8	pz1

inserciones inducidas

COMPONENTE	
pieza1	pieza2
pz1	pz3
pz3	pz8
pz1	pz8
pz8	pz1
pz8	pz3
pz8	pz8
pz1	pz1
pz3	pz1
pz3	pz3

3.2 Comprobación de la integridad

Restricción de integridad:

W: ∀x ∀y (COMPONENTE (x,y) → ¬ COMPONENTE (y,x))

insertar (COMPONENTE(pz8,pz1))

¬ COMPONENTE (pz1, pz8)

Si D es un estado íntegro entonces
D' satisface W si y sólo si D' satisface ¬ COMPONENTE (pz1, pz8) → D' viola W

3.2 Comprobación de la integridad

Enunciado general del problema:

Dados, el esquema (L,RI) de una base de datos deductiva, un estado D de ese esquema tal que D *satisface* W (para toda W∈RI), una transacción T formada por dos conjuntos de sentencias de base de datos, $T=T_{ins} \cup T_{del}$, ($T_{ins} \cap T_{del} = \varnothing$, $T_{del} \subseteq D$ y $T_{ins} \cap D = \varnothing$), el estado D' resultante de aplicar a D la transacción T, ($D' = (D \cup T_{ins}) \setminus T_{del}$), comprobar que D' *satisface* W (para toda W∈RI).

3.2 Comprobación de la integridad

Caracterización del problema:

1) Concepto de satisfacción
2) Corrección y completitud de un método
2) Fases en la comprobación simplificada de la integridad

3.2 Comprobación de la integridad

1) Concepto de satisfacción:

a) Punto de vista de la demostración:

D satisface W sii Tr |= W.

b) Punto de vista de la consistencia:

D satisface W sii Tr ∪ {W} es consistente.

El concepto de violación se define en términos del concepto de satisfacción:

D viola W sii no (D satisface W).

Diremos que un estado D es íntegro si, para toda restricción W perteneciente a RI, D satisface W.

Tr es la teoría de 1er orden que representa la base de datos en la semántica asumida

3.2 Comprobación de la integridad

Ejemplo 1:

D = { p(a) ← q(x) ∧ ¬r(x),
q(a),
r(x) ← r(x) }.

Si asumimos la semántica de la compleción:

Tr=comp(D) = {∀y(p(y) ↔ y=a ∧ ∃x(q(x) ∧ ¬r(x))),
∀ x(q(x) ↔ x=a),
∀ x(r(x) ↔ r(x)) }.

Desde el punto de vista de la consistencia D satisface: W_1=q(a), W_2=r(a), W_3=p(a), W_4=¬r(a).

Desde el punto de vista de la demostración D satisface: W_1=q(a).

Tr es la teoría de 1er orden que representa la base de datos en la semántica asumida

3.2 Comprobación de la integridad

Si asumimos la semántica del punto fijo iterado: M_D={q(a), p(a)}

Tr(M_D) = {∀x (p(x) ↔ x=a),
∀x (q(x) ↔ x=a),
∀x (¬r(x)).

D satisface W_1=q(a) y W_2=p(a) en los dos conceptos de satisfacción.

Tr es la teoría de 1er orden que representa la base de datos en la semántica asumida

3.2 Comprobación de la integridad

2) Corrección y completitud de un método:

Sean:

✓M un método de comprobación de la integridad. D satisface$_M$ (resp. viola$_M$) W significa que el método M decide que el estado D satisface (resp. viola) la restricción W (W∈RI).

✓CS el concepto de satisfacción asumido por el método M. D satisface$_{CS}$ (resp. viola$_{CS}$) W significa que el estado D satisface (resp. viola) la restricción W (W∈RI) en el concepto de satisfacción CS.

Un método M es correcto cuando se cumple:

si D satisface$_M$ W entonces D satisface$_{CS}$ W (correcto para satisfacción)

si D viola$_M$ W entonces D viola$_{CS}$ W (correcto para violación).

Un método M es completo cuando se cumple:

si D satisface$_{CS}$ W entonces D satisface$_M$ W (completo para satisfacción)

si D viola$_{CS}$ W entonces D viola$_M$ W (completo para violación).

3.2 Comprobación de la integridad

3) Fases en la comprobación simplificada de la integridad

Hipótesis: D es íntegro.

Comprobación de la integridad:

FASE I: Fase de Generación

Paso 1: Cálculo del conjuntos de literales que "representan" la diferencia entre los estados consecutivos D y D'.

Paso 2: Identificación de las restricciones relevantes.

Paso 3: Instanciación de las restricciones relevantes.

Paso 4: Simplificación de las instancias de las restricciones relevantes.

FASE II: Fase de Evaluación

Paso 5: Comprobación en D' de las instancias simplificadas de las restricciones relevantes.

3.2 Comprobación de la integridad

3) Fases en la comprobación simplificada de la integridad

Ejemplo 2:

D = { p(x) ← q(x,y) ∧ ¬ t(y),
t(y) ← s(y),
q(a,b), q(b,a), s(c) }

D' = { p(x) ← q(x,y) ∧ ¬ t(y),
t(y) ← s(y),
t(y) ← q(y,z) ∧ t(z)
q(a,b), q(b,a), s(c), s(b) }

T= {ins(s(b), ins(t(y) ← q(y,z) ∧ t(z))}

W_1 = ∀x ∀y (q(x,y) → q(y,x))

W_2 = ∀x (s(x) → p(x))

Fase de Generación

Paso 1: Cálculo del conjuntos de literales que "representan" la diferencia entre los estados consecutivos D y D'.

D = { p(x) ← q(x,y) ∧ ¬ t(y),
t(y) ← s(y),
q(a,b), q(b,a), s(c) }

T

D' = { p(x) ← q(x,y) ∧ ¬ t(y),
t(y) ← s(y),
t(y) ← q(y,z) ∧ t(z)
q(a,b), q(b,a), s(c), s(b) }

T= {ins(s(b), ins(t(y) ← q(y,z) ∧ t(z))}

Actualizaciones inducidas por T:

Inserciones = {A: A∈B_D, comp(D') |= A y comp(D) |≠ A } = {s(b), t(a), t(b)}

Borrados = {A: A∈B_D, comp(D) |= A y comp(D') |≠ A } = {p(a), p(b)}

Se asume la semántica de la compleción y el punto de vista de la demostración.

Fase de Generación

Paso 2: Identificación de restricciones relevantes

D = { p(x) ← q(x,y) ∧ ¬ t(y),
t(y) ← s(y),
q(a,b), q(b,a), s(c) }

T

T={ins(s(b), ins(t(y) ← q(y,z) ∧ t(z))}

D' = { p(x) ← q(x,y) ∧ ¬ t(y),
t(y) ← s(y),
t(y) ← q(y,z) ∧ t(z)
q(a,b), q(b,a), s(c), s(b) }

Inserciones = {s(b), t(a), t(b)}
Borrados = {p(a), p(b)}

W_1 = ∀x ∀y (q(x,y) → q(y,x)) no es relevante para T

Paso 2

W_2 = ∀x (s(x) → p(x)) es relevante para T

Fase de Generación

Paso 3: Instanciación de las restricciones relevantes

Paso 4: Simplificación de las instancias de las restricciones relevantes

D = { p(x) ← q(x,y) ∧ ¬ t(y), t(y) ← s(y), q(a,b), q(b,a), s(c) } —T→ D' = { p(x) ← q(x,y) ∧ ¬ t(y), t(y) ← s(y), t(y) ← q(y,z) ∧ t(z), q(a,b), q(b,a), s(c), s(b) }

Inserciones = {s(b), t(a), t(b)}
Borrados = {p(a), p(b)}

$W_2 = \forall x (s(x) \rightarrow p(x))$

- $W_2\theta_1 = s(b) \rightarrow p(b)$ ⟶ $W_2\theta_1 = p(b)$
- $W_2\theta_2 = s(a) \rightarrow p(a)$ ⟶ $W_2\theta_2 = \neg s(a)$
- $W_2\theta_3 = s(b) \rightarrow p(b)$ ⟶ $W_2\theta_3 = \neg s(b)$

Paso 3 Paso 4

3.2 Comprobación de la integridad

3) Fases en la comprobación simplificada de la integridad

Hipótesis: D es íntegro.

Comprobación de la integridad:

FASE I: Fase de Generación

Paso 1: Cálculo del conjuntos de lit... la diferencia entre los estados consecutivos ...

Paso 2: Identifica...

Paso 3: ...

Paso 4: ...as restricciones relevantes.

FASE II: Fase d...

Paso 5: ...ión en D' de las instancias simplificadas de las restricciones ...evantes.

Paso 1: El cálculo de las actualizaciones inducidas por la transacción puede ser muy costoso en base de datos voluminosas. Sólo algunas de estas actualizaciones serán relevantes para la integridad.

3.2 Comprobación de la integridad

3) Fases en la comprobación simplificada de la integridad

Hipótesis: D es íntegro.

Comprobación de la integridad: Solución

FASE I: Fase de Generación Potencial

Paso 1: Cálculo del conjuntos de literales que "capturen" la diferencia entre los estados consecutivos D y D' sin acceder a la BDE.

Paso 2: Identificación de las restricciones relevantes.

Paso 3: Instanciación de las restricciones relevantes.

Paso 4: Simplificación de las instancias de las restricciones relevantes.

FASE II: Fase de Evaluación

Paso 5: Comprobación en D' de las instancias simplificadas de las restricciones relevantes.

Fase de Generación Potencial

Paso 1: Cálculo del conjuntos de literales que "capturen" la diferencia entre los estados consecutivos D y D'.

D = { p(x) ← q(x,y) ∧ ¬ t(y), t(y) ← s(y), q(a,b), q(b,a), s(c) } —T→ D' = { p(x) ← q(x,y) ∧ ¬ t(y), t(y) ← s(y), t(y) ← q(y,z) ∧ t(z), q(a,b), q(b,a), s(c), s(b) }

T = {ins(s(b), ins(t(y) ← q(y,z) ∧ t(z))}

Actualizaciones potenciales inducidas por T:	
Inserciones	Borrados
{s(b), t(y)}	
{t(b)}	{p(x)}

instancias

Actualizaciones reales inducidas por T:
Inserciones = {s(b), t(a), t(b)}
Borrados = {p(a), p(b)}

Fase de Generación Potencial

Paso 3: Instanciación de las restricciones relevantes

Paso 4: Simplificación de las instancias de las restricciones relevantes

D = { p(x) ← q(x,y) ∧ ¬ t(y), t(y) ← s(y), q(a,b), q(b,a), s(c) } —T→ D' = { p(x) ← q(x,y) ∧ ¬ t(y), t(y) ← s(y), t(y) ← q(y,z) ∧ t(z), q(a,b), q(b,a), s(c), s(b) }

Inserciones = {s(b), t(y), t(b)}
Borrados = {p(x)}

$W_2 = \forall x (s(x) \rightarrow p(x))$

- $W_2\theta_1 = s(b) \rightarrow p(b)$ ⟶
- $W_2\theta_2 = \forall x\, s(x) \rightarrow p(x)$ ⟶ $W_2 = \forall x\, s(x) \rightarrow p(x)$
- $W_2\theta_3 = \forall x\, s(x) \rightarrow p(x)$ ⟶

Paso 3 Paso 4

Fase de Generación Potencial

Paso 3: Instanciación de las restricciones relevantes

Paso 4: Simplificación de las instancias de las restricciones relevantes

D = { p(x) ← q(x,y) ∧ ¬ t(y), t(y) ← s(y), q(a,b), q(b,a), …, s(c) } —T→ D' = { p(x) ← q(x,y) ∧ ¬ t(y), t(y) ← s(y), t(y) ← q(y,z) ∧ t(z), q(a,b), q(b,a), …, s(c), s(b) }

Inserciones = {s(b), t(y), t(b)}
Borrados = {p(x)}

Si la extensión de q es grande los borrados sobre p inducidos por T pueden ser muchos.

$W_3 = \forall x (p(x) \rightarrow r(x))$ no es relevante para T.

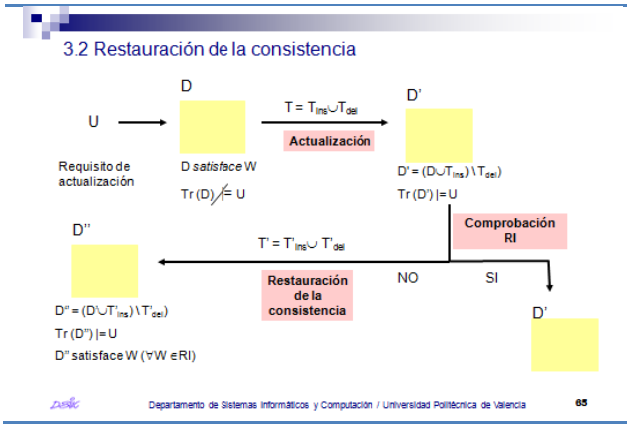
3.2 Comprobación de la integridad

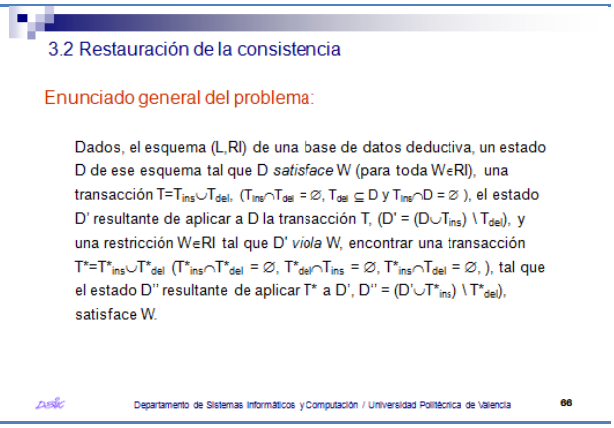
Estudio de un método de simplificación:

1. Semántica asumida: referencia para interpretar el concepto de satisfacción.
2. Concepto de satisfacción utilizado.
3. Requisitos sintácticos: forma sintáctica de las reglas y de las restricciones de integridad.
4. Corrección y completitud del método.
5. Estrategia del método:
   - Fase de Generación: potencial (sin acceso a la BDE), real (con acceso a la BDE).
   - Intercalación de las fases de Generación y Evaluación.
   - Etapa de compilación independiente de la transacción.

La Lógica en el desarrollo de las Bases de Datos

1. Lógica y Bases de Datos.
2. Bases de datos deductivas.
3. Actualización de bases de datos deductivas.
   - 3.1 Actualización
   - 3.2 Comprobación de la integridad
   - 3.3 Restauración de la consistencia





Bases de datos multidimensionales

Son bases de datos ideadas para desarrollar aplicaciones muy concretas. Básicamente almacena sus datos con varias dimensiones, es decir que en vez de un valor, encontramos varios dependiendo de los ejes definidos o una base de datos de estructura basada en dimensiones orientada a consultas complejas y alto rendimiento. En una base de datos multidimensional, la información se representa como matrices multidimensionales, cuadros de múltiples entradas o funciones de varias variables sobre conjuntos finitos. Cada una de estas matrices se denomina cubo. Eso facilita el manejo de grandes cantidades de datos dentro de empresas, dándole a esto una amplia aplicación dentro de varias áreas y diferentes campos del conocimiento humano.

Bases de datos transaccionales

Son bases de datos caracterizadas por su velocidad para gestionar el intercambio de información, se utilizan sobre todo en sistemas bancarios, análisis de calidad y datos de producción industrial. Son bases de datos muy fiables, ya que en ellas cada una de las operaciones de inserción, actualización o borrado se realizan completamente o se descartan.

5.- Tipos de bases de datos.

Caso práctico

__María__ pregunta a __Ada:__ —Si nuestras aplicaciones van a ser accesibles desde Internet ¿Qué tipo de base de datos utilizaremos?

__Ada__ responde: —Pues la respuesta es amplia. Lo normal es que sean bases de datos de acceso múltiple, cuya información cambie en el tiempo, podrán ser centralizadas o distribuidas, además es probable que su acceso deba estar restringido sólo a los usuarios que se indiquen y su temática será diversa. Como ves, hay una gran variedad de tipos de bases de datos y dependiendo de las necesidades y presupuesto de nuestros clientes tendremos que adaptar nuestras aplicaciones a dichos tipos.

—Lo importante es que hagamos un buen diseño y planificación de nuestras bases de datos. De este modo, el software que desarrollemos irá sobre ruedas. — Añade __Juan.__

Como hemos visto, por cada modelo de datos se establecen sustanciales diferencias entre unas bases de datos y otras, pero, ¿Esta es la única clasificación de las bases de datos existente? No, vamos a ver a continuación una detallada descripción de los tipos de bases de datos teniendo en cuenta varios criterios.

Diferentes clasificaciones de las bases de datos

- **Bases de datos según su contenido**
  - **Bases de datos con información actual**: Contienen información muy concreta y actualizada, normalmente, de tipo numérico: estadísticas, series históricas, resultados de encuestas, convocatorias de becas o subvenciones, convocatorias de eventos, ofertas de empleo, …
  - **Directorios**: recogen datos sobre personas o instituciones especializadas en una actividad o materia concreta. Hay directorios de profesionales, de investigadores, de centros de investigación, de bibliotecas, de revistas científicas, de empresas, de editoriales, …
  - **Bases de datos documentales**: En éste último grupo, cada registro se corresponde con un documento, sea éste de cualquier tipo: una publicación impresa, un documento audiovisual, gráfico. Dependiendo de si incluyen o no el contenido completo de los documentos que describen, podremos tener:
    - **Bases de datos de texto completo**: constituidas por los propios documentos en formato electrónico, con un volcado completo de su texto.
    - **Archivos electrónicos de imágenes**: Constituidos por referencias que permiten un enlace directo con la imagen del documento original, sea éste un documento iconográfico (fotografías, imágenes de televisión, …) o un documento impreso digitalizado en formato de imagen.
    - **Bases de datos referenciales**: Sus registros no contienen el texto original sino tan sólo la información fundamental para describir y permitir la localización de documentos impresos, sonoros, iconográficos, audiovisuales o electrónicos. En estos sistemas de información sólo se puede obtener referencias sobre documentos que habrá que localizar posteriormente en otro servisio (archivo, biblioteca, fonoteca,…) o solicitar a un servicio de suministro de documentos.
- **Bases de datos según su uso**:
  - **Base de datos individual**: Es una base de datos utilizada básicamente por una persona. El sistema administrador de la base de datos y los datos son controlados por el mismo usuario. Puede estar almacenada en la unidad de disco duro del usuario o en el servidor de archivos de una red de área local. Por ejemplo, un gerente de ventas podría contar con una base de datos para el control de sus vendedores y su desmpeño.
  - **Base de datos compartida**: Son bases de datos con múltiples usuarios y que muy probablemente pertenezcan a la misma organización, como la base de datos de una compañía. Se encuentra almacenada en una computadora potente y bajo el cuidado de un profesional en el área, el administrador de la base de datos. Los usuarios tienen acceso a la base de datos mediante una red de área local o una red de área extensa.
  - **Bases de datos de acceso público**: Son bases de datos accesibles por cualquier persona. Puede no ser necesario pagar un canon para hacer uso de los datos contenidos en ellas.

- **Bases de datos propietarias o bancos de datos**: Se trata en general de bases de datos de gran tamaño, desarrolladas por una organización y que contienen temas especializados o de carácter particular. El público general puede tener acceso a estas bases a veces de forma gratuita y otras mediante el pago de una cuota. Pueden ofrecer información que va desde negocios, economía, inversión, técnica y científica hasta servicios de entretenimiento. Permiten encontrar en minutos lo que tardaría horas ojeando revistas.

✓ **Bases de datos según la variabilidad de la información**

- **Bases de datos estáticas**: Son bases de datos de sólo lectura. Se utilizan para el almacenamiento de datos históricos que pueden ser analizados y utilizados para el estudio del comportamiento de un conjunto de datos a través del tiempo. Permiten realizar proyecciones y toma de decisiones.
- **Bases de datos dinámicas**: Son bases de datos donde la información almacenada se modifica con el tiempo, permitiendo operaciones como actualización y adición de datos, además de las operaciones fundamentales de consulta.

✓ **Bases de datos según la localización de la información**

- **Bases de datos centralizadas**: Se trata de bases de datos ubicadas en un único lugar, un único computador. Pueden ser bases de datos monousuario que se ejecutan en ordenadores personales o sistemas de bases de datos de alto rendimiento que se ejecutan en grandes sistemas. Este tipo de organización facilita las labores de mantenimiento, sin embargo, hace que la información contenida en dicha base, sea más vulnerable a posibles fallos y limita su acceso. Este tipo de bases de datos puede ofrecer dentro de la arquitectura Cliente/Servidor dos configuraciones:
  - **Basada en anfitrión**: ocurre cuando la máquina cliente y la máquina servidor son la misma. Los usuarios se conectarán directamente a la máquina donde se encuentra la base de datos.
  - **Basada en Cliente/Servidor**: ocurrirá cuando la base de datos reside en una máquina servidor y los resultados acceden a la base de datos desde su máquina cliente a través de una red
- **Bases de datos distribuidas**: Según la naturaleza de la organización es probable que los datos no se almacenen en un único punto, sino que se sitúen en un lugar o lugares diferentes a donde se encuentran los usuarios. Una base de datos distribuida es la unión de las bases de datos mediante redes. Los usuarios se vinculan a los servicios de bases de datos distantes mediante una amplia variedad de redes de comunicación. Puede imaginarse una compañía con diferentes oficinas regionales, donde se encuentra distribuida la base de datos. Si embargo, los ejecutivos pueden tener acceso a la información de todas las oficinas reginales.

✓ **Bases de datos según el organismo productor**

- **Bases de datos de organismos públicos y de la administración**: Las bibliotecas y centros de documentación de los ministerios, instituciones públicas, universidades y organismos públicos de investigación elaboran gran cantidad de recursos de información. Estos sistemas pueden ser:
  - Bases de datos de acceso público, sean gratuitas o no.
  - Bases de datos de uso interno, con información de acceso restringido.
- **Bases de datos de instituciones sin ánimo de lucro**: Fundaciones, asociaciones, sindicatos y organizaciones no gubernamentales elaboran frecuentemente sus propios sistemas de información especializados.
- **Bases de datos de entidades privadas o comerciales**: Los centros de documentación, bibliotecas y archivos de las empresas pueden elaborar distintos tipos de sistemas de información:
  - Bases de datos de uso interno para facilitar la circulación de información dentro de la empresa.
  - Bases de datos de uso interno que ocasionalmente ofrecen servicio hacia el exterior (usuarios particulares u otras instituciones)

- Bases de datos comerciales, diseñadas específicamente para ser utilizadas por usuarios externos.

- **Bases de datos realizadas por cooperación en red**: Se trata de sistemas de información cuya elaboración es compartida por diversas instituciones. Bases de datos internacionales se elaboran a través de este sistema de trabajo, con diversos centros nacionales responsables de la información perteneciente a cada país.

✓ **Bases de datos según el modelo de acceso**:

- **Bases de datos de acceso local**: Para consultarlas es necesario acudir al organismo productor, a su biblioteca o centro de documentación. Pueden ser consultables en monopuesto o en varios puntos de una red local.
- **Bases de datos en CD-ROM**: Pueden adquirirse por compra o suscripción bien directamente por un particular o por una biblioteca o centro de documentación que permita su consulta a sus usuarios. En algunas instituciones se instalan diferentes CD-ROM en una red local para permitir su consulta desde cualquier ordenador conectado a la misma.
- **Bases de datos en línea**: Pueden consultarse desde cualquier ordenador conectado a Internet. La consulta puede ser libre (gratuita) o exigir la solicitud previa de una clave personal entrada (denominada comúnmente con el término inglés *password*). Para obtener un password puede exigir la firma de un contrato. Hay diferentes tipos de acceso en línea:
  - **Acceso via telnet o mediante línea de Internet**: el usuario realiza una conexión estable al host (gran ordenador) en donde se halla la base de datos, a través de Internet. La interfaz de usuario instalada en dicho ordenador remoto determinará si la interrogación debe realizarse por menús o por comandos o expresiones de un lenguaje determinado. Cuando un usuario entra en una base de datos vía telnet establece una sesión de trabajo interactiva con el programa que gestiona la base de datos, que le permite aplicar todas las posibilidades de interrogación que tenga el sistema selección, combinación y visualización o impresión de resultados. En cualquier momento podrá visualizar todas las búsquedas realizadas hasta ese instante y establecer combinaciones entre ellas.
  - **Acceso vía web**: conexión a través de un formulario existente en una página web de Internet, diseñado para lanzar preguntas a una base de datos.

  Una misma base de datos puede tener acceso local y además una edición en CD-ROM y un sistema de acceso en línea. Sin embargo, puede haber diferencias en el contenido presente en cada uno de estos formatos o en el grado de actualización de la información. Por ejemplo, el productor de una base de datos puede ofrecer la conexión en línea a la base de datos completa con actualización diaria y, en cambio, editar un CD_ROM que tan sólo contenga los últimos cinco años de información y se actualice semestralmente.

✓ **Bases de datos según cobertura temática**:

- **Bases de datos científico-tecnológicas**: Contienen información destinada a los investigadores de cualquier ámbito científico o técnico. A su vez, este grupo puede dividirse en:
  - Bases de datos multidisciplinares: abarcan varias disciplinas científicas o técnicas.
  - Bases de datos especializadas: recopilan y analizan documentos pertinentes para una disciplina o subdisciplina concreta: investigación biomédica, farmacéutica, química, agroalimentaria, social, humanística, etc
- **Bases de datos económico-empresariales**: Contienen información de interés para empresas, entidades financieras, …
- **Bases de datos de medios de comunicación**: contienen información de interés para los profesionales de medios de comunicación de masas: prensa, radio, televisión,…
- **Bases de datos de ámbito político-administrativo y jurídico**: Contienen información de interés para los organismos de la administración y los profesionales del Derecho: legislación, jurisprudencia, …

- **Bases de datos de ámbito sanitario**: Además de las propias del primer grupo especializadas en ciencias de la salud, existen otros sistemas con información de interés sanitario: historiales médicos, archivos hospitalarios, …
- **Bases de datos para el gran público**: Contiene información destinada a cubrir necesidades de información general, de interés para un gran número de usuarios.

Las bases de datos en las que sus registros no contienen el texto original sino tan sólo la información fundamental para describir y permitir la localización de documentos impresos, sonoros, iconográficos, audiovisuales o electrónicos, reciben el nombre de:

- ○ Bases de datos documentales.
- ○ Bases de datos distribuidas.
- ◉ **Bases de datos referenciales.**

6 - Sistemas gestores de bases de datos

Caso práctico

__Ada__ explica a __Juan__ y __María__ que la elección de un buen Sistema Gestor de Base de Datos es fundamental. A través de esta herramienta podrán definir, construir y manejar las bases de datos con las que sus aplicaciones informáticas han de trabajar. Conocer sus funciones, componentes y tipos será la base fundamental para llevar a cabo una elección adecuada.

__Juan__ dice: -Yo he utilizado varios sistemas gestores diferentes, cada uno tiene sus ventajas e inconvenientes, pero en general todos se parecen un poco. Eso sí, facilitan mucho el trabajo, son fiables y ahorran tiempo.

Para poder tratar la información contenida en las bases de datos se utilizan los sistemas gestores de bases de datos o SGBD, también llamados DBMS (DataBase Management System), que ofrecen un conjunto de programas que permiten acceder y gestionar dichos datos.
El objetivo fundamental de los SGBD es proporcionar eficiencia y seguridad a la hora de recuperar o insertar información en las bases de datos. Estos sistemas están diseñados para la manipulación de grandes bloques de información.

Sistema Gestor de Base de Datos: Conjunto coordinado de programas, procedimientos, lenguajes, etc., que suministra, tanto a los usuarios no informáticos, como a los analistas programadores, o al administrador, los medios necesarios para describir y manipular los datos contenidos en la base de datos, manteniendo su integridad, confidencialidad y seguridad.

El SGBD permite a los usuarios la creación y el mantenimiento de una base de datos, facilitando la definición, construcción y manipulación de la información contenida en éstas. Definir una base de datos consistirá en especificar los tipos de datos, las estructuras y las restricciones que los datos han de cumplir a la hora de almacenarse en dicha base. Por otro lado, la construcción de la base será el proceso de almacenamiento de datos concretos en algún medio o soporte de almacenamiento que esté supervisado por el SGBD. Finalmente, la manipulación de la base de datos incluirá la posibilidad de realización de consultas para recuperar información específica, la actualización de los datos y la generación de informes a partir de su contenido.

Las ventajas del uso de SGBD son:

- ✓ Proporcionan al usuario una visión abstracta de los datos, ocultando parte de la complejidad relacionada con cómo se almacenan y mantienen los datos.
- ✓ Ofrecen **Independencia física**, es decir, la visión que tiene de la información el usuario, y la manipulación de los datos almacenados en la Base de Datos, es independiente de cómo estén almacenados físicamente.
- ✓ Disminuyen la redundancia y la inconsistencia de datos.
- ✓ Aseguran la integridad de los datos.
- ✓ Facilitan el acceso a los datos, aportando rapidez y evitando la pérdida de datos.
- ✓ Aumentan la seguridad y privacidad de los datos.
- ✓ Mejoran la eficiencia.
- ✓ Permiten compartir datos y accesos concurrentes.
- ✓ Facilitan el intercambio de datos entre distintos sistemas.
- ✓ Incorporan mecanismos de copias de seguridad y recuperación para restablecer la información en caso de fallos en el sistema.

El SGBD interacciona con otros elementos software existentes en el sistema, concretamente con el sistema operativo (SO). Los datos almacenados de forma estructurada en la base de datos son utilizados indistintamente por otras aplicaciones, será el SGBD quien ofrecerá una serie de facilidades

a éstas para el acceso y manipulación de la información, basándose en las funciones y métodos propios del sistema operativo.

6.1.- Funciones.

Un SGBD desarrolla tres funciones fundamentales como son las de descripción, manipulación y utilización de los datos. A continuación se detallan cada una de ellas.

1. **Función de descripción o definición:** Permite al diseñador de la base de datos crear las estructuras apropiadas para integrar adecuadamente los datos. Esta función es la que permite definir las tres estructuras de la base de datos: Estructura interna, Estructura conceptual y Estructura externa. (Estos conceptos se verán más adelante en el epígrafe sobre arquitectura del SGBD).
   Esta función se realiza mediante el **lenguaje de descripción de datos** o **DDL**. Mediante ese lenguaje: se definen las estructuras de datos,se definen las relaciones entre los datos y se definen las reglas (restricciones) que han de cumplir los datos.
   Se especificarán las características de los datos a cada uno de los tres niveles.
   - **A nivel interno** (estructura interna), se ha de indicar el espacio de disco reservado para la base de datos, la longitud de los campos, su modo de representación (lenguaje para la definición de la estructura externa).
   - **A nivel conceptual** (estructura conceptual), se proporcionan herramientas para la definición de las entidades y su identificación, atributos de las mismas, interrelaciones entre ellas, restricciones de integridad, etc.; es decir, el esquema de la base de datos (lenguaje para la definición de estructura lógico global).
   - **A nivel externo** (estructura externa), se deben definir las vistas de los distintos usuarios a través del lenguaje para la definición de estructuras externas. Además, el SGBD se ocupará de la transformación de las estructuras externas orientadas a los usuarios a las estructuras conceptuales y de la relación de ésta y la estructura física.
2. **Función de manipulación:** permite a los usuarios de la base buscar, añadir, suprimir o modificar los datos de la misma, siempre de acuerdo con las especificaciones y las normas de seguridad dictadas por el administrador. Se llevará a cabo por medio de un **lenguaje de manipulación de datos** (**DML**) que facilita los instrumentos necesarios para la realización de estas tareas.
   También se encarga de definir **la vista externa** de todos los usuarios de la base de datos o vistas parciales que cada usuario tiene de los datos definidos con el DDL.
   Por manipulación de datos entenderemos:
   - La recuperación de información almacenada en la base de datos, lo que se conoce como **consultas**.
   - La inserción de información nueva en la base de datos.
   - El borrado de información de la base de datos.
   - La modificación de información almacenada en la base de datos.
3. **Función de control:** permite al administrador de la base de datos establecer mecanismos de protección de las diferentes visiones de los datos asociadas a cada usuario, proporcionando elementos de creación y modificación de dichos usuarios. Adicionalmente, incorpora sistemas para la creación de copias de seguridad, carga de ficheros, auditoría, protección de ataques, configuración del sistema, etc. El lenguaje que implementa esta función es el **lenguaje de control de datos** o **DCL.**

¿Y a través de qué lenguaje podremos desarrollar estas funciones sobre la base de datos? Lo haremos utilizando el **Lenguaje Estructurado de Consultas** (**SQL**: Structured Query Language). Este lenguaje proporciona sentencias para realizar operaciones de DDL, DML y DCL. SQL fue publicado por el ANSI en 1986 (American National Standard Institute) y ha ido evolucionando a lo largo del tiempo. Además, los SGBD suelen proporcionar otras herramientas que complementan a estos lenguajes como generadores de formularios, informes, interfaces gráficas, generadores de aplicaciones, etc.

El DDL de una base de datos sirve para:

- La introducción de los datos en una base de datos.
- **Definir la estructura lógica de la base de datos.**
- Interrogar a la base de datos (consultar la información de dicha base).

6.2.- Componentes.

Una vez descritas las funciones que un SGBD debe llevar a cabo, imaginarás que un SGBD es un paquete de software complejo que ha de proporcionar servicios relacionados con el almacenamiento y la explotación de los datos de forma eficiente. Para ello, cuenta con una serie de componentes que se detallan a continuación:

1. **Lenguajes de la base de datos.** Cualquier sistema gestor de base de datos ofrece la posibilidad de utilizar lenguajes e interfaces adecuadas para sus diferentes tipos de usuarios. A través de los lenguajes se pueden especificar los datos que componen la BD, su estructura, relaciones, reglas de integridad, control de acceso, características físicas y vistas externas de los usuarios. Los lenguajes del SGBD son: Lenguaje de Definición de los Datos (**DDL**), Lenguaje de Manejo de Datos (**DML**) y Lenguaje de Control de Datos (**DCL**).
2. **El diccionario de datos.** Descripción de los datos almacenados. Se trata de información útil para los programadores de aplicaciones. Es el lugar donde se deposita la información sobre la totalidad de los datos que forman la base de datos. Contiene las características lógicas de las estructuras que almacenan los datos, su nombre, descripción, contenido y organización. En una base de datos relacional, el diccionario de datos aportará información sobre:
   - ✓ Estructura lógica y física de la BD.
   - ✓ Definición de tablas, vistas, indices, disparadores, procedimientos, funciones, etc.
   - ✓ Cantidad de espacio asignado y utilizado por los elementos de la BD.
   - ✓ Descripción de las restricciones de integridad.
   - ✓ Información sobre los permisos asociados a cada perfil de usuario.
   - ✓ Auditoría de acceso a los datos, utilización, etc.
3. **El gestor de la base de datos.** Es la parte de software encargada de garantizar el correcto, seguro, íntegro y eficiente acceso y almacenamiento de los datos. Este componente es el encargado de proporcionar una interfaz entre los datos almacenados y los programas de aplicación que los manejan. Es un intermediario entre el usuario y los datos. Es el encargado de garantizar la privacidad, seguridad e integridad de los datos, controlando los accesos concurrentes e interactuando con el sistema operativo.
4. **Usuarios de la base de datos.** En los SGBD existen diferentes perfiles de usuario, cada uno de ellos con una serie de permisos sobre los objetos de la BD. Generalmente existirán:
   - ✓ El **administrador de la base de datos** o Database Administrator (DBA), que será la persona o conjunto de ellas encargadas de la función de administración de la base de datos. Tiene el control centralizado de la base de datos y es el responsable de su buen funcionamiento. Es el encargado de autorizar el acceso a la base de datos, de coordinar y vigilar su utilización y de adquirir los recursos software y hardware que sean necesarios.
   - ✓ Los **usuarios de la base de datos**, que serán diferentes usuarios de la BD con diferentes necesidades sobre los datos, así como diferentes accesos y privilegios. Podemos establecer la siguiente clasificación:
     - ➔ Diseñadores.
     - ➔ Operadores y personal de mantenimiento.
     - ➔ Analistas y programadores de aplicaciones.
     - ➔ Usuarios finales: ocasionales, simples, avanzados y autónomos.
5. **Herramientas de la base de datos.** Son un conjunto de aplicaciones que permiten a los administradores la gestión de la base de datos, de los usuarios y permisos, generadores de formularios, informes, interfaces gráficas, generadores de aplicaciones, etc.

6.3.- Arquitectura.

Un SGBD cuenta con una arquitectura a través de la que se simplifica a los diferentes usuarios de la base de datos su labor. El objetivo fundamental es separar los programas de aplicación de la base de datos física.

Encontrar un estándar para esta arquitectura no es una tarea sencilla, aunque los tres estándares que más importancia han cobrado 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). Tanto ANSI (EEUU), como ISO (Resto del mundo), son el referente en cuanto a estandarización de bases de datos, conformando un único modelo de bases de datos.

La arquitectura propuesta proporciona tres niveles de abstracción: **nivel interno o físico, nivel lógico o conceptual y nivel externo o de visión del usuario**. A continuación se detallan las características de cada uno de ellos:

- ✓ **Nivel interno o físico:** En este nivel se describe la estructura física de la base de datos a través de un esquema interno encargado de detallar el sistema de almacenamiento de la base de datos y sus métodos de acceso. Es el nivel más cercano al almacenamiento físico. A través del esquema físico se indican, entre otros, los archivos que contienen la información, su organización, los métodos de acceso a los registros, los tipos de registros, la longitud, los campos que los componen, las unidades de almacenamiento, etc.
- ✓ **Nivel lógico o conceptual:** En este nivel se describe la estructura completa de la base de datos a través de un esquema que detalla las entidades, atributos, relaciones, operaciones de los usuarios y restricciones. Los detalles relacionados con las estructuras de almacenamiento se ocultan, permitiendo realizar una abstracción a más alto nivel.
- ✓ **Nivel externo o de visión del usuario:** En este nivel se describen las diferentes vistas que los usuarios percibirán de la base de datos. Cada tipo de usuario o grupo de ellos verá sólo la parte de la base de datos que le interesa, ocultando el resto.

Para una base de datos, sólo existirá un único esquema interno, un único esquema conceptual y podrían existir varios esquemas externos definidos para uno o varios usuarios.
Gracias a esta arquitectura se consigue la **independencia de datos** a dos niveles:

- ✓ **Independencia lógica:** Podemos modificar el esquema conceptual sin alterar los esquemas externos ni los programas de aplicación.
- ✓ **Independencia física:** Podemos modificar el esquema interno sin necesidad de modificar el conceptual o el externo. Es decir, se puede cambiar el sistema de almacenamiento, reorganizar los ficheros, añadir nuevos, etc., sin que esto afecte al resto de esquemas.

En el siguiente gráfico se puede apreciar la estructura de la que estamos hablando:

El esquema conceptual de la totalidad de la base de datos puede obtenerse de la unión de todos los esquemas externos definidos para cada usuario de la base de datos.

Verdadero ⚽ Falso ☐

. .

6.4.- Tipos.

¿Qué tipos de SGBD existen? Para responder a esta pregunta podemos realizar la siguiente clasificación, atendiendo a diferentes criterios:

a. El primer criterio que se suele utilizar es **por el modelo lógico en que se basan**. Actualmente, el modelo lógico que más se utiliza es el **relacional**. Los modelos en red y jerárquico han quedado obsoletos. Otro de los modelos que más extensión está teniendo es el modelo **orientado a objetos**. Por tanto, en esta primera clasificación tendremos:
   - ✓ Modelo Jerárquico.
   - ✓ Modelo de Red.
   - ✓ Modelo Relacional.
   - ✓ Modelo Orientado a Objetos.

   (Para recordar los modelos de bases de datos vistos, sitúate en el epígrafe 4 de esta Unidad de Trabajo y analiza su contenido.)
b. El segundo criterio de clasificación se centra en el **número de usuarios** a los que da servicio el sistema:
   - ✓ **Monousuario**: sólo atienden a un usuario a la vez, y su principal uso se da en los ordenadores personales.
   - ✓ **Multiusuario**: entre los que se encuentran la mayor parte de los SGBD, atienden a varios usuarios al mismo tiempo.
c. El tercer criterio se basa en el **número de sitios en los que está distribuida la base de datos:**
   - ✓ **Centralizados:** sus datos se almacenan en un solo computador. Los SGBD centralizados pueden atender a varios usuarios, pero el SGBD y la base de datos en sí residen por completo en una sola máquina.
   - ✓ **Distribuidos (Homogéneos, Heterogéneos): la base de datos real y el propio software del SGBD pueden estar distribuidos en varios sitios conectados por una red. Los sistemas homogéneos utilizan el mismo SGBD en múltiples sitios. Una tendencia reciente consiste en crear software para tener acceso a varias bases de datos autónomas preexistentes almacenadas en sistemas distribuidos heterogéneos. Esto da lugar a los SGBD federados o sistemas multibase de datos en los que los SGBD participantes tienen cierto grado de autonomía local.**
d. **El cuarto criterio toma como referencia el coste**. La mayor parte de los paquetes cuestan entre 10.000 y 100.000 euros. Los sistemas monousuario más económicos para microcomputadores cuestan entre 0 y 3.000 euros. En el otro extremo, los paquetes más completos cuestan más de 100.000 euros.
e. El quinto, y último, criterio establece su clasificación **según el propósito**:
   - ✓ **Propósito General:** pueden ser utilizados para el tratamiento de cualquier tipo de base de datos y aplicación.
   - ✓ **Propósito Específico:** Cuando el rendimiento es fundamental, se puede diseñar y construir un software de propósito especial para una aplicación específica, y este sistema no sirve para otras aplicaciones. Muchos sistemas de reservas de líneas aéreas son de propósito especial y pertenecen a la categoría de **sistemas de procesamiento de transacciones en línea**, que deben atender un gran número de transacciones concurrentes sin imponer excesivos retrasos.

7.- SGBD comerciales.

Caso práctico

—¿Conocéis la multinacional Oracle? ¿Y su sistema de gestión de bases de datos Oracle 10g? —Pregunta ***Ada.***

Juan, *que está terminando de instalar un nuevo disco duro en su equipo le responde: —Por supuesto* ***Ada****, es el número uno en el mundo de las bases de datos y sus productos tienen una gran aceptación en el mercado. Según conozco, sus posibilidades y fiabilidad son impresionantes, aunque hay que pagar una licencia.*

BK Programación ha de tener en cuenta que sus aplicaciones deben estar sustentadas en un sistema que ofrezca garantías, pero que se ajuste a sus necesidades, dimensiones y presupuesto. Es el momento de pensar su próxima jugada.

Actualmente, en el mercado de software existen multitud de sistemas gestores de bases de datos comerciales. En este epígrafe se desglosan las características fundamentales de los más importantes y extendidos hasta la fecha. Pero, como podrás observar, la elección de un SGBD es una decisión muy importante a la hora de desarrollar proyectos. A veces, el sistema más avanzado, "el mejor" según los entendidos, puede no serlo para el tipo de proyecto que estemos desarrollando. Hemos de tener en cuenta qué volumen de carga debe soportar la base de datos, qué sistema operativo utilizaremos como soporte, cuál es nuestro presupuesto, plazos de entrega, etc.

A través de la siguiente tabla se exponen los SGBD comerciales más utilizados y sus características más relevantes:

Sistemas Gestores de Bases de Datos Comerciales.		
SGBD	**Descripción**	**URL**
ORACLE	Reconocido como uno de los mejores a nivel mundial. Es multiplataforma, confiable y seguro. Es Cliente/Servidor. Basado en el modelo de datos Relacional. De gran potencia, aunque con un precio elevado hace que sólo se vea en empresas muy grandes y multinacionales. Ofrece una versión gratuita **Oracle Database 10*g* Express Edition.**	http://www.oracle.com/us/products/database/product-editions-066501.html?ssSourceSiteId=ocomes
MYSQL	Sistema muy extendido que se ofrece bajo dos tipos de licencia, comercial o libre. Para aquellas empresas que deseen incorporarlo en productos privativos, deben comprar una licencia específica. Es Relacional, Multihilo, Multiusuario y Multiplataforma. Su gran velocidad lo hace ideal para consulta de bases de datos y plataformas web.	http://www.mysql.com/
DB2	Multiplataforma, el motor de base de datos relacional integra XML de manera nativa, lo que IBM ha llamado pureXML, que permite almacenar documentos completos para realizar operaciones y búsquedas de manera jerárquica dentro de éste, e integrarlo con búsquedas relacionales.	http://www.ibm.com/developerworks/ssa/downloads/im/udbexp/
INFORMIX	Otra opción de IBM para el mundo empresarial que necesita un DBMS sencillo y confiable. Es un gestor de base de datos relacional basado en SQL. Multiplataforma. Consume menos recursos que Oracle, con utilidades muy avanzadas respecto a conectividad y funciones relacionadas con tecnologías de Internet/Intranet, XML, etc.	http://www-01.ibm.com/software/es/data/informix/discover-informix/index.html
Microsoft SQL SERVER	Sistema Gestor de Base de Datos producido por Microsoft. Es relacional, sólo funciona bajo Microsoft Windows, utiliza arquitectura Cliente/Servidor. Constituye la alternativa a	http://www.microsoft.com/spain/sql/2008/overview.aspx

	otros potentes SGBD como son Oracle, PostgreSQL o MySQL.	
SYBASE	Un DBMS con bastantes años en el mercado, tiene 3 versiones para ajustarse a las necesidades reales de cada empresa. Es un sistema relacional, altamente escalable, de alto rendimiento, con soporte a grandes volúmenes de datos, transacciones y usuarios, y de bajo costo.	http://www.sybase.es/products/databasemanagement/adaptiveserverenterprise

Otros SGBD comerciales importantes son: DBASE, ACCESS, INTERBASE y FOXPRO.

Puedes completar más información sobre estos y otros sistemas a través de los enlaces que te proponemos a continuación:

http://es.wikipedia.org/wiki/Sistema_de_gesti%C3%B3n_de_bases_de_datos#SGBD_no_libres

http://4.bp.blogspot.com/_BF5cCSiGL2M/TUjX04ZuzgI/AAAAAAAAAIo/2_0gKVhrpO0/s1600/gartnerMQ_2011.bmp

8.- SGBD libres.

Caso práctico

__Juan__, que tiene especial debilidad por el software libre, comenta que existen alternativas muy potentes a coste cero. __Ada,__ agradece la información que __Juan__ aporta e indica que también tendrán en cuenta los sistemas gestores de bases de datos libres en sus desarrollos, ya que algunos de ellos están ampliamente extendidos y ofrecen importantes ventajas. __María,__ que ha trabajado alguna vez con MySQL, está deseosa de aprender nuevos sistemas gestores ya sean comerciales o libres.

La alternativa a los sistemas gestores de bases de datos comerciales la encontramos en los SGBD de código abierto o libres, también llamados Open Source. Son sistemas distribuidos y desarrollados libremente. En la siguiente tabla se relacionan los cinco más utilizados actualmente, así como sus principales características y enlaces a sus páginas web:

Sistemas Gestores de Bases de Datos Libres.		
SGBD	**Descripción**	**URL**
MySQL	Es un sistema de gestión de base de datos relacional, multihilo y multiusuario con más de seis millones de instalaciones. Distribuido bajo dos tipos de licencias, comercial y libre. Multiplataforma, posee varios motores de almacenamiento, accesible a través de múltiples lenguajes de programación y muy ligado a aplicaciones web.	http://www.mysql.com/
PostgreSQL	Sistema Relacional Orientado a Objetos. Considerado como la base de datos de código abierto más avanzada del mundo. Desarrollado por una comunidad de desarrolladores que trabajan de forma desinteresada, altruista, libre y/o apoyados por organizaciones comerciales. Es multiplataforma y accesible desde múltiples lenguajes de programación.	http://www.postgresql.org/
Firebird	Sistema Gestor de Base de Datos relacional, multiplataforma, con bajo consumo de recursos, excelente gestión de la concurrencia, alto rendimiento y potente soporte para diferentes lenguajes.	http://www.firebirdsql.org/
Apache Derby	Sistema Gestor escrito en Java, de reducido tamaño, con soporte multilenguaje, multiplataforma, altamente portable, puede funcionar embebido o en modo cliente/servidor.	http://db.apache.org/derby/
SQLite	Sistema relacional, basado en una biblioteca escrita en C que interactua directamente con los programas, reduce los tiempos de acceso siendo más rápido que MySQL o PostGreSQL, es multiplataforma y con soporte para varios lenguajes de programación.	http://www.sqlite.org/

El tamaño máximo de una tabla en PostGreSQL es de 1,6 Terabytes.

Verdadero ◉ Falso ○

9.- Bases de datos centralizadas.

Caso práctico

__Ada, Juan__ y __María__ están visitando un centro de cómputo cercano a BK Programación. La estructura del sistema informático está centralizada y limita las posibilidades de uso de la información contenida en dicho sistema. __Ada__ indica que con la ayuda de la tecnología de redes de computadoras la información se puede mantener localizada en diversos lugares, permitiendo accesos más rápidos y múltiples ventajas adicionales en comparación con los sistemas centralizados. Los tres continúan su visita, analizando las ventajas e inconvenientes del sistema centralizado que están viendo.

Si nos preguntamos cómo es la arquitectura de un sistema de base de datos, hemos de saber que todo depende del sistema informático que la sustenta. Tradicionalmente, la arquitectura centralizada fue la que se utilizó inicialmente, aunque hoy en día es de las menos utilizadas.

Sistema de base de datos centralizado: Es aquella estructura en la que el SGBD está implantado en una sola plataforma u ordenador desde donde se gestiona directamente, de modo centralizado, la totalidad de los recursos. Es la arquitectura de los centros de proceso de datos tradicionales. Se basa en tecnologías sencillas, muy experimentadas y de gran robustez.

Los sistemas de los años sesenta y setenta eran totalmente centralizados, como corresponde a los sistemas operativos de aquellos años, y al hardware para el que estaban hechos: un gran ordenador para toda la empresa y una red de terminales sin inteligencia ni memoria.

Las principales características de las bases de datos centralizadas son:

- ✓ Se almacena completamente en una ubicación central, es decir, todos los componentes del sistema residen en un solo computador o sitio.
- ✓ No posee múltiples elementos de procesamiento ni mecanismos de intercomunicación como las bases de datos distribuidas.
- ✓ Los componentes de las bases de datos centralizadas son: los datos, el software de gestión de bases de datos y los dispositivos de almacenamiento secundario asociados.
- ✓ Son sistemas en los que su seguridad puede verse comprometida más fácilmente.

En la siguiente tabla se representan las ventajas e inconvenientes destacables de esta arquitectura de bases de datos.

Ventajas e inconvenientes de las bases de datos centralizadas.	
Ventajas	**Inconvenientes**
Se evita la redundancia debido a la posibilidad de inconsistencias y al desperdicio de espacio.	Un mainframe en comparación de un sistema distribuido no tiene mayor poder de cómputo.
Se evita la inconsistencia. Ya que si un hecho específico se representa por una sola entrada, la no-concordancia de datos no puede ocurrir.	Cuando un sistema de bases de datos centralizado falla, se pierde toda disponibilidad de procesamiento y sobre todo de información confiada al sistema.
La seguridad se centraliza.	En caso de un desastre o catástrofe, la recuperación es difícil de sincronizar.
Puede conservarse la integridad.	Las cargas de trabajo no se pueden difundir entre varias computadoras, ya que los trabajos siempre se ejecutarán en la misma máquina.
El procesamiento de los datos ofrece un mejor rendimiento.	Los departamentos de sistemas retienen el control de toda la organización.
Mantenimiento más barato. Mejor uso de los recursos y menores recursos humanos.	Los sistemas centralizados requieren un mantenimiento central de datos.

10.- Bases de datos distribuidas.

Caso práctico

*Para poder apreciar la diferencia, **Ada** ha organizado una vídeoconferencia en la que intervienen dos técnicos de bases de datos y un gerente de una gran cadena hotelera, amigos suyos. Cada uno de ellos se encuentra en sedes diferentes dispersas geográficamente. **Juan** y **María**, permanecen atentos a las intervenciones que se realizan y toman buena nota de las valoraciones de los sistemas de bases de datos distribuidos hechas por los conferenciantes.*

La necesidad de integrar información de varias fuentes y la evolución de las tecnologías de comunicaciones, han producido cambios muy importantes en los sistemas de bases de datos. La respuesta a estas nuevas necesidades y evoluciones se materializa en los sistemas de bases de datos distribuidas.

Base de datos distribuida (BDD): es un conjunto de múltiples bases de datos lógicamente relacionadas las cuales se encuentran distribuidas entre diferentes nodos interconectados por una red de comunicaciones.

Sistema de bases de datos distribuida (SBDD): es un sistema en el cual múltiples sitios de bases de datos están ligados por un sistema de comunicaciones, de tal forma que, un usuario en cualquier sitio puede acceder los datos en cualquier parte de la red exactamente como si los datos estuvieran almacenados en su sitio propio.

Sistema gestor de bases de datos distribuida (SGBDD): es aquel que se encarga del manejo de la BDD y proporciona un mecanismo de acceso que hace que la distribución sea transparente a los usuarios. El término transparente significa que la aplicación trabajaría, desde un punto de vista lógico, como si un solo SGBD ejecutado en una sola máquina, administrara esos datos.

Un SGBDD desarrollará su trabajo a través de un conjunto de sitios o nodos, que poseen un sistema de procesamiento de datos completo con una base de datos local, un sistema de gestor de bases de datos e interconcetados entre sí. Si estos nodos están dispersos geográficamente se internocetarán a través de una red de área amplia o WAN, pero si se encuentran en edificios relativamente cercanos, pueden estar interconectados por una red de área local o LAN. Este tipo de sistemas es utilizado en: organizaciones con estructura descentralizada, industrias de manufactura con múltiples sedes (automoción), aplicaciones militares, lineas aéreas, cadenas hoteleras, servicios bancarios, etc.

En la siguiente tabla se representan las ventajas e inconvenientes destacables de las BDD:

Ventajas e inconvenientes de las bases de datos distribuidas.	
Ventajas	**Inconvenientes**
El acceso y procesamiento de los datos es más rápido ya que varios nodos comparten carga de trabajo.	La probabilidad de violaciones de seguridad es creciente si no se toman las precauciones debidas.
Desde una ubicación puede accederse a información alojada en diferentes lugares.	Existe una complejidad añadida que es necesaria para garantizar la coordinación apropiada entre los nodos.
Los costes son inferiores a los de las bases	La inversión inicial es menor, pero el

centralizadas.	mantenimiento y control puede resultar costoso.
Existe cierta tolerancia a fallos. Mediante la replicación, si un nodo deja de funcionar el sistema completo no deja de funcionar.	Dado que los datos pueden estar replicados, el control de concurrencia y los mecanismos de recuperación son mucho más complejos que en un sistema centralizado.
El enfoque distribuido de las bases de datos se adapta más naturalmente a la estructura de las organizaciones. Permiten la incorporación de nodos de forma flexible y fácil.	El intercambio de mensajes y el cómputo adicional necesario para conseguir la coordinación entre los distintos nodos constituyen una forma de sobrecarga que no surge en los sistemas centralizados.
Aunque los nodos están interconectados, tienen independencia local.	Dada la complejidad del procesamiento entre nodos es difícil asegurar la corrección de los algoritmos, el funcionamiento correcto durante un fallo o la recuperación.

Si deseas completar más información sobre las bases de datos distribuidas, puedes hacerlo a través del siguiente documento:
http://sinbad.dit.upm.es/docencia/grado/curso0910/Tema%20VII%20Arquitecturas%20SGBD%20Distribuidos/2009-10%20Docu%20Todo%20el%20Tema%20VII%20BSDT.pdf

10.1.- Fragmentación.

Sabemos que en los sistemas de bases de datos distribuidas la información se encuentra repartida en varios lugares. La forma de extraer los datos consultados puede realizarse mediante la fragmentación de distintas tablas pertenecientes a distintas bases de datos que se encuentran en diferentes servidores. El problema de fragmentación se refiere al particionamiento de la información para distribuir cada parte a los diferentes sitios de la red.

Pero hay que tener en cuenta el **grado de fragmentación** que se aplicará, ya que éste es un factor determinamente a la hora de la ejecución de consultas. Si no existe fragmentación, se tomarán las relaciones o tablas como la unidad de fragmentación. Pero también puede fragmentarse a nivel de tupla (fila o registro) o a nivel de atributo (columna o campo) de una tabla. No será adecuado un grado de fragmentación nulo, ni tampoco un grado de fragmentación demasiado alto. El grado de fragmentación deberá estar equilibrado y dependerá de las particularidades de las aplicaciones que utilicen dicha base de datos. Concretando, el objetivo de la fragmentación es encontrar un nivel de particionamiento adecuado en el rango que va desde tuplas o atributos hasta relaciones completas.

Cuando se lleva a cabo una fragmentación, existen tres reglas fundamentales a cumplir:

- ✓ **Completitud.** Si una relación R se descompone en fragmentos R1, R2, ..., Rn, cada elemento de datos que pueda encontrarse en R deberá poder encontrarse en uno o varios fragmentos Ri.
- ✓ **Reconstrucción.** Si una relación R se descompone en una serie de fragmentos R1, R2, ..., Rn, la reconstrucción de la relación a partir de sus fragmentos asegura que se preservan las restricciones definidas sobre los datos.
- ✓ **Disyunción.** Si una relación R se descompone verticalmente, sus atributos primarios clave normalmente se repiten en todos sus fragmentos.

Existen tres tipos de fragmentación:

- ✓ **Fragmentación horizontal:** La fragmentación horizontal se realiza sobre las tuplas de la relación, dividiendo la relación en subrelaciones que contienen un subconjunto de las tuplas que alberga la primera. Existen dos variantes de la fragmentación horizontal: la primaria y la derivada.
- ✓ **Fragmentación vertical:** La fragmentación vertical, en cambio, se basa en los atributos de la relación para efectuar la división. Una relación R produce fragmentos R1, R2, ..., Rr, cada uno de los cuales contiene un subconjunto de los atributos de R así como la llave primaria de R. El

objetivo de la fragmentación vertical es particionar una relación en un conjunto de relaciones más pequeñas de manera que varias de las aplicaciones de usuario se ejecutarán sobre un fragmento. En este contexto, una fragmentación óptima es aquella que produce un esquema de fragmentación que minimiza el tiempo de ejecución de las consultas de usuario. La fragmentación vertical es más complicada que la horizontal, ya que existe un gran número de alternativas para realizarla.

- ✓ **Fragmentación Híbrida o mixta:** Podemos combinar ambas, utilizando por ello la denominada fragmentación mixta. Si tras una fragmentación vertical se lleva a cabo otra horizontal, se habla de la fragmentación mixta (HV). Para el caso contrario, estaremos ante una fragmentación (VH). Para representar los dos tipos de fragmentación, se utilizan los árboles.

Una base de datos almacenada entre distintos computadores conectados en red, de forma que unos tienen acceso a los datos de otros, se dice que:

- ○ Utiliza un modelo jerárquico
- ◉ **Es de tipo distribuido con fragmentación**
- ○ Utiliza un modelo en red

11.- Primeros pasos en Oracle Database 10g Express Edition.

Caso práctico

*Después de valorar todas las opciones (comerciales y libres) existentes en el mercado, BK Programación se decantará por un consagrado sistema de base de datos comercial, pero en su versión gratuita. Será **Oracle Database 10g Express Edition**, que ofrece ser completamente gratuito para desarrollar y distribuir los desarrollos de la empresa, está disponible para Microsoft Windows y Linux, puede ser actualizado a versiones superiores de Oracle 10g y permite trabajar con diferentes lenguajes de programación.*

***Juan** y **María** están muy interesados en aprender a manejar este sistema, saben que Oracle es una de las herramientas más potentes en el mundo de las bases de datos y están dispuestos a afrontar el reto.*

¿Qué es Oracle Database 10g Express Edition?

Es un sistema de bases de datos libre para el desarrollo, implementación y distribución. Es un sistema para la iniciación, con un consumo reducido de recursos, basado en el producto Oracle Database 10g revisión 2. Su descarga es rápida y brinda un sistema de administración sencillo. Es un buen sistema de iniciación para desarrolladores en PHP, Java, XML y aplicaciones de código abierto, para administradores de bases de datos que necesitan una base de datos para su adiestramiento e implementación, para proveedores independientes de software o hardware que desean una base de datos inicial para distribuir libre de costes sus productos o para instituciones educativas o estudiantes que necesitan una base de datos libre con la que completar su curriculum.

Si quieres conocer más características destacables de este sistema de bases de datos, aquí puedes acceder a la hoja de especificación de Oracle 10g Express Edition (en Inglés).

http://www.oracle.com/technetwork/database/express-edition/overview/dbxe-datasheet-130365.pdf

¿Por dónde empezamos?

El primer paso que debemos dar es descargar el software necesario desde la página oficial de Oracle. A través del siguiente enlace podrás acceder a la zona de descarga de Oracle Database 10g Express Edition, regístrate, escoge el que se ajuste a tus necesidades y descárgalo en tu ordenador.

http://www.oracle.com/technetwork/database/express-edition/downloads/index.html

¿Cómo se realiza la instalación?

Para llevar a cabo la instalación del software descargado, dependiendo de tu sistema operativo, puedes visualizar alguno de los vídeos que te proponemos a continuación:

Instalación de Oracle Database 10g Express Edition bajo Windows 7

El vídeo comienza con los datos de la autora y la universidad a la que pertenece. A continuación, se accede al escritorio de un equipo con sistema operativo Windows 7, en el que ya se encuentra descargado el instalador de Oracle Database 10g Express Edition. Haciendo doble clic sobre dicho instalador, se inicia el asistente de instalación con una barra de progreso de color verde que se va completando. Una vez ha terminado de prepararse el instalador, aparece en pantalla una ventana de bienvenida del producto, se pulsa en el botón de siguiente y aparece el texto de aceptación de licencia de uso. Hay que seleccionar que se aceptan las condiciones y se pulsa en siguiente. Seguidamente, se solicita si se desea cambiar la ubicación de la instalación, en el vídeo se ha dejado la ubicación por defecto. En la siguiente ventana se solicita que introduzcamos una contraseña para la cuenta SYS y SYSTEM para la base de datos. Introducimos la contraseña elegida y se pulsa en siguiente. Se presenta ahora un resumen de lo que se va a instalar y dónde, para que lo confirmemos. Pulsamos en siguiente. El proceso de instalación se inicia y aparece una barra de progreso que se va completando. Una vez completada la instalación, el asistente nos pregunta si queremos iniciar la página inicial de gestión de la base de datos. Se abre el navegador web y aparece un formulario de login, en el que se introduce el nombre de usuario SYSTEM y la contraseña la que se

definió anteriormente. Una vez introducidos se pulsa en conectar y tras unos instantes, aparece un interfaz web para la gestión de la base de datos.

Instalación de Oracle Database 10g Express Edition bajo Ubuntu Linux.

Inicialmente, se muestran las librerías y requisitos necesarios para poder llevar a cabo la instalación en un equipo con sistema operativo Ubuntu. Se indica, a continuación, que se realice la descarga del paquete de instalación desde la página de Oracle, que se dejen los puertos por defecto de la instalación y que se realice un login con el usuario y contraseña establecidos durante este proceso. Una vez hechas esta indicaciones, se muestra una ventana del explorados Nautilus en el que aparece el paquete de instalación con extensión ***.deb*** Al hacer doble clic sobre él, se inicia el instalador de paquetes de Ubuntu. Durante el proceso de instalación se solicita la contraseña de administrador del sistema. Una vez completada la instalación, se muestra en pantalla el comando para realizar la configuración de esta herramienta. Al teclear en una terminal dicho comando, se abre un asistente de configuración en modo texto en el que se van indicando diferentes parámetros de configuración: puerto http, puerto de escucha de la base de datos, contraseña del usuario SYSTEM y SYS y establecimiento de Oracle como aplicación que se inicia por defecto en el arranque. Tras unos instantes, la configuración se lleva a cabo y el interfaz de consola de comandos indica la dirección web que ha de insertarse en el navegador web para acceder al interfaz web de Oracle 10g Express Edition. Se carga en el navegador web dicha dirección, se abre sesión con el usuario SYSTEM y finalmente, el vídeo termina recordando que los requisitos iniciales es importante cumplirlos.

Gestión básica de datos en Oracle Database 10g Express Edition

A partir del inicio de sesión con el usuario SYSTEM, se accede al interfaz web de la base de datos. Se inicia el recorrido por el interfaz accediendo al explorador de objetos. Al pulsar sobre él aparece un menú de acciones, se selecciona crear y dentro de este submenú, se selecciona tabla. Se crea una tabla: carne, nombre, dirección y teléfono. A continuación, se pulsa en siguiente y se solicita cuál es el campo clave primaria de la tabla. En este caso, no se establece clave. Tampoco claves foráneas, ni restricciones. Se pulsa sobre el borón de crear y se visualiza la tabla y sus características en pantalla. El siguiente elemento del interfaz web que se analiza es el titulado SQL, al pulsar sobre su icono se muestra una consola de comandos SQL. A través de esta consola se realiza un ejemplo de inserción en la tabla mediante comandos. Una vez preparado el comando, se pulsa sobre ejecutar y se muestra el resultado de la ejecución de dicho comando de inserción. Para comprobar la inserción, a través de la misma consola, se lanza una consulta de datos y se obtiene el resultado que confirma la inserción, a través de la misma consola, se lanza una consulta de datos y se obtiene el resultado que confirma la inserción. Posteriormente, se ejecuta un comando de modificación de datos y una consulta asociada para confirmar la modificación. Por último, se lleva a cabo un borrado de datos mediante línea de comandos y consulta asociada para ver los resultados.

Administración simple de usuarios en Oracle Database 10g Express Edition

Utilizando el navegador web, en el interfaz web de la aplicación, se posiciona sobre el menú de administración. Se despliega un submenú en el que se selecciona usuarios de la base de datos y en su interior, gestión de usuarios. Seguidamente, se visualiza un único usuario con nombre HR que está bloqueado y cuya cuenta ha expirado. Se pulsa sobre el usuario y se muestra un formulario en el que pueden modificarse los datos básicos de dicho usuario, su clave, desbloqueo y privilegios. A continuación, una vez desbloqueado el usuario HR y asignada una nueva clave, se cierra sesión con el usuario SYSTEM y se entra con el usuario HR. En el mismo interfaz web inicial, se accede al explorador de objetos y puede verse que el usuario puede crear tablas u otros objetos, existiendo diferentes modelos para utilizar como base. Se cierra sesión con el usuario HR y se cierra el navegador. Ahora, en el escritorio del equipo se selecciona el icono de Equipo y se pulsa con botón derecho. Se selecciona la opción administrar y en servicios y aplicaciones, selecciona servicios y busca por orden alfabético los servicios asociados a Oracle. Encuentra los servicios OracleServiceXE y OracleXETSNListener, los selecciona con doble clic y hace que no se inicien de forma automática al

iniciarse Windows 7. De este modo consigue que el arranque de Windows sea mucho más rápido al no iniciarse por defecto estos servicios.

TEMA 2

INDICE

1.- Modelo de datos. 3
2.- Terminología del modelo relacional 4
2.1.- Relación o tabla. Tuplas. Dominios. 4
2.3.- Sinónimos 6
3.- Relaciones. Características de una relación (tabla). 7
3.1.- Tipos de relaciones (tablas). 7
4.- Tipos de datos. 9
5.- Claves. 10
5.1.- Clave candidata. Clave primaria. Clave alternativa. 10
5.2.- Clave externa, ajena o secundaria. 11
6.- Índices. Características. 13
7.- El valor NULL. Operaciones con este valor. 14
8.- Vistas. 15
9.- Usuarios. Roles. Privilegios 16
10.- SQL. 17
10.1.- Elementos del lenguaje. Normas de escritura. 17
Elementos del lenguaje SQL. 18
Instalación de Oracle XE. 20
11.- Lenguaje de descripción de datos (DDL). 23
11.1.- Creación de bases de datos. Objetos de la base de datos. 23
11.2.- Creación de tablas. 24
11.3.- Restricciones. 25
11.3.1.- Restricción NOT NULL. 26
11.3.2.- Restricción UNIQUE. 27
11.3.3.- Restricción PRIMARY KEY. 27
11.3.4.- Restricción REFERENCES. FOREIGN KEY. 28
11.3.5.- Restricción DEFAULT Y VALIDACIÓN. 29
11.4.- Eliminación de tablas. 29
11.5.- Modificación de tablas (I). 30
Ejercicio resuelto 31
11.5.1.- Modificación de tablas (II). 31
11.6.- Creación y eliminación de índices 32
Ejercicio resuelto 32
12.- Lenguaje de control de datos (DCL). 33
12.1.- Permisos (I). 34
12.1.1.- Permisos (II) 35

BASES DE DATOS RELACIONALES

CASO PRÁCTICO.
Ada ha asignado un proyecto a Juan que contará con Ana para trabajar en él. De este modo Ana irá aprendiendo a la vez que ayuda a Juan en tan importante tarea.
Se trata de un proyecto importante y puede suponer muchas ventas, por tanto, una gran expansión para la empresa. Así que Ada supervisará todo el trabajo de Juan para que no haya ningún problema.
El director de una importante empresa se dirigió a BK programación para pedirles que desarrollen un sitio web de juegos online, al que se podrán conectar usuarios para jugar partidas. Se tiene que realizar un diseño de la base de datos que soporte la operativa de este sitio web.
Una cuestión vital en la aplicación es el almacenamiento de los datos. Los datos de los usuarios, el acceso de éstos, registro de las distintas partidas y juegos que se crean y el control de las compras de crédito por parte de los jugadores. Todo deberá guardarse en bases de datos, para su tratamiento y recuperación las veces que haga falta.
Como en BK programación trabajan sobre todo con Oracle, desde el primer momento Juan, con el visto bueno de Ada, tiene claro que van a tener que utilizar bases de datos relacionales y Oracle.

1.- Modelo de datos.

Caso práctico

Juan y Ana se han puesto en marcha con este nuevo proyecto. Ambos saben que lo primero que tienen que hacer es trabajar con la información que les han dado. Ya saben las ideas que el cliente tiene, ahora es necesario pasarlo a un formato con el que poder trabajar. Una de las primeras cosas que deben hacer es trazar un borrador donde plasmar lo que ahora mismo está en sus cabezas y en las anotaciones recogidas mientras hablaban con el cliente.

Según el DRAE, un modelo es, entre otras definiciones, el esquema teórico, generalmente en forma matemática, de un sistema o de una realidad compleja. Podemos decir que es la representación de cualquier aspecto o tema extraído del mundo real. ¿Qué sería entonces un modelo de datos? Aquél que nos permite describir los elementos que intervienen en una realidad o en un problema dado y la forma en que se relacionan dichos elementos entre sí.

En informática, un **modelo de datos** es un **lenguaje utilizado para la descripción de una base de datos**. Con este lenguaje vamos a poder describir las **estructuras** de los datos (tipos de datos y relaciones entre ellos), las **restricciones de integridad** (condiciones que deben cumplir los datos, según las necesidades de nuestro modelo basado en la realidad) y las operaciones de manipulación de los datos (insertado, borrado, modificación de datos).

Es importante distinguir entre modelo de datos y esquema.

Según Dittrich (1994): "La descripción específica de un determinado mini-mundo en términos de un modelo de datos se denomina esquema (o esquema de datos) del mini-mundo. La colección de datos que representan la información acerca del mini-mundo constituye la base de datos"

De Miguel, Piattini y Marcos (1999): "Representación de un determinado mundo real (universo del discurso) en términos de un modelo de datos".

Para clasificar los modelos debemos pensar en el nivel de abstracción, es decir, en lo alejado que esté del mundo real:

- ✓ Los **modelos de datos conceptuales** son aquellos que describen las estructuras de datos y restricciones de integridad. Se utilizan durante la etapa de análisis de un problema dado, y están orientados a representar los elementos que intervienen y sus relaciones. Ejemplo, Modelo Entidad-Relación.
- ✓ Los **modelos de datos lógicos** se centran en las operaciones y se implementan en algún sistema gestor de base de datos. Ejemplo, Modelo Relacional.
- ✓ Los **modelos de datos físicos**, son estructuras de datos a bajo nivel, implementadas dentro del propio sistema gestor de base de datos.

Hemos dicho que un modelo de datos es un lenguaje y por lo general, presenta dos sublenguajes:

- ✓ **Lenguaje de Definición de Datos o DDL (Data Definition Language)**, cuya función es describir, de una forma abstracta, las estructuras de datos y las restricciones de integridad.
- ✓ **Lenguaje de Manipulación de Datos o DML (Data Manipulation Language)**, que sirven para describir las operaciones de manipulación de los datos.

¿Cuáles son los modelos que se centran en las operaciones y se implementan en algún sistema gestor de base de datos?

- ○ Modelo de datos conceptuales.
- ◉ **Modelo de datos lógico.**
- ○ Modelo de datos físicos.

2.- Terminología del modelo relacional

Caso práctico

Ana se pregunta cuál será el modelo con el que se suele trabajar. Actualmente, para la mayoría de las aplicaciones de gestión que utilizan bases de datos, el modelo más empleado es el modelo relacional, por su gran versatilidad, potencia y su base matemática.

¿Sabes que el modelo relacional te va a permitir representar la información del mundo real de una manera intuitiva? Así es, pudiendo introducir conceptos cotidianos y fáciles de entender por cualquiera, aunque no sea experto en informática.

El modelo relacional fue propuesto por Edgar Frank Codd en los laboratorios de IBM en California. Como hemos visto, se trata de un modelo lógico que establece una estructura sobre los datos, independientemente del modo en que luego los almacenemos. Es como si guardamos nuestra colección de libros, dependiendo del número de habitaciones que tenga en casa, del tamaño y forma de nuestras estanterías, podremos disponer nuestros libros de un modo u otro para facilitarnos el acceso y consulta. Los libros serán los mismos pero puedo disponerlos de distinta forma.

El nombre de modelo relacional viene de la estrecha relación entre el elemento básico de este modelo y el concepto matemático de relación. Si tenemos dos conjuntos A y B, una relación entre estos dos conjuntos sería un subconjunto del producto cartesiano AxB.

El producto cartesiano nos dará la relación de todos los elementos de un conjunto con todos los elementos de los otros conjuntos de ese producto. Al estar trabajando con conjuntos, no puede haber elementos repetidos.

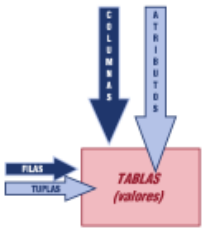
2.1.- Relación o tabla. Tuplas. Dominios.

Pero... ¿qué es eso de "relación"? Hemos dicho que el modelo relacional se basa en el concepto matemático de relación, ya que Codd, que era un experto matemático, utilizó una terminología perteneciente a las matemáticas, en concreto a la teoría de conjuntos y a la lógica de predicados.

Aquí tienes unos enlaces sobre teoría de conjuntos y lógica de predicados:

http://es.wikipedia.org/wiki/Teor%C3%ADa_de_conjuntos

http://es.wikipedia.org/wiki/L%C3%B3gica_de_primer_orden



A partir de ahora, nosotros veremos una relación como una **tabla** con **filas** y **columnas**. Podemos asociar atributos a columna y tuplas a filas.

- **Atributos:** es el nombre de cada dato que se almacena en la relación (tabla). Ejemplos serían: DNI, nombre, apellidos, etc.
  El nombre del atributo debe describir el significado de la información que representa. En la tabla **Empleados**, el atributo **Sueldo** almacenará el valor en euros del sueldo que recibe cada empleado. A veces es necesario añadir una pequeña descripción para aclarar un poco más el contenido. Por ejemplo, si el sueldo es neto o bruto.
- **Tuplas:** Se refiere a cada elemento de la relación. Si una tabla guarda datos de un cliente, como su DNI o Nombre, una tupla o registro sería ese DNI y nombre concreto de un cliente.
  Cada una de las filas de la tabla se corresponde con la idea de registro y 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).

Está claro que un atributo en una tupla no puede tomar cualquier valor. No sería lógico que en un atributo Población se guarde "250€". Estaríamos cometiendo un error, para evitar este tipo de

situaciones obligaremos a que cada atributo sólo pueda tomar los valores pertenecientes a un conjunto de valores previamente establecidos, es decir, un atributo tiene asociado un **dominio** de valores.

A menudo un dominio se define a través de la declaración de un tipo para el atributo (por ejemplo, diciendo que es un número entero entre 1 y 16), pero también se pueden definir dominios más complejos y precisos. Por ejemplo, para el atributo Sexo de mis usuarios, podemos definir un dominio en el que los valores posibles sean "M" o "F" (masculino o femenino).

Una característica fundamental de los dominios es que sean **atómicos**, es decir, que los valores contenidos en los atributos no se pueden separar en valores de dominios más simples.

Un dominio debe tener: **Nombre, Definición lógica, Tipo de datos y Formato.**

Por ejemplo, si consideramos el Sueldo de un empleado, tendremos:
- ✓ **Nombre**: Sueldo.
- ✓ **Definición lógica**: Sueldo neto del empleado
- ✓ **Tipo de datos**: número entero.
- ✓ **Formato**: 9.999€.

¿Cuáles de las siguientes afirmaciones son ciertas sobre las tuplas y los atributos?

- [x] **Las tuplas deben corresponderse con un elemento del mundo real.**
- [ ] Podríamos tener dos o más tuplas iguales.
- [x] **Un atributo se define en un dominio de valores.**
- [x] **El nombre de cada dato que se almacena en la relación se denomina Atributo.**

2.2.- Grado. Cardinalidad.

Ya hemos visto que una relación es una tabla con filas y columnas. Pero ¿hasta cuántas columnas puede contener? ¿Cuántos atributos podemos guardar en una tabla?

Llamaremos **grado** al tamaño de una tabla en base a su número de atributos (columnas). Mientras mayor sea el grado, mayor será su complejidad para trabajar con ella.

¿Y cuántas tuplas (filas o registros) puede tener?

Llamaremos **cardinalidad** al número de tuplas o filas de una relación o tabla.

Vamos a verlo con un ejemplo. Relación de grado 3, sobre los dominios A={Carlos, María}, B={Matemáticas, Lengua}, C={Aprobado, Suspenso}.

Las posibles relaciones que obtenemos al realizar el producto cartesiano AxBxC es el siguiente:

Producto Cartesiano AxBxC.		
A={Carlos, María}	**B={Matemáticas, Lengua}**	**C={Aprobado, Suspenso}**
CARLOS	MATEMÁTICAS	APROBADO
CARLOS	MATEMÁTICAS	SUSPENSO
CARLOS	LENGUA	APROBADO
CARLOS	LENGUA	SUSPENSO
CARLOS	INGLÉS	APROBADO

CARLOS	INGLÉS	SUSPENSO
MARÍA	MATEMÁTICAS	APROBADO
MARÍA	MATEMÁTICAS	SUSPENSO
MARÍA	LENGUA	APROBADO
MARÍA	LENGUA	SUSPENSO
MARÍA	INGLÉS	APROBADO
MARÍA	INGLÉS	SUSPENSO

Si cogemos un subconjunto de ésta con 5 filas, tendríamos una relación de cardinalidad 5:

Subconjunto del Producto Cartesiano AxBxC con cardinalidad 5.		
A={Carlos, María}	**B={Matemáticas, Lengua}**	**C={Aprobado, Suspenso}**
CARLOS	MATEMÁTICAS	APROBADO
CARLOS	LENGUA	APROBADO
CARLOS	INGLÉS	APROBADO
MARÍA	MATEMÁTICAS	APROBADO
MARÍA	INGLÉS	SUSPENSO

2.3.- Sinónimos.

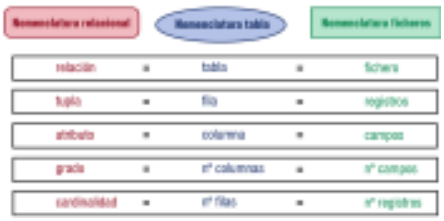
Caso práctico

Ana está un poco liada con tantos términos nuevos. ¿Si Juan habla de tuplas se está refiriendo a registros? Los registros eran las filas de las las tablas, ¿no? Será mejor que hagamos un resumen.

Los términos vistos hasta ahora tienen distintos sinónimos según la nomenclatura utilizada.

Trabajaremos con tres:

- ✓ **En el modelo relacional**: RELACIÓN - TUPLA - ATRIBUTO - GRADO - CARDINALIDAD.
- ✓ **En tablas**: TABLA - FILA - COLUMNAS - NÚMERO COLUMNAS - NÚMERO FILAS.
- ✓ **En términos de registros**: FICHEROS - REGISTROS - CAMPOS - NÚMERO CAMPOS - NÚMERO REGISTROS.



Relaciona cada término del modelo relacional con la terminología de Tablas.

Terminología del modelo relacional.	Relación.	Terminología en Tablas.
RELACIÓN	5	**1.** COLUMNAS
TUPLA	4	**2.** NÚMERO DE COLUMNAS
ATRIBUTO	1	**3.** NÚMERO DE FILAS
GRADO	2	**4.** FILA
CARDINALIDAD	3	**5.** TABLA

3.- Relaciones. Características de una relación (tabla).

Caso práctico

Juan, tras su análisis y varios días de trabajo, ha obtenido las relaciones con las que trabajará y los atributos que desea guardar en la base de datos. Junto con Ana va a repasar que se cumplan todas las propiedades y así asegurarse que el modelo es el adecuado. También necesitará saber qué información podrá consultar el usuario para así crear algunas tablas de modo temporal.

¿En un modelo relacional se puede utilizar cualquier relación? ¿Es válida cualquier tabla o se deben cumplir algunas propiedades?

Debes saber que:

- ✓ Cada tabla tiene un **nombre distinto**.
- ✓ Como hemos visto antes, cada atributo (columna) de la tabla **toma un solo valor** en cada tupla (fila).
- ✓ Cada atributo (columna) tiene un **nombre distinto** en cada tabla (pero puede ser el mismo en tablas distintas).
- ✓ **No** puede haber dos tuplas (filas) **completamente iguales**.

Carlos	Matemáticas	Aprobado
Carlos	Matemáticas	Suspenso
Carlos	Lengua	Aprobado
Carlos	Lengua	Aprobado

- ✓ El **orden** de las tuplas (filas) **no importa**.

a	20	=	a	20
b	30		c	70
c	70		b	30

- ✓ El **orden** de los atributos (columnas) **no importa**.

a	20	=	20	a
b	30		70	c
c	70		30	b

- ✓ Todos los datos de un atributo (columna) deben ser del **mismo dominio**.

Carlos	Matemáticas	Aprobado
Carlos	Lengua	Suspenso
Carlos	Inglés	NOTABLE
María	Matemáticas	Suspenso
María	Lengua	Suspenso

¿Cuál de las siguientes afirmaciones no es cierta en una relación?

- ○ Todos los atributos deben estar en el mismo dominio.
- ○ No puede haber dos tuplas completamente iguales.
- ○ Cada atributo de la tabla toma un único valor en cada tupla.
- ◉ **Podemos tener tablas con el mismo nombre en la misma base de datos.**

3.1.- Tipos de relaciones (tablas).

Caso práctico

Juan le está contando a Ana que hay que distinguir las relaciones en función del uso que se le vaya a dar. Tal y como han hablado con el cliente, sabe que unos jugadores accederán a un tipo de tablas como usuarios, y las personas que administran la base de datos lo harán a otras. Es obvio que tenemos que distinguir entre unas y otras.

Existen varios tipos de relaciones y las vamos a clasificar en:

- ✓ **Persistentes**: Sólo pueden ser borradas por los usuarios.
  - ➔ **Base**: Independientes, se crean indicando su estructura y sus ejemplares (conjunto de tuplas o filas).
  - ➔ **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 utilizan esos datos también cambiarán.
  - ➔ **Instantáneas**: son vistas (se crean de la misma forma) que sí almacenan los datos que muestran, además de la consulta que la creó. Solo modifican su resultado cuando el sistema se refresca cada cierto tiempo. Es como una fotografía de la relación, que sólo es válida durante un periodo de tiempo concreto.

- ✓ **Temporales**: Son tablas que son eliminadas automáticamente por el sistema.

Las relaciones que se crean indicando su estructura y sus ejemplares se denominan:

- Instantáneas
- Vistas
- **Base**

4.- Tipos de datos.

Caso práctico

Juan le ha pedido a Ana que repase cada una de las relaciones y en función de los datos que contenga cada atributo, elegir el tipo de datos más adecuado. Más adelante habrá que restringir esos valores para que al introducir datos no se produzcan errores. Los tipos de datos ocupan espacio en el disco duro del servidor donde se guarde y eso significa un gasto para la empresa, así que hay que optimizar.

Además, Ana todavía recuerda aquella vez que tuvo que entregar una práctica en la facultad sobre base de datos. Guardó el teléfono con un formato de número y cuando fue a imprimir un informe... ¡no quiere ni acordarse! Le salieron unos números de teléfonos que nada tenían que ver con los datos introducidos.

¿Qué es un DNI? ¿Con qué datos lo representamos? DNI es una información que es susceptible de ser guardada. Normalmente el DNI está formado por dígitos y una letra al final. Si tuviéramos que clasificarlo diríamos que es un conjunto de caracteres alfanuméricos. ¿Y si pensamos en Sueldo? Aquí lo tenemos un poco más claro, evidentemente es un número entero o con decimales.

Hasta ahora hemos visto que vamos a guardar información relacionada en forma de filas y columnas. Las columnas son los atributos o información que nos interesa incluir del mundo real que estamos modelando.

Hemos visto que esos atributos se mueven dentro de un dominio, que formalmente es un conjunto de valores. Pues bien, en términos de sistemas de base de datos, se habla más de tipos de datos que de dominios. Al crear la relación (tabla) decidimos qué conjunto de datos deberá ser almacenado en las filas de los atributos que hemos considerado. Tenemos que asignar un tipo de dato a cada atributo.

Con la asignación de tipos de datos, también habremos seleccionado un dominio para un atributo.
Cada campo:

- ✓ debe poseer un **Nombre** (relacionado con los datos que va a contener) y
- ✓ debe tener asociado un **Tipo de dato.**

Existen distintas formas de nombrar los tipos de datos dependiendo del lenguaje que utilicemos (C, Java, PHP, MySQL, SQL, Pascal, etc.).

Veamos cuales son los tipos de datos más comunes con los que nos encontraremos generalmente:

- ✓ **Texto**: almacena cadenas de caracteres (números con los que **no** vamos a realizar operaciones matemáticas, letras o símbolos).
- ✓ **Numérico**: almacena números con los que vamos a realizar operaciones matemáticas.
- ✓ **Fecha/hora**: almacena fechas y horas.
- ✓ **Sí/No**: almacena datos que solo tienen dos posibilidades (verdadero/falso).
- ✓ **Autonumérico**: valor numérico secuencial que el SGBD incrementa de modo automático al añadir un registro (fila).
- ✓ **Memo**: almacena texto largo (mayor que un tipo texto).
- ✓ **Moneda**: se puede considerar un subtipo de Numérico ya que almacena números, pero con una característica especial, y es que los valores representan cantidades de dinero.
- ✓ **Objeto OLE**: almacena gráficos, imágenes o textos creados por otras aplicaciones.

Si quieres saber un poco más sobre los tipos de datos puedes ver este enlace de Wikipedia:
http://es.wikipedia.org/wiki/Tipo_de_dato

5.- Claves.

Caso práctico

Juan está revisando la relación Usuarios. En esta tabla va a guardar los siguientes atributos: Login del jugador que será nuestro usuario, Password o Contraseña, Nombre y Apellidos, Dirección, Código Postal, Localidad, Provincia, País, Fecha de nacimiento para comprobar que no es menor de edad, Fecha de ingreso en la web, Correo electrónico, Sexo y por último los Créditos (dinero "ficticio") que tenga.

¿Cómo diferenciamos unos usuarios de otros? ¿Cómo sabemos que no estamos recogiendo la misma información? ¿Cómo vamos a distinguir unas tuplas de otras? Lo haremos mediante los valores de sus atributos. Para ello, buscaremos un atributo o un conjunto de atributos que identifiquen de modo único las tuplas (filas) de una relación (tabla). A ese atributo o conjunto de atributos lo llamaremos **superclaves**.

Hemos visto que una característica de las tablas era que no puede haber dos tuplas (filas) completamente iguales, con lo que podemos decir que toda la fila como conjunto sería una superclave.

Por ejemplo, en la tabla Usuarios tenemos las siguientes superclaves:

- ✓ {Nombre, Apellidos, login, e_mail, F_nacimiento}
- ✓ {Nombre, Apellidos, login, e_mail}
- ✓ {login, e_mail}
- ✓ {login}

Tendríamos que elegir alguna de las superclaves para diferenciar las tuplas. En el modelo relacional trabajamos con tres tipos de claves:

- ✓ Claves **candidatas**.
- ✓ Claves **primarias**.
- ✓ Claves **alternativas.**
- ✓ Claves **ajenas**.

A continuación veremos cada una de ellas.

En este enlace tienes más información sobre las superclaves:
http://www.victorgarcia.org/pfc/modeloER/claves.php

5.1.- Clave candidata. Clave primaria. Clave alternativa.

Si puedo elegir entre tantas claves, ¿con cuál me quedo? Tendremos que elegir entre las claves "candidatas" la que mejor se adapte a mis necesidades. ¿Y cuáles son éstas? Las claves **candidatas** serán aquel conjunto de atributos que identifiquen de manera única cada tupla (fila) de la relación (tabla). Es decir, las columnas cuyos valores no se repiten en ninguna otra fila de la tabla. Por tanto, cada tabla debe tener **al menos una clave candidata** aunque puede haber más de una.

Siguiendo con nuestro ejemplo, podríamos considerar los atributos Login o E_mail como claves candidatas, ya que sabemos que el Login debe ser único para cada usuario, a E_mail le sucede lo mismo. Pero también cabe la posibilidad de tomar: Nombre, Apellidos y F_nacimiento, las tres juntas como clave candidata.

Las claves candidatas pueden estar formadas por más de un atributo, siempre y cuando éstos identifiquen de forma única a la fila. Cuando una clave candidata está formada por más de un atributo, se dice que es una **clave compuesta**.

Una clave **candidata** debe cumplir los siguientes requisitos:

- ✓ **Unicidad**: no puede haber dos tuplas (filas) con los mismos valores para esos atributos.
- ✓ **Irreducibilidad**: si se elimina alguno de los atributos deja de ser única.

Si elegimos como clave candidata **Nombre, Apellidos y F_nacimiento**, cumple con la unicidad puesto que es muy difícil encontrarnos con dos personas que tengan el mismo nombre, apellidos y fecha de nacimiento iguales. Es irreducible puesto que sería posible encontrar dos personas con el mismo nombre y apellidos o con el mismo nombre y fecha de nacimiento, por lo que son necesarios los tres atributos (campos) para formar la clave.

Para identificar las claves candidatas de una relación no nos fijaremos en un momento concreto en el que vemos una base de datos. Puede ocurrir que en ese momento no haya duplicados para un atributo o conjunto de atributos, pero esto no garantiza que se puedan producir. El único modo de identificar las claves candidatas es conociendo el significado real de los atributos (campos), ya que así podremos saber si es posible que aparezcan duplicados. Es posible desechar claves como candidatas fijándonos en los posibles valores que podemos llegar a tener. Por ejemplo, podríamos pensar que Nombre y Apellidos podrían ser una clave candidata, pero ya sabemos que cabe la posibilidad de que dos personas puedan tener el mismo Nombre y Apellidos, así que lo descartamos.

Hasta ahora, seguimos teniendo varias claves con la que identificamos de modo único nuestra relación. De ahí el nombre de candidatas. Hemos de quedarnos con una.

La **clave primaria** de un relación es aquella clave candidata que se escoge para identificar sus tuplas de modo único. Ya que una relación no tiene tuplas duplicadas, siempre hay una clave candidata y, por lo tanto, la relación siempre tiene clave primaria. En el peor caso, la clave primaria estará formada por todos los atributos de la relación, pero normalmente habrá un pequeño subconjunto de los atributos que haga esta función. En otros casos, podemos crear un campo único que identifique las tuplas, por ejemplo un código de usuario, que podrían estar constituidos por valores autonuméricos.

Las claves candidatas que no son escogidas como clave primaria son denominadas claves alternativas.

Si en nuestra tabla Usuarios escogemos Login como clave primaria, el E_mail o {Nombre, Apellidos, F_Nacimiento} serán nuestras claves alternativas.

Rellena los huecos con los conceptos adecuados.

Dentro del conjunto de superclaves, se llaman claves candidatas a aquellas que identifican unívocamente a cada una de las tuplas. De entre éstas, escogeremos la clave primaria. Aquellas que no escogemos se denominarán claves alternativas.

5.2.- Clave externa, ajena o secundaria.

Hasta ahora no nos hemos planteado cómo se relacionan unas tablas con otras dentro de una base de datos. Si tenemos las tablas Usuarios y Partidas, necesariamente habrá una "relación" entre ellas. Deben compartir algún dato en común que las relacione. Una partida es jugada por un jugador (Usuarios), por lo que en la tabla Partida deberíamos guardar algún dato del usuario-jugador, pero ¿cuál?

Una clave **ajena, también llamada externa o secundaria,** es un atributo o conjunto de atributos de una relación cuyos valores coinciden con los valores de la clave primaria de alguna otra relación (o de la misma). Las claves ajenas **representan relaciones entre datos**. Dicho de otra manera, son los datos de atributos de una tabla cuyos valores están relacionados con atributos de otra tabla.

En la tabla Partidas, se recogen datos como **Cod_partida**, **Fecha** y **Hora de creación**, **Nombre de la partida**, etc. ¿Qué campo utilizaremos para relacionarla con la tabla Usuarios? Si nos basamos en la definición, deberíamos utilizar la clave primaria de la tabla Usuarios. Por tanto, el atributo Login que es la clave principal en su tabla aparecerá en la tabla Partidas como clave ajena, externa o secundaria. El Login en Partidas hace referencia a cada jugador que juega esa partida. En lugar de guardar todos los datos de ese jugador en la misma tabla, lo hacemos en otra y lo "referenciamos" por su clave primaria tomándola como ajena.

Es lógico que las claves ajenas no tengan las mismas propiedades y restricciones que tienen como clave primaria en su tabla, por tanto, sí que pueden repetirse en la tabla. En nuestro ejemplo, un mismo jugador puede jugar varias partidas.

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

No podemos tener una partida de un jugador que previamente no se ha registrado. Pero sí podemos tener los datos de una partida y desconocer el jugador de ésta.

¿Cuáles de las siguientes afirmaciones sobre las claves ajenas son correctas?

- [x] **Puede "referenciar" a la clave primaria de la misma tabla donde se encuentra.**
- [x] **Puede "referenciar" a la clave primaria de otra tabla.**
- [x] **Representa relaciones entre datos.**
- [x] **Puede contener valores nulos.**
- [ ] No puede repetirse en la tabla.

Interesante artículo sobre las claves ajenas y su importancia:
http://blogs.ua.es/fbdblog/2011/03/29/de-las-claves-ajenas-foraneas-externas/
Si necesitas refrescar o simplemente aprender el concepto de clave primaria, en la wikipedia puedes consultarlo:
http://es.wikipedia.org/wiki/Clave_primaria

6.- Índices. Características.

Caso práctico

Juan considera que es beneficioso crear un índice para la tabla Usuarios. Podría agilizar las búsquedas de usuarios registrados. Le ha contado a Ana que es conveniente tenerlo, aunque también le ha explicado que tener muchos índices no es bueno. Tendrán que hacer una buena elección del número de índices que van a manejar.

Imagina que estás creando un diccionario de términos informáticos. Podrías elegir la opción de escribirlo en una única hoja muy larga (estilo pergamino) o bien distribuirlo por hojas. Está claro que lo mejor sería distribuirlo por páginas. Y si buscamos el término "informática" en nuestro diccionario, podríamos comenzar a buscar en la primera página y continuar una por una hasta llegar a la palabra correspondiente. O bien crear un índice al principio, de manera que podamos consultar a partir de qué página podemos localizar las palabras que comienzan por "i". Esta última opción parece la más lógica.

Pues bien, en las bases de datos, cada tabla se divide internamente en páginas de datos, y se define el índice a través de un campo (o campos) y es a partir de este campo desde donde de busca.
Un **índice** es una estructura de datos que permite acceder a diferentes filas de una misma tabla a través de un campo o campos . Esto permite un acceso mucho más rápido a los datos.
Los índices son útiles cuando se realizan consultas frecuentes a un rango de filas o una fila de una tabla. Por ejemplo, si consultamos los usuarios cuya fecha de ingreso es anterior a una fecha concreta.

Los cambios en los datos de las tablas (agregar, actualizar o borrar filas) son incorporados automáticamente a los índices con transparencia total.

Debes saber que los índices son independientes, lógica y físicamente de los datos, es por eso que pueden ser creados y eliminados en cualquier momento, sin afectar a las tablas ni a otros índices.
¿Cuándo indexamos? No hay un límite de columnas a indexar, si quisiéramos podríamos crear un índice para cada columna, pero no sería operativo. Normalmente tiene sentido crear índices para ciertas columnas ya que agilizan las operaciones de búsqueda de base de datos grandes. Por ejemplo, si la información de nuestra tabla Usuarios se desea consultar por apellidos, tiene sentido indexar por esa columna.

Al crear índices, las operaciones de modificar o agregar datos se ralentizan, ya que al realizarlas es necesario actualizar tanto la tabla como el índice.

Si se elimina un índice, el acceso a datos puede ser más lento a partir de ese momento.

Si quieres conocer más sobre los índices y MySQL puedes leer este artículo:
http://www.programacion.com/articulo/indices_y_optimizacion_de_consultas_305

7.- El valor NULL. Operaciones con este valor.

Caso práctico

Ana tiene un poco más claro el concepto de relación y las características de los atributos. Sabe que éstos se definen en un dominio. Pero ¿qué ocurre si no conozco algún valor de un dato? ¿Si en la tabla de usuarios estoy pidiendo que se guarde el sexo y el jugador no quiere decirlo? ¿Qué puede ocurrir? Si se permite que ese dato no sea obligatorio lo que me quedaría sería un dato vacío de información.

Vamos a ver que eso es posible y que ese valor tiene una denominación propia.

¿Qué sucede si al guardar los datos de los Usuarios hay algún dato que no tengo o no necesito guardarlo porque no corresponde?
Independientemente del dominio al que pertenezca un campo, éste puede tomar un valor especial denominado **NULO** (**NULL** en inglés) que designará la ausencia de dato.

Cuando por cualquier motivo se desconoce el valor de un campo, por ejemplo, desconocemos el teléfono del usuario, o bien ese campo carece de sentido (siguiendo con el mismo ejemplo, puede que el usuario no tenga teléfono), podemos asignar a ese campo el valor especial NULO.

Cuando trabajamos con claves secundarias el valor nulo indica que la tupla o fila no está relacionada con ninguna otra tupla o fila. Este valor NULO es común a cualquier dominio.

**Pero ten en cuenta una cosa, no es lo mismo valor NULO que ESPACIO EN BLANCO.
Tampoco será lo mismo valor NULO que el valor CERO.**

Un ordenador tomará un espacio en blanco como un carácter como otro cualquiera. Por tanto, si introducimos el carácter "espacio en blanco" estaríamos introduciendo un valor que pertenecería al dominio **texto** y sería distinto al concepto "ausencia de valor" que sería **no incluir nada** (nulo).
Este valor se va a utilizar con frecuencia en las bases de datos y es imprescindible saber cómo actúa cuando se emplean operaciones lógicas sobre ese valor. En la lógica booleana tenemos los valores VERDADERO y FALSO, pero un valor NULO no es ni verdadero ni falso.

Cuando necesitemos comparar dos campos, si ambos son nulos no podremos obtener ni verdadero ni falso. Necesitaremos definir la lógica con este valor. Veamos los operadores lógicos más comunes y sus resultados utilizando el valor nulo:

- ✓ VERDADERO Y (AND) NULO daría como resultado NULO.
- ✓ FALSO Y (AND) NULO daría como resultado FALSO.
- ✓ VERDADERO O (OR) NULO daría como resultado VERDADERO.
- ✓ FALSO O NULO daría como resultado NULO.
- ✓ NO (NOT) NULO daría como resultado NULO.

En todas las bases de datos relacionales se utiliza un operador llamado ES NULO (IS NULL) que devuelve VERDADERO si el valor con el que se compara es NULO.

El uso del valor nulo es un tema que da mucho que hablar, aquí puedes leer sobre ello:
http://es.wikipedia.org/wiki/Null_%28SQL%29

¿Cuáles de las siguientes afirmaciones sobre el valor nulo son ciertas?

- [x] **Designa ausencia de dato**
- [ ] Es lo mismo que espacio en blanco
- [ ] Es lo mismo que cero

8.- Vistas.

Caso práctico

Ana lleva un buen rato pensando cómo hacer si necesitara consultar datos de dos tablas distintas, por ejemplo, sería interesante obtener los nombres de los usuarios que estén jugando una determinada partida. O quizás consultar otros datos por el estilo. ¿Cómo lo hace si ya están definidas las tablas del modelo? ¿Cómo crear esas tablas? Juan le va a explicar que esa información la puede obtener a través de las vistas.

Cuando vimos los distintos tipos de relaciones, aprendimos que, entre otros, estaban las vistas. Ahora ya tenemos más conocimientos para comprender mejor este concepto.

Una **vista** es una tabla "virtual" cuyas filas y columnas se obtienen a partir de una o de varias tablas que constituyen nuestro modelo. Lo que se almacena no es la tabla en sí, sino su definición, por eso decimos que es "virtual". Una vista actúa como filtro de las tablas a las que hace referencia en ella.

La consulta que define la vista puede provenir de una o de varias tablas, o bien de otras vistas de la base de datos actual u otras bases de datos.

No existe ninguna restricción a la hora de consultar vistas y muy pocas restricciones a la hora de modificar los datos de éstas.

Podemos dar dos razones por las que queramos crear vistas:

- ✓ **Seguridad,** nos puede interesar que los usuarios tengan acceso a una parte de la información que hay en una tabla, pero no a toda la tabla.
- ✓ **Comodidad**, como veremos al pasar nuestras tablas/relaciones a un lenguaje de base de datos, puede que tengamos que escribir sentencias bastante complejas, las vistas no son tan complejas.

Las vistas no tienen una copia física de los datos, son consultas a los datos que hay en las tablas, por lo que si actualizamos los datos de una vista, estamos actualizando realmente la tabla, y si actualizamos la tabla estos cambios serán visibles desde la vista.

Aunque **no siempre** podremos actualizar los datos de una vista, dependerá de la complejidad de la misma y del gestor de base de datos. No todos los gestores de bases de datos permiten actualizar vistas, Oracle, por ejemplo, no lo permite, mientras que SQL Server sí.

Una vista puede proceder de:

- ☑ Una tabla
- ☑ Varias tablas
- ☑ Otras vistas de la misma base de datos
- ☑ Otras vistas de otras bases de datos

9.- Usuarios. Roles. Privilegios

Caso práctico

Juan debe consultar al cliente qué usuarios van a acceder a la base de datos y qué privilegios se les va a otorgar. Esta parte es primordial si queremos salvaguardar el contenido de la base de datos. ¿Qué ocurriría si cualquiera pudiera ver la información personal de todos los usuarios registrados? Estaríamos cometiendo un fallo de seguridad además de incumplir la ley de protección de datos.

A la hora de conectarnos a la base de datos es necesario que utilicemos un modo de acceso, de manera que queden descritos los permisos de que dispondremos durante nuestra conexión. En función del nombre de usuario tendremos unos permisos u otros.

Un **usuario** es un conjunto de permisos que se aplican a una conexión de base de datos. Tiene además otras funciones como son:

- ✓ Ser el propietario de ciertos objetos (tablas, vistas, etc.).
- ✓ Realiza las copias de seguridad.
- ✓ Define una cuota de almacenamiento.
- ✓ Define el tablespace por defecto para los objetos de un usuario en Oracle.

Pero no todos los usuarios deberían poder hacer lo mismo cuando acceden a la base de datos. Por ejemplo, un administrador debería tener más privilegios que un usuario que quiere realizar una simple consulta.

¿Qué es un **privilegio**? No es más que un permiso dado a un usuario para que realice ciertas operaciones, que pueden ser de dos tipos:

- ✓ **De sistema**: necesitará el permiso de sistema correspondiente.
- ✓ **Sobre objeto**: necesitará el permiso sobre el objeto en cuestión.

¿Y no sería interesante poder agrupar esos permisos para darlos juntos? Para eso tenemos el rol.

Un **rol** de base de datos no es más que una agrupación de permisos de sistema y de objeto.

Podemos tener a un grupo determinado de usuarios que tengan permiso para consultar los datos de una tabla concreta y no tener permiso para actualizarlos. Luego un rol permite asignar un grupo de permisos a un usuario. De este modo, si asignamos un rol con 5 permisos a 200 usuarios y luego queremos añadir un permiso nuevo al rol, no tendremos que ir añadiendo este nuevo permiso a los 200 usuarios, ya que el rol se encarga de propagarlo automáticamente.

Rellena los huecos con los conceptos adecuados.

Al conjunto de permisos que se aplican a una conexión de base de datos, se le llama usuario. Los permisos dados a usuarios para que realicen ciertas operaciones se les llama privilegios. Si tengo una agrupación de permisos juntos, tenemos un rol.

10.- SQL.

Caso práctico

Hasta ahora Ana y Juan no han tenido que utilizar mucho el ordenador, ya es hora de ponerse manos a la obra. El diseño está casi finalizado y ahora es necesario pasarlo a un lenguaje adecuado. Juan había acordado con Ada que usarían Oracle como SGBD. Para trabajar con esta aplicación es necesario tener conocimientos del lenguaje que utiliza, en concreto SQL para Oracle, que tiene ciertas variaciones con el estándar. Ana está deseando comenzar a introducir los datos necesarios.

SQL (Structured Query Language) es el lenguaje fundamental de los SGBD relacionales. Es uno de los lenguajes más utilizados en informática en todos los tiempos. Es un lenguaje declarativo y por tanto, lo más importante es definir **qué** se desea hacer, y **no cómo** hacerlo. De esto último ya se encarga el SGBD.
Hablamos por tanto de un lenguaje normalizado que nos permite trabajar con cualquier tipo de lenguaje (ASP o PHP) en combinación con cualquier tipo de base de datos (Access, SQL Server, MySQL, Oracle, etc.).

El hecho de que sea estándar no quiere decir que sea idéntico para cada base de datos. Así es, determinadas bases de datos implementan funciones específicas que no tienen necesariamente que funcionar en otras.

Aunque SQL está estandarizado, siempre es recomendable revisar la documentación del SGBD con el que estemos trabajando para conocer su sintaxis concreta, ya que algún comando, tipo de dato, etc., puede no seguir el estándar.

SQL posee dos características muy apreciadas, **potencia** y **versatilidad**, que contrastan con su facilidad para el aprendizaje, ya que utiliza un lenguaje bastante natural. Es por esto que las instrucciones son muy parecidas a órdenes humanas. Por esta característica se le considera un Lenguaje de Cuarta Generación.
Aunque frecuentemente oigas que SQL es un "lenguaje de consulta", ten en cuenta que no es exactamente cierto ya que contiene muchas otras capacidades además de la de consultar la base de datos:

- ✓ la definición de la propia estructura de los datos,
- ✓ su manipulación,
- ✓ y la especificación de conexiones seguras.

Por tanto, el lenguaje estructurado de consultas SQL es un lenguaje que permite operar con los datos almacenados en las bases de datos relacionales.

Para saber más

Ya hemos llegado a los lenguajes de quinta generación, en el siguiente enlace puedes ver sus características más generales:

http://es.wikipedia.org/wiki/Generaciones_de_lenguajes_de_programaci%C3%B3n

En este enlace encontrarás de una manera breve, pero interesante, la historia del SQL.

http://www.htmlpoint.com/sql/sql_04.htm

10.1.- Elementos del lenguaje. Normas de escritura.

Imagínate que cada programador utilizara sus propias reglas para escribir. Esto sería un caos. Es muy importante establecer los elementos con los que vamos a trabajar y unas normas que seguir.

El lenguaje SQL está compuesto por comandos, cláusulas, operadores, funciones y literales. Todos estos elementos se combinan en las instrucciones y se utilizan para crear, actualizar y manipular bases de datos. Estos conceptos son bastante amplios por eso será mejor que vayamos por partes.

- ✓ **COMANDOS**: Van a ser las instrucciones que se pueden crear en SQL. Se pueden distinguir en tres grupos que veremos con más detenimiento a lo largo de las siguientes unidades:
  - ➔ De definición de datos (DDL, Data Definition Language), que permiten crear y definir nuevas bases de datos, tablas, campos, etc.
  - ➔ De manipulación de datos (DML, Data Manipulation Language), que permiten generar consultas para ordenar, filtrar y extraer datos de la base de datos.
  - ➔ De control y seguridad de datos (DCL, Data Control Language), que administran los derechos y restricciones de los usuarios.
- ✓ **CLÁUSULAS:** Llamadas también condiciones o criterios, son palabras especiales que permiten modificar el funcionamiento de un comando.
- ✓ **OPERADORES**: Permiten crear expresiones complejas. Pueden ser aritméticos (+, -, *, /, ...) o lógicos (< , >, , < >, And, Or, etc.).
- ✓ **FUNCIONES:** Para conseguir valores complejos. Por ejemplo, la función promedio para obtener la media de un salario.
- ✓ **LITERALES**: Les podemos llamar también constantes y serán valores concretos, como por ejemplo un número, una fecha, un conjunto de caracteres, etc.

Y tendremos que seguir unas normas sencillas pero primordiales:
- ✓ Todas las instrucciones **terminan con un signo de punto y coma**.
- ✓ No se distingue entre mayúsculas y minúsculas.
- ✓ Cualquier comando puede ser partido con saltos de línea o espacios para facilitar su lectura y comprensión.
- ✓ Los comentarios comienzan por /* y terminan con */ (excepto en algunos SGBD).

Juan le ha dicho a Ana que es hora de ponerse a trabajar con la aplicación. Para aprender mejor le ha pedido permiso a Juan para instalar Oracle en su ordenador y así ir probando todo sobre la marcha para no cometer errores. El SQL estándar y el SQL de Oracle son bastante parecidos, pero con algunas diferencias.

A continuación te mostramos aquellos comandos, cláusulas, operadores y funciones más generales con las que vamos a trabajar a lo largo del curso.

Elementos del lenguaje SQL.

El lenguaje SQL está compuesto por comandos, cláusulas, operadores, funciones y literales . Todos estos elementos se combinan en las instrucciones y se utilizan para crear, actualizar y manipular bases de datos.

- ✓ **COMANDOS:**

Comandos DDL. Lenguaje de Definición de Datos.	
Comando:	**Descripción:**
CREATE	Se utiliza para crear nuevas tablas, campos e índices.
DROP	Se utiliza para eliminar tablas e índices.
ALTER	Se utiliza para modificar tablas.

Comandos DML. Lenguaje de Manipulación de Datos.	
Comando:	**Descripción:**
SELECT	Se utiliza para consultar filas que satisfagan un criterio determinado.
INSERT	Se utiliza para cargar datos en una única operación.
UPDATE	Se utiliza para modificar valores de campos y filas específicos.
DELETE	Se utiliza para eliminar filas de una tabla.

Comandos DCL. Lenguaje de Control de Datos.	
Comando:	**Descripción:**
GRANT	Permite dar permisos a uno o varios usuarios o roles para realizar tareas determinadas.
REVOKE	Permite eliminar permisos que previamente se han concedido con GRANT.

✓ **CLÁUSULAS:**

Llamadas también condiciones o criterios, son palabras especiales que permiten modificar el funcionamiento de un comando.

Cláusulas	
Cláusulas:	**Descripción:**
FROM	Se utiliza para especificar la tabla de la que se van a seleccionar las filas.
WHERE	Se utiliza para especificar las condiciones que deben reunir las filas que se van a seleccionar.
GROUP BY	Se utiliza para separar las filas seleccionadas en grupos específicos.
HAVING	Se utiliza para expresar la condición que debe satisfacer cada grupo.
ORDER BY	Se utiliza para ordenar las filas seleccionadas de acuerdo a un orden específico.

✓ **OPERADORES:**

Permiten crear expresiones complejas. Pueden ser aritméticos (+, -, *, /, ...) o lógicos (< , >, , < >, And, Or, …).

Operadores lógicos.	
Operadores:	**Descripción:**
AND	Evalúa dos condiciones y devuelve un valor de verdad sólo si ambas son ciertas.
OR	Evalúa dos condiciones y devuelve un valor de verdad si alguna de las dos es cierta.
NOT	Devuelve el valor contrario de la expresión.

Operadores de comparación.	
Operadores:	**Descripción:**
<	Menor que.
>	Mayor que.
< >	Distinto de.
< =	Menor o igual.
> =	Mayor o igual.
=	Igual.
BETWEEN	Se utiliza para especificar un intervalo de valores.
LIKE	Se utiliza para comparar.
IN	Se utiliza para especificar filas de una base de datos.

✓ **FUNCIONES:**

Para conseguir valores complejos. Por ejemplo, la función promedio para obtener la media de un salario. Existen muchas funciones, aquí tienes la descripción de algunas.

Funciones de agregado.	
Función:	**Descripción:**
AVG	Calcula el promedio de los valores de un campo determinado.
COUNT	Devuelve el número de filas de la selección.
SUM	Devuelve la suma de todos los valores de un campo determinado.
MAX	Devuelve el valor más alto de un campo determinado.
MIN	Devuelve el valor mínimo de un campo determinado.

✓ **LITERALES:**

Les podemos llamar también constantes y serán valores concretos, como por ejemplo un número, una fecha, un conjunto de caracteres, etc.

Literales	
Literales:	**Descripción:**
23/03/97	Literal fecha.
María	Literal caracteres.
5	Literal número.

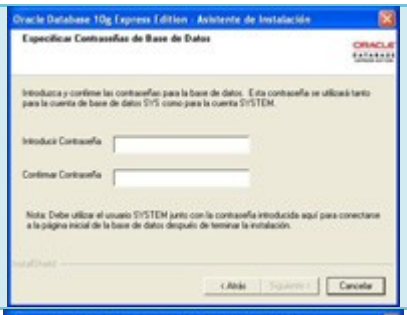
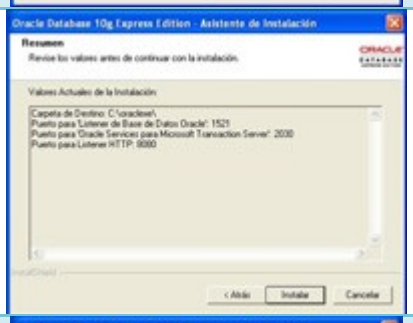
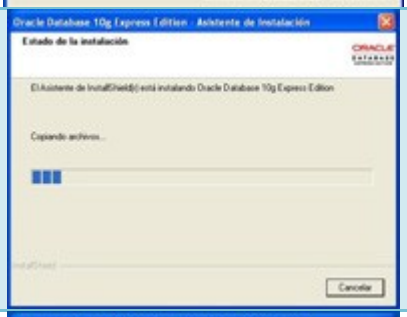
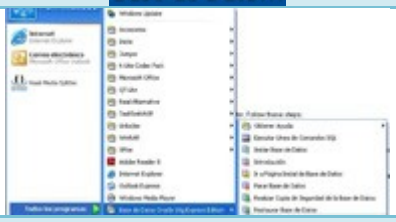
Para trabajar con Oracle tendrás que instalar el programa adecuado, aquí tienes el enlace donde puedes bajarte la aplicación gratuita:
http://www.oracle.com/technetwork/database/express-edition/downloads/index.html

Ahora te mostramos los pasos que debes seguir para la instalación de la aplicación en tu ordenador:

Instalación de Oracle XE.

A partir de ahora vamos a trabajar con este Sistema Gestor de Base de Datos. Para ello debes ir a la página oficial de Oracle:

http://www.oracle.com/technetwork/database/express-edition/downloads/index.html
En ella podemos elegir entre dos tipos de descargas según sea nuestro Sistema Operativo:
La página de Oracle solicitará nuestro registro para realizar la descarga, pues tenemos que ser usuarios registrados para poder bajarlo. 
Una vez bajado el archivo, tendremos que ejecutarlo haciendo doble clic sobre él. 
Al hacerlo, aparecerá una ventana donde podrás ver que está preparándose para la instalación y tras unos segundos aparecerá la página de bienvenida donde pulsaremos en siguiente.
A continuación se nos muestra otra ventana donde tenemos que aceptar los términos del acuerdo para poder continuar con la instalación: 
Tras aceptar y pulsar en siguiente aparecerá otra ventana donde podemos elegir donde instalar el programa, en principio es mejor dejar lo que aparece por defecto. 

Pulsamos en siguiente. En la ventana que aparece a continuación nos solicitará que introduzcamos una contraseña para los usuarios SYS y SYSTEM (será la misma para los dos).	
Pulsamos en siguiente y aparecerá otra ventana donde nos informa entre otras cosa, que la instalación se realizará para conectar mediante HTTP en el puerto 8080. Ahora el botón que tenemos que pulsar es Instalar.	
El asistente se pondrá a instalar.	
Transcurrido un tiempo aparecerá otra ventana que indicará que el proceso ha terminado y que pulsemos en Terminar.	
Tras estos pasos, Oracle se abrirá automáticamente para que puedas comenzar a trabajar. También se habrá creado un icono de acceso:	
También podrás ver que tienes un acceso en el botón de inicio.	
Si accedemos a Ir a Página Inicial de Base de Datos, debe aparecer la siguiente ventana donde podremos acceder a la base de datos que trae por defecto. Aquí usaremos las claves que pusimos antes.	
Ya estamos listos para trabajar.	

Otro Sistema Gestor de Base de Datos muy utilizado en algunos entornos como el de desarrollo web es MySQL. Sería interesante que lo conocieras y supieras instalarlo:
http://dev.mysql.com/doc/refman/5.0/es/index.html
Otra página recomendable donde puedes aprender MySQL desde cero es la siguiente:
http://mysql.conclase.net/curso/index.php

11.- Lenguaje de descripción de datos (DDL).

Caso práctico

__Ana__ y __Juan__ han realizado concienzudamente el diseño de las tablas necesarias para la base de datos de la aplicación en la que están trabajando.
También se han decantado por el sistema gestor de bases de datos a utilizar. Emplearán un sistema gestor de bases de datos relacional. Una vez instalado el sistema gestor, tendrán que programar los accesos a la base de datos para guardar los datos, recuperarlos, realizar las consultas para los informes y documentos que sean necesarios, etc.
__Ana__ está creando las primeras tablas de la base de datos. Una de las principales es USUARIO, aunque también tendrá que crear la de PARTIDAS y JUEGOS.

La primera fase del trabajo con cualquier base de datos comienza con sentencias DDL, puesto que antes de poder almacenar y recuperar información debemos definir las estructuras donde almacenar la información. Las estructuras básicas con las que trabaja SQL son las tablas.

Conocer el Lenguaje de Definición de Datos (DDL) es imprescindible para crear, modificar y eliminar objetos de la base de datos (es decir, los metadatos). En el mercado hay suficientes aplicaciones y asistentes que nos facilitan esta labor, a través de una interfaz visual que nos oculta el lenguaje SQL y en los cuales nos limitamos a poner nombres a los campos, elegir el tipo de datos y activar una serie de propiedades.

Es cierto que estas herramientas nos facilitan el trabajo, pero resulta imprescindible comprender y conocer en profundidad el lenguaje, ya que nos veremos en muchas situaciones donde necesitaremos crear un objeto, modificarlo o eliminarlo sin depender de esas herramientas visuales.

En Oracle, cada usuario de una base de datos tiene un esquema, que tendrá el mismo nombre que el usuario con el que se ha accedido y sirve para almacenar los objetos que posea ese usuario.

¿De qué objetos estamos hablando? Éstos podrán ser tablas, vistas, índices u otros objetos relacionados con la definición de la base de datos. ¿Y quién puede crear y manipularlos? En principio el usuario propietario (el que los creó) y los administradores de la base de datos. Más adelante veremos que podemos modificar los privilegios de los objetos para permitir el acceso a otros usuarios.

Las instrucciones DDL generan acciones que no se pueden deshacer, por eso es conveniente usarlas con precaución y tener copias de seguridad cuando manipulamos la base de datos.

Si quieres saber un poco más sobre el Lenguaje de Definición de Datos, puedes visitar la Wikipedia, aquí tienes el enlace:
http://es.wikipedia.org/wiki/Lenguaje_de_definici%C3%B3n_de_datos

11.1.- Creación de bases de datos. Objetos de la base de datos.

Básicamente, la creación de la base de datos consiste en crear las tablas que la componen. Aunque antes de ésto tendríamos que definir un espacio de nombres separado para cada conjunto de tablas. Es lo que antes hemos llamado **esquemas** o **usuarios**.

Crear una base de datos implica indicar los archivos y ubicaciones que se van a utilizar además de otras indicaciones técnicas y administrativas. Es obvio que todo esto sólo lo puede realizar si se tiene privilegio de Administrador.

Con el estándar de SQL la instrucción a usar sería Create Database, pero cada SGBD tiene un procedimiento para crear las bases de datos. Crearíamos una base de datos con el nombre que se indique a continuación.

```
CREATE DATABASE NombredemiBasedeDatos;
```

Por ejemplo, a la base de datos que están creando Juan y Ana se le va a llamar RyMjuegos, entonces nos quedaría:

```
CREATE DATABASE RyMjuegos;
```

Hemos estado hablando de objetos de la base de datos, ahora veremos a qué nos referimos.
Según los estándares, **una base de datos es un conjunto de objetos que nos servirán para gestionar los datos**. Estos objetos están contenidos en esquemas y éstos a su vez suelen estar asociados a un usuario. De ahí que antes dijéramos que cada base de datos tiene un esquema que está asociado a un usuario.

Si quieres aprender a crear bases de datos con MySQL, aquí puedes aprender:
http://www.conclase.net/mysql/curso/index.php?cap=007

11.2.- Creación de tablas.

Tabla PARTIDAS

Código	Nombre	Cod_Juego	...
1	PARTIDA01	3	
2	PARTIDA02	2	
3	PARTIDA03	4	
4	PARTIDA04	3	
5	PARTIDA05	1	

¿Qué necesitamos para poder guardar los datos? Lo primero será definir los objetos donde vamos a agrupar esos datos. Los objetos básicos con los que trabaja SQL son las tablas, que como ya sabemos es un conjunto de filas y columnas cuya intersección se llama celda. Es ahí donde se almacenarán los elementos de información, los datos que queremos recoger.
Antes de crear la tabla es conveniente planificar algunos detalles:

- ✓ **Qué nombre** le vamos a dar **a la tabla**.
- ✓ **Qué nombre** le vamos a dar a cada una de las **columnas**.
- ✓ **Qué tipo y tamaño de datos** vamos a almacenar en cada columna.
- ✓ **Qué restricciones** tenemos sobre los datos.
- ✓ Alguna otra **información adicional** que necesitemos.

Y debemos tener en cuenta otras **reglas** que se deben cumplir **para los nombres de las tablas**:

- ✓ No podemos tener nombres de tablas duplicados en un mismo esquema (usuario).
- ✓ Deben comenzar por un carácter alfabético.
- ✓ Su longitud máxima es de 30 caracteres.
- ✓ Solo se permiten letras del alfabeto inglés, dígitos o el signo de guión bajo.
- ✓ No puede coincidir con las palabras reservadas de SQL (por ejemplo, no podemos llamar a una tabla WHERE).
- ✓ No se distingue entre mayúsculas y minúsculas.
- ✓ 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 SQL99 (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 "USUARIOS"y "Usuarios").

La sintaxis básica del comando que permite crear una tabla es la siguiente:

```
CREATE TABLE [esquema.] nombredeTabla (
   columna1 Tipo_Dato,
   columna2 Tipo_Dato,
   ...
   columnaN Tipo_Dato );
```

donde:

- ✓ **columna1, columna2, ..., columnaN** son los nombres de las columna que contendrá la tabla.

✓ **Tipo_Dato** indica el tipo de dato de cada columna.

Ana va a crear la primera tabla llamada USUARIOS con un solo campo de tipo VARCHAR:

```
CREATE TABLE USUARIOS (Nombre VARCHAR(25));
```

Recuerda que solo podrás crear tablas si posees los permisos necesarios para ello.

Durante nuestro aprendizaje vamos a tener que crear muchas tablas, para ello necesitaremos manejar los tipos de datos que utiliza Oracle. En el siguiente enlace tienes una relación de estos tipos y su descripción.
http://www.ajpdsoft.com/modules.php?name=News&file=article&sid=268

MySQL trabaja con otros tipos de datos. Si quieres conocerlos puedes entrar en este enlace.
http://www.desarrolloweb.com/articulos/1054.php

Señala cuales de las siguientes afirmaciones sobre los nombres de las tablas son ciertas:

- ☐ Puede haber nombres de tablas duplicados en la misma base de datos.
- ☑ **Su longitud máxima es de 30 caracteres.**
- ☑ **La tabla JUEGOS es la misma que la tabla Juegos.**
- ☑ **No puede coincidir con las palabras reservadas de SQL.**

11.3.- Restricciones.

Hay veces que necesitamos que un dato se incluya en una tabla de manera obligatoria, otras veces necesitaremos definir uno de los campos como llave primaria o ajena. Todo esto podremos hacerlo cuando definamos la tabla, además de otras opciones.

Una **restricción** es una condición que una o varias columnas deben cumplir obligatoriamente.

Cada restricción que creemos llevará un nombre, si no se lo ponemos nosotros lo hará Oracle o el SGBD que estemos utilizando. Es conveniente que le pongamos un nombre que nos ayude a identificarla y que sea único para cada esquema (usuario). 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. La sintaxis en SQL estándar es la siguiente:

```
CREATE TABLE NOMBRETABLA (
     Columna1 Tipo_Dato
          [CONSTRAINT nombredelarestricción]
          [NOT NULL]
          [UNIQUE]
          [PRIMARY KEY]
          [FOREIGN KEY]
          [DEFAULT valor]
          [REFERENCES nombreTabla [(columna [, columna ])]
          [ON DELETE CASCADE]]
          [CHECK condición],
     Columna2 Tipo Dato
          [CONSTRAINT nombredelarestricción]
          [NOT NULL]
          [UNIQUE]
          [PRIMARY KEY]
          [FOREIGN KEY]
          [DEFAULT valor]
          [REFERENCES nombreTabla [(columna [, columna ])]
          [ON DELETE CASCADE]]
          [CHECK condición],...);
```

Veamos un ejemplo:

```
CREATE TABLE USUARIOS (
    Login VARCHAR(15) CONSTRAINT usu_log_PK PRIMARY KEY,
    Password VARCHAR (8) NOT NULL,
    Fecha_Ingreso DATE DEFAULT SYSDATE);
```

Otra opción es definir las columnas de la tabla y después especificar las restricciones, de este modo podrás referir varias columnas en una única restricción.

En los siguientes apartados veremos cada una de las restricciones, su significado y su uso.

Oracle nos 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:**
- ➔ **PK = Primary Key.**
- ➔ **FK = Foreign Key.**
- ➔ **NN = Not Null.**
- ➔ **UK = Unique.**
- ➔ **CK = Check (validación).**

11.3.1.- Restricción NOT NULL.

Con esta restricción obligaremos a que esa columna tenga un valor o lo que es lo mismo, prohíbe los valores nulos para una columna en una determinada tabla.

Podremos ponerlo cuando creamos o modificamos el campo añadiendo la palabra `NOT NULL` después de poner el tipo de dato.

Si en la tabla USUARIOS queremos que el campo "F_Nacimiento" sea obligatorio ponerlo, nos quedaría así:

```
CREATE TABLE USUARIOS (
    F_Nacimiento DATE
    CONSTRAINT Usu_Fnac_NN NOT NULL);
```

o bien, de esta otra forma:

```
 CREATE TABLE USUARIOS (
    F_Nacimiento DATE NOT NULL);
```

Debemos tener cuidado con los valores nulos en las operaciones, ya que `1*NULL` es igual a `NULL`.

Si queremos que un campo no admita valores nulos, al crear la tabla pondremos después del nombre del campo y del tipo de datos:

- ○ NULL
- ○ VARCHAR
- ◉ **NOT NULL**

11.3.2.- Restricción UNIQUE.

Habrá ocasiones en la que nos interese que no se puedan repetir valores en la columna, en estos casos utilizaremos la restricción UNIQUE. Oracle crea un índice automáticamente cuando se habilita esta restricción y lo borra al deshabilitarla.

También para esta restricción tenemos dos posibles formas de ponerla, veámoslo con un ejemplo. Supongamos que el campo Login de nuestra tabla va a ser único. Lo incluiremos en la tabla que estamos creando. Nos quedaría así:

```
CREATE TABLE USUARIOS (
    Login VARCHAR2 (25)
        CONSTRAINT Usu_Log_UK UNIQUE);
```

Veamos otra forma:

```
CREATE TABLE USUARIOS (
    Login VARCHAR2 (25) UNIQUE);
```

También podemos poner esta restricción a varios campos a la vez, por ejemplo, si queremos que Login y correo electrónico sean únicos podemos ponerlo así:

```
CREATE TABLE USUARIOS (
    Login VARCHAR2 (25),
    Correo VARCHAR2 (25),
        CONSTRAINT Usuario_UK UNIQUE (Login, Correo));
```

Si te fijas, detrás del tipo de datos de Correo hay una coma, eso es así porque la restricción es independiente de ese campo y común a varios. Por eso después de `UNIQUE` hemos puesto entre paréntesis los nombres de los campos a los que afecta la restricción.

11.3.3.- Restricción PRIMARY KEY.

En el modelo relacional las tablas deben tener una clave primaria. Es evidente que cuando creamos la tabla tendremos que indicar a quién corresponde.

Sólo puede haber una clave primaria por tabla pero ésta puede estar formada por varios campos. Dicha clave podrá ser referenciada como clave ajena en otras tablas.

La clave primaria hace que los campos que forman sean `NOT NULL` y que los valores de los campos sean de tipo `UNIQUE`.

Veamos como quedaría si la clave fuese el campo Login:

- ✓ Si la clave la forma un único campo:

```
CREATE TABLE USUARIOS (
   Login VARCHAR2 (25) PRIMARY KEY);
```

- ✓ bien poniendo un nombre a la restricción:

```
CREATE TABLE USUARIOS (
     Login VARCHAR2 (25)
          CONSTRAINT Usu_log_PK PRIMARY KEY);
```

- ✓ Si la clave está formada por más de un campo, por ejemplo Nombre, Apellidos y Fecha de Nacimiento:

```
CREATE TABLE USUARIOS (
   Nombre VARCHAR2 (25),
   Apellidos VARCHAR2 (30),
   F_Nacimiento DATE,
   CONSTRAINT Usu_PK PRIMARY KEY(Nombre, Apellidos, F_Nacimiento));
```

11.3.4.- Restricción REFERENCES. FOREIGN KEY.

Ya vimos que las claves ajenas, secundarias o foráneas eran campos de una tabla que se relacionaban con la clave primaria (o incluso con la clave candidata) de otra tabla.

Cuando creemos la tabla tendremos que indicar de alguna forma quién es clave ajena. Lo haremos "haciendo referencia" a la tabla y los campos de donde procede.

En nuestra tabla vamos a tener una clave ajena procedente de la tabla PARTIDAS que será su Cod_Partida, por tanto tendremos que hacer referencia a éste:

```
 CREATE TABLE USUARIOS (
    Cod_Partida NUMBER(8)
        CONSTRAINT Cod_Part_FK
        REFERENCES PARTIDAS(Cod_Partida));
```

Si el campo al que hace referencia es clave principal en su tabla no es necesario indicar el nombre del campo:

```
CREATE TABLE USUARIOS (
    Cod Partida NUMBER(8)
        CONSTRAINT Cod_Part_FK
        REFERENCES PARTIDAS);
```

Si la definición de la clave ajena se pone al final, tendremos que colocar el texto `FOREIGN KEY` para especificar a qué campo se está refiriendo.

Vamos a verlo en el caso en que la clave ajena estuviera formada por Cod_Partida y Fecha de la partida de la tabla PARTIDAS:

```
CREATE TABLE USUARIOS (
    Cod_Partida NUMBER(8),
    F Partida DATE,
        CONSTRAINT Partida Cod F FK FOREIGN KEY (Cod Partida, F Partida)
        REFERENCES PARTIDAS);
```

Al relacionar campos necesitamos que el dato del campo que es clave ajena en una tabla (que llamaremos secundaria) previamente haya sido incluido en su tabla de procedencia donde es clave primaria o candidata. En nuestro ejemplo, cualquier código de partida que incluyamos en la tabla USUARIO, debería estar previamente en la tabla de la que procede, es decir, en la tabla PARTIDAS. **A esto se le llama Integridad Referencial.**

Esto puede crear algunos errores, pues puede ocurrir lo siguiente:

- ✓ Si hacemos referencia a una tabla que no está creada: Oracle buscará la tabla referenciada y al no encontrarla dará fallo. Esto se soluciona creando en primer lugar las tablas que no tengan claves ajenas.
- ✓ Si queremos borrar las tablas tendremos que proceder al contrario, borraremos las tablas que tengan claves ajenas antes.

Tenemos otras soluciones y es añadir tras la cláusula `REFERENCE`:

- ✓ `ON DELETE CASCADE`: te permitirá borrar todos los registros cuya clave ajena sea igual a la clave del registro borrado.
- ✓ `ON DELETE SET NULL`: colocará el valor `NULL` en todas las claves ajenas relacionadas con la borrada.

11.3.5.- Restricción DEFAULT Y VALIDACIÓN.

A veces es muy tedioso insertar siempre lo mismo en un campo. Imagínate que casi todos los jugadores fuesen de España y tenemos un campo País. ¿No sería cómodo asignarle un valor por defecto? Eso es lo que hace la restricción `DEFAULT`.

En nuestro ejemplo vamos a añadir a la tabla USUARIOS el campo País y le daremos por defecto el valor "España".

```
CREATE TABLE USUARIOS (
    Pais VARCHAR2(20) DEFAULT ' España ' );
```

En las especificaciones de `DEFAULT` vamos a poder añadir distintas expresiones: constantes, funciones SQL y variables.

Si queremos incluir en un campo la fecha actual, independientemente del día en el que estemos, podremos utilizar la función `SYSDATE` como valor por defecto:

```
CREATE TABLE USUARIOS (
      Fecha_ingreso DATE DEFAULT SYSDATE);
```

También vamos a necesitar que se compruebe que los valores que se introducen son adecuados para ese campo. Para ello utilizaremos `CHECK`.

Esta restricción comprueba que se cumpla una condición determinada al rellenar una columna. Dicha condición se puede construir con columnas de esa misma tabla.

Si en la tabla USUARIOS tenemos el campo Crédito y éste sólo puede estar entre 0 y 2000, lo especificaríamos así:

```
CREATE TABLE USUARIOS (
    Credito NUMBER(4) CHECK (Crédito BETWEEN 0 AND 2000));
```

Una misma columna puede tener varios `CHECK` asociados a ella, para ello ponemos varios `CONSTRAINT` seguidos y separados por comas.

Si queremos obtener una descripción de una tabla, sinonimo, paquete o función, podemos utilizar el comando DESCRIBE.
http://ora.u440.com/sqlplus/describe.html

Relaciona estos términos utilizados para las restricciones en la creación de tablas con su significado o función:

Términos.	Relación.	Función.
CHECK	1	**1**. Comprueba que los valores que se introducen son los adecuados para un campo.
DEFAULT	6	**2**. Designa a un campo como clave ajena.
PRIMARY KEY	5	**3**. Impide que un campo pueda contener valores nulos
FOREIGN KEY	2	**4**. Impide que se repitan valores para un campo.
NOT NULL	3	**5**. Designa a un campo como clave principal.
UNIQUE	4	**6**. Incluye un valor en un campo de forma predeterminada.

11.4.- Eliminación de tablas.

Cuando una tabla ya no es útil y no la necesitamos es mejor borrarla, de este modo no ocupará espacio y podremos utilizar su nombre en otra ocasión.

Para eliminar una tabla utilizaremos el comando `DROP TABLE`.

```
DROP TABLE NombreTabla [CASCADE CONSTRAINTS];
```

Esta instrucción borrará la tabla de la base de datos incluido sus datos (filas). También se borrará toda la información que existiera de esa tabla en el Diccionario de Datos.

La opción `CASCADE CONSTRAINTS` se puede incluir para los casos en que alguna de las columnas sea clave ajena en otra tabla secundaria, lo que impediría su borrado. Al colocar esta opción las restricciones donde es clave ajena se borrarán antes y a continuación se eliminará la tabla en cuestión.

Vamos a eliminar la tabla con la que hemos estado trabajando:

```
DROP TABLE USUARIOS ;
```

Ten cuidado al utilizar este comando, el borrado de una tabla es irreversible y no hay una petición de confirmación antes de ejecutarse.

Al borrar una tabla:

- ✓ Desaparecen todos sus datos
- ✓ Cualquier vista asociada a esa tabla seguirá existiendo pero ya no funcionará.

Oracle dispone de la orden `TRUNCATE TABLE` que te permitirá eliminar los datos (filas) de una tabla sin eliminar su estructura.

Y recuerda que solo podrás borrar aquellas tablas sobre las que tengas permiso de borrado.

11.5.- Modificación de tablas (I).

Es posible que después de crear una tabla nos demos cuenta que se nos ha olvidado añadir algún campo o restricción, quizás alguna de las restricciones que añadimos ya no es necesaria o tal vez queramos cambiar el nombre de alguno de los campos. ¿Es posible esto? Ahora veremos que sí y en casi todos los casos utilizaremos el comando `ALTER TABLE`.

- ✓ **Si queremos cambiar el nombre de una tabla:**

```
RENAME NombreViejo TO NombreNuevo;
```

- ✓ **Si queremos añadir columnas a una tabla: l**as columnas se añadirán al final de la tabla.

```
ALTER TABLE NombreTabla ADD
( ColumnaNueva1 Tipo Datos [Propiedades]
[, ColumnaNueva2 Tipo_Datos [Propiedades]
... );
```

- ✓ **Si queremos eliminar columnas de una tabla:** se eliminará la columna indicada sin poder deshacer esta acción. Además de la definición de la columna, se eliminarán todos los datos que contuviera. No se puede eliminar una columna si es la única que forma la tabla, para ello tendremos que borrar la tabla directamente.

```
ALTER TABLE NombreTabla DROP COLUMN (Columna1 [, Columna2, ...]);
```

- ✓ **Si queremos modificar columnas de una tabla:** podemos modificar el tipo de datos y las propiedades de una columna. Todos los cambios son posibles si la tabla no contiene datos. En general, si la tabla no está vacía podremos aumentar la longitud de una columna, aumentar o disminuir en número de posiciones decimales en un tipo `NUMBER`, reducir la anchura siempre que los datos no ocupen todo el espacio reservado para ellos.

```
 ALTER TABLE NombreTabla MODIFY
(Columna1 TipoDatos [propiedades] [, columna2 TipoDatos [propiedades] ...] );
```

✓ **Si queremos renombrar columnas de una tabla:**

```
ALTER TABLE NombreTabla RENAME COLUMN NombreAntiguo TO NombreNuevo;
```

Tenemos la siguiente tabla creada:

```
 CREATE TABLE USUARIOS (
    Credito NUMBER(4) CHECK (Crédito BETWEEN 0 AND 2000));
```

Nos gustaría incluir una nueva columna llamada User que será tipo texto y clave primaria:

```
ALTER TABLE USUARIO ADD
    (User VARCHAR(10) PRIMARY KEY);
```

Nos damos cuenta que ese campo se llamaba Login y no User, vamos a cambiarlo:

```
ALTER TABLE USUARIO RENAME COLUMN User TO Login;
```

Ejercicio resuelto

Tenemos creada la siguiente tabla:

```
CREATE TABLE EMPLEADOS (
    Cod_Cliente VARCHAR(5) PRIMARY KEY,
    Nombre VARCHAR(10),
    Apellidos VARCHAR(25),
    Sueldo NUMBER(2));
```

Ahora queremos poner una restricción a sueldo para que tome valores entre 1000 y 1200, ¿cómo lo harías?

Respuesta:

```
ALTER TABLE EMPLEADOS MODIFY (Sueldo NUMBER(2) CHECK (Sueldo BETWEEN 1000 AND 1200));
```

11.5.1.- Modificación de tablas (II).

Utilizando el comando ALTER TABLE, podemos modificar las restricciones o bien eliminarlas:

✓ **Si queremos borrar restricciones:**

```
ALTER TABLA NombreTabla DROP CONSTRAINT NombreRestriccion;
```

✓ **Si queremos modificar el nombre de las restricciones:**

```
ALTER TABLE NombreTabla RENAME CONSTRAINT NombreViejo TO NombreNuevo;
```

✓ **Si queremos activar o desactivar restricciones:**

A veces es conveniente desactivar temporalmente una restricción para hacer pruebas o porque necesitemos saltarnos esa regla. Para ello usaremos esta sintaxis:

```
ALTER TABLE NombreTabla DISABLE CONSTRAINT NombreRestriccion [CASCADE];
```

La opción `CASCADE` desactiva las restricciones que dependan de ésta.

Para activar de nuevo la restricción:

```
ALTER TABLE NombreTabla ENABLE CONSTRAINT NombreRestriccion [CASCADE];
```

Puede ocurrir que no hayamos puesto nombre a las restricciones o bien que lo hiciéramos pero no lo recordemos. Sería interesante que se pudiera consultar en algún lado.

http://ubuntulife.wordpress.com/2009/03/16/tip-ver-todas-las-constraints-en-oracle/

11.6.- Creación y eliminación de índices

Sabemos que crear índices ayuda a la localización más rápida de la información contenida en las tablas. Ahora aprenderemos a crearlos y eliminarlos:

```
CREATE INDEX NombreIndice ON NombreTabla (Columna1 [, Columna2 ...]);
```

No es aconsejable que utilices campos de tablas pequeñas o que se actualicen con mucha frecuencia. Tampoco es conveniente si esos campos no se usan en consultas de manera frecuente o en expresiones.

El diseño de indices es un tema bastante complejo para los Administradores de Bases de Datos, ya que una mala elección ocasiona ineficiencia y tiempos de espera elevados. Un uso excesivo de ellos puede dejar a la Base de Datos colgada simplemente con insertar alguna fila.

Para eliminar un índice es suficiente con poner la instrucción:

```
DROP INDEX NombreIndice;
```

La mayoría de los índices se crean de manera implícita cuando ponemos las restricciones `PRIMARY KEY`, `FOREIGN KEY` o `UNIQUE`.

Ejercicio resuelto

Tenemos creada la siguiente tabla:

```
CREATE TABLE EMPLEADOS (
    Cod Cliente VARCHAR(5) PRIMARY KEY,
    Nombre VARCHAR(10),
    Apellidos VARCHAR(25),
    Sueldo NUMBER(2));
```

Crea un índice con el campo Apellidos, luego elimínalo.

Respuesta:

```
CREATE INDEX miIndice ON EMPLEADOS (Apellidos);
    DROP INDEX miIndice;
```

12.- Lenguaje de control de datos (DCL).

Caso práctico

Juan cree que será necesario conocer quiénes acceden a la base de datos para poder crearles sus contraseñas y darles los permisos necesarios, de manera que la administración de la base quede en manos de quien corresponde, y el sistema sea seguro. No quiere ni imaginarse que quedara algún cabo suelto, y cualquier usuario con algo de conocimientos pudiera acceder sin consentimiento y con los permisos suficientes, como para manipular los datos a su antojo.

Ya hemos visto que necesitamos una cuenta de usuario para acceder a los datos de una base de datos. Las claves de acceso se establecen cuando se crea el usuario y pueden ser modificados por el Administrador o por el propietario de dicha clave. La Base de Datos almacena encriptadas las claves en una tabla del diccionario llamada `DBA_USERS`.

¿Cómo se crean los usuarios? La sintaxis es:

```
CREATE USER NombreUsuario
IDENTIFIED BY ClaveAcceso
[DEFAULT TABLESPACE tablespace ]
[TEMPORARY TABLESPACE tablespace]
[QUOTA int {K | M} ON tablespace]
[QUOTA UNLIMITED ON tablespace]
[PROFILE perfil];
```

donde:

- ✓ `CREATE USER`: crea un nombre de usuario que será identificado por el sistema.
- ✓ `IDENTIFIED BY`: permite dar una clave de acceso al usuario creado.
- ✓ `DEFAULT TABLESPACE`: asigna a un usuario el Tablespace por defecto para almacena los objetos que cree. Si no se asigna ninguna, será `SYSTEM`.
- ✓ `TEMPORARY TABLESPACE`: especifica el nombre del Tablespace para trabajos temporales. Por defecto será `SYSTEM`.
- ✓ `QUOTA`: asigna un espacio en Megabytes o Kilobytes en el Tablespace asignado. Si no se especifica el usuario no tendrá espacio y no podrá crear objetos.
- ✓ `PROFILE`: asigna un perfil al usuario. Si no se especifica se asigna el perfil por defecto.

Recuerda que para crear usuarios debes tener una cuenta con privilegios de Administrador.

Para ver todos los usuarios creados utilizamos las vistas `ALL_USERS` y `DBA_USERS`. Y para ver en mi sesión los usuarios que existen pondría: `DESC SYS.ALL_USERS;`
Practiquemos un poco con este comando. Creemos una cuenta de usuario limitado, que no tenga derecho ni a guardar datos ni a crear objetos, más tarde le daremos permisos:

```
CREATE USER UsuarioLimitado IDENTIFIED BY passworddemiusuariolimitado ;
```

Podemos modificar usuarios mediante el comando ALTER USER, cuya sintaxis es la siguiente:

```
ALTER USER NombreUsuario
IDENTIFIED BY clave acceso
[DEFAULT TABLESPACE tablespace ]
[TEMPORARY TABLESPACE tablespace]
[QUOTA int {K | M} ON tablespace]
[QUOTA UNLIMITED ON tablespace]
[PROFILE perfil];
```

Un usuario sin privilegios de Administrador únicamente podrá cambiar su clave de acceso.
Para eliminar o borrar un usuario utilizamos el comando `DROP USER` con la siguiente sintaxis:

```
DROP USER NombreUsuario [CASCADE];
```

La opción `CASCADE` borra todos los objetos del usuario antes de borrarlo. Sin esta opción no nos dejaría eliminar al usuario si éste tuviera tablas creadas.

12.1.- Permisos (I).

Ningún usuario puede llevar a cabo una operación si antes no se le ha concedido el permiso para ello. En el apartado anterior hemos creado un usuario para iniciar sesión, pero si con él intentáramos crear una tabla veríamos que no tenemos permisos suficientes para ello.

Para poder acceder a los objetos de una base de datos necesitas tener privilegios (permisos). Éstos se pueden agrupar formando **roles**, lo que simplificará la administración. Los roles pueden activarse, desactivarse o protegerse con una clave. Mediante los roles podemos gestionar los comandos que pueden utilizar los usuarios. Un permiso se puede asignar a un usuario o a un rol.

Un privilegio o permiso se especifica con el comando `GRANT` (conceder).

Si se dan privilegios **sobre los objetos**:

```
GRANT {privilegio objeto [, privilegio objeto]...|ALL|[PRIVILEGES]}
ON [usuario.]objeto
FROM {usuario1|rol1|PUBLIC} [,{usuario2|rol2|PUBLIC] ...
[WITH GRANT OPTION];
```

donde:

- ✓ `ON` especifica el objeto sobre el que se conceden los privilegios.
- ✓ `TO` señala a los usuarios o roles a los que se conceden privilegios.
- ✓ `ALL` concede todos los privilegios sobre el objeto especificado.
- ✓ `[WITH GRANT OPTION]` permite que el receptor del privilegio se lo asigne a otros.
- ✓ `PUBLIC` hace que un privilegio esté disponible para todos los usuarios.

En el siguiente ejemplo Juan ha accedido a la base de datos y ejecuta los siguientes comandos:

- ✓ `GRANT INSERT TO Usuarios TO Ana;` (permitirá a Ana insertar datos en la tabla Usuarios)
- ✓ `GRANT ALL ON Partidas TO Ana;` (Juan concede todos los privilegios sobre la tabla Partidas a Ana)

Los privilegios **de sistema** son los que dan derecho a ejecutar comandos SQL o acciones sobre objetos de un tipo especificado. Existen gran cantidad de privilegios distintos.

La sintaxis para dar este tipo de privilegios la tienes aquí:

```
GRANT {Privilegio1 | rol1 } [, privilegio2 | rol2}, ...]
TO {usuario1 | rol1| PUBLIC} [, usuario2 | rol2 | PUBLIC} ... ]
[WITH ADMIN OPTION];
```

Donde

- ✓ `TO` señala a los usuarios o roles a los que se conceden privilegios.
- ✓ `WITH ADMIN OPTION` es una opción que permite al receptor de esos privilegios que pueda conceder esos mismos privilegios a otros usuarios o roles.
- ✓ `PUBLIC` hace que un privilegio esté disponible para todos los usuarios.

Veamos algunos ejemplos:

```
GRANT CONNECT TO Ana;
```

Concede a Ana el rol de `CONNECT` con todos los privilegios que éste tiene asociados.

```
GRANT DROP USER TO Ana WITH ADMIN OPTION;
```

Concede a Ana el privilegio de borrar usuarios y que ésta puede conceder el mismo privilegio de borrar usuarios a otros.

Si quieres conocer más sobre permisos y objetos sobre los que se conceden privilegios, visita este enlace:

http://www.redcientifica.com/oracle/c0004p0004.html

12.1.1.- Permisos (II).

Hasta ahora hemos aprendido a conceder permisos o privilegios. Será importante aprender a retirarlos.

Con el comando `REVOKE` se retiran los privilegios:

- ✓ Sobre **objetos**:

```
REVOKE {privilegio_objeto [, privilegio_objeto]...|ALL|[PRIVILEGES]}
ON [usuario.]objeto
FROM {usuario|rol|PUBLIC} [,{usuario|rol|PUBLIC] ...;
```

- ✓ **Del sistema o roles a usuarios**:

```
REVOKE {privilegio stma | rol} [, {privilegio stma | rol}]...|ALL|[PRIVILEGES]}
ON [usuario.]objeto
FROM {usuario|rol|PUBLIC} [,{usuario|rol|PUBLIC] ...;
```

Juan va a quitar el permiso de seleccionar y de actualizar sobre la tabla Usuarios a Ana:

```
REVOKE SELECT, UPDATE ON Usuarios FROM Ana;
```

y va a quitarle el permiso de eliminar usuarios:

```
REVOKE DROP USER FROM Ana;
```

Asocia cada comando con su uso.

Usuarios y permisos.

Comando	Relación	Función
CREATE USER	3	1. Se utiliza para dar permisos a los usuarios o roles.
DROP USER	2	2. Se utiliza para eliminar usuarios.
GRANT	1	3. Se utiliza para crear usuarios.
REVOKE	4	4. Se utiliza para quitar permisos.

TEMA 3

INDICE

1. Análisis y diseño de bases de datos. 3
2.- ¿Qué es el Modelo E/R? 4
3.- Entidades. 5
3.1.- Tipos: fuertes y débiles. 5
4.- Atributos. 7
4.1.- Tipos de atributos 7
4.2.- Claves. 8
4.3.- Atributos de una relación. 9
5.- Relaciones. 11
5.1.- Grado de una relación. 11
5.2.- Cardinalidad de relaciones. 12
5.3.- Cardinalidad de entidades. 13
6.- Simbología del modelo E/R. 15
7.- El modelo E/R Extendido. 16
7.1.- Restricciones en las relaciones. 16
7.2.- Generalización y especialización. 17
Ejercicio resuelto 19
7.3.- Agregación. 19
8.- Elaboración de diagramas E/R. 21
8.1.- Identificación de entidades y relaciones. 21
8.2.- Identificación de atributos, claves y jerarquías. 22
8.3.- Metodologías. 23
8.4.- Redundancia en diagramas E/R. 24
8.5.- Propiedades deseables de un diagrama E/R. 25
9.- Paso del diagrama E/R al modelo relacional. 27
9.1.- Simplificación previa de diagramas. 28
10.- Paso del diagrama E/R al Modelo Relacional. 30
Ejercicio resuelto 32
11.- Normalización de modelos relacionales. 33
11.1.- Tipos de dependencias. 34
Ejercicio resuelto 34
11.2.- Formas Normales. 35
1ª Forma Normal 35
2ª Forma Normal 35
3ª Forma Normal 35
Forma Normal de Boyce Codd 36
Otras formas normales 36
Ejercicio resuelto 36

Interpretación de diagramas entidad/relación.

Caso práctico

__Ada__ está analizando la manera en la que __Juan__ y __María__ han comenzando a construir la base de datos que sustentará el sitio web de juegos online. Parece que la aplicación del modelo relacional está marchando correctamente, aunque le interesa que el proceso se realice siguiendo un método lo más estandarizado posible y que les ofrezca independencia del SGBD que escojan.

De este modo, podrán planificar el desarrollo de cada una de las fases y ajustar mejor los tiempos dedicados a cada una de ellas.

Como se ha descrito en unidades anteriores, un modelo de datos es una colección de herramientas conceptuales que permiten llevar a cabo la descripción de los datos, sus relaciones, su semántica o significado y las restricciones que se les pueden aplicar. Sabemos que los SGBD cuentan con una arquitectura que simplifica, a los diferentes usuarios de la base de datos, su labor. El objetivo fundamental de esta arquitectura es separar los programas de aplicación de la base de datos física, proponiendo tres niveles de abstracción: **nivel interno o físico, nivel lógico o conceptual y nivel externo o de visión del usuario**.

1. Análisis y diseño de bases de datos.

El **Nivel lógico o conceptual** describe la estructura completa de la base de datos a través de lo que llamamos **Esquema Conceptual,** que se encarga de representar la información de una manera totalmente independiente del Sistema Gestor de Base de Datos.

Cuando hemos de desarrollar una base de datos se distinguen claramente dos fases de trabajo: **Análisis** y **Diseño**. En la siguiente tabla te describimos las etapas que forman parte de cada fase.

Pasos de las fases de Análisis y de Dsiseño	
Fase de Análisis	**Fase de Diseño**
Análisis de entidades: Se trata de localizar y definir las entidades y sus atributos.	Diseño de tablas.
Análisis de relaciones: Se definirán las relaciones existentes entre entidades.	Normalización.
Obtención del Esquema Conceptual a través del modelo E-R.	Aplicación de retrodiseño, si fuese necesario.
Fusión de vistas: Se reúnen en un único esquema todos los esquemas existentes en función de las diferentes vistas de cada perfil de usuario.	Diseño de transacciones: localización del conjunto de operaciones o transacciones que operarán sobre el esquema conceptual.
Aplicación del enfoque de datos relacional.	Diseño de sendas de acceso: se formalizan los métodos de acceso dentro de la estructura de datos.

Llevando a cabo una correcta fase de análisis estaremos dando un paso determinante en el desarrollo de nuestras bases de datos. El hecho de saltarse el esquema conceptual conlleva un problema de pérdida de información respecto al problema real a solucionar. El esquema conceptual debe reflejar todos los aspectos relevantes del mundo real que se va a modelar.

Para la realización de esquemas que ofrezcan una visión global de los datos, Peter Chen en 1976 y 1977 presenta dos artículos en los que se describe el **modelo Entidad/Relación** (entity/relationship). Con el paso del tiempo, este modelo ha sufrido modificaciones y mejoras. Actualmente, el modelo **entidad/relación extendido (ERE)** es el más aceptado, aunque existen variaciones que hacen que este modelo no sea totalmente un estándar. Ambos modelos serán estudiados a lo largo de esta unidad.

2.- ¿Qué es el Modelo E/R?

Caso práctico

*—**María**, ¿Si un carpintero recibe un encargo de un mueble, en qué crees que se basa para fabricarlo? —Pregunta **Ada**.*

*Levantando la vista de la pantalla de su ordenador, **María** contesta: —Supongo que un esquema o croquis, a veces hay detalles que es necesario consultar en la documentación, porque todo no es posible memorizarlo.*

*Juan interviene: —Me temo que ya sé por dónde van los tiros, **Ada**. ¿Con esa pregunta te estás refiriendo a los esquemas gráficos que se deben crear para la construcción de bases de datos?*
***Ada** sonríe y hace un gesto para que ambos se acerquen: —¿Sabéis lo qué es el modelo Entidad – Relación?*

Es una herramienta de referencia para la representación conceptual de problemas del mundo real. Su objetivo principal, facilitar el diseño de bases de datos permitiendo la especificación de un esquema que representa la estructura lógica completa de una base de datos. Este esquema partirá de las descripciones textuales de la realidad, que establecen los requerimientos del sistema, buscando ser lo más fiel posible al comportamiento del mundo real para modelarlo.

El modelo de datos E-R representa el significado de los datos, es un modelo semántico. De ahí que no esté orientado a ningún sistema físico concreto y tampoco tiene un ámbito informático puro de aplicación, ya que podría utilizarse para describir procesos de producción, estructuras de empresa, etc. Además, las características actuales de este modelo favorecen la representación de cualquier tipo de sistema y a cualquier nivel de abstracción o refinamiento, lo cual da lugar a que se aplique tanto a la representación de problemas que vayan a ser tratados mediante un sistema informatizado, como manual.

Gracias al modelo Entidad-Relación, creado por **Peter Chen** en los años setenta, se puede representar el mundo real mediante una serie de símbolos y expresiones determinados. El modelo de datos Entidad/Relación (E/R ó E-R) está basado en una percepción consistente en objetos básicos llamados **entidades** y **relaciones** entre estos objetos, estos y otros conceptos se desarrollan a continuación.

La información con la que el modelo Entidad-Relación trabaja ha de ser lo más detallada y fiel posible a la realidad del problema a representar.

Verdadero ◉ Falso ○

3.- Entidades.

Caso práctico

*—¿Cada una de las tablas que hemos estado generando equivale a una entidad en el modelo E/R? —Pregunta **Juan**.*

*—Algunas de ellas corresponden a entidades y otras a relaciones, depende del problema a resolver. Por ejemplo, la tabla USUARIO sí se correspondería con una entidad. Además, hay que tener cuidado a la hora de identificar entidades porque algunas veces podemos confundir entidades con atributos y viceversa —responde **Ada**.*

Para los miembros de BK Programación va a ser necesario que conozcan bien cómo se aplica este modelo si quieren que el proceso de creación de bases de datos sea correcto.

Si utilizamos las bases de datos para guardar información sobre cosas que nos interesan o que interesan a una organización, ¿No crees que hay que identificar esas cosas primero para poder guardar información sobre ellas? Para ello, vamos a describir un primer concepto, el de **Entidad**.

Una **entidad** puede ser un objeto físico, un concepto o cualquier elemento que queramos modelar, que tenga importancia para la organización y del que se desee guardar información. Cada entidad debe poseer alguna característica, o conjunto de ellas, que lo haga único frente al resto de objetos. Por ejemplo, podemos establecer una entidad llamada ALUMNO que tendrá una serie de características. El alumnado podría ser distinguido mediante su número de identificación escolar (NIE), por ejemplo.

Entidad: objeto real o abstracto, con características diferenciadoras capaces de hacerse distinguir de otros objetos.

¿Ponemos otro ejemplo? Supongamos que tienes que desarrollar el esquema conceptual para una base de datos de mapas de montaña, los elementos: camping, pista forestal, valle, río, pico, refugio, etc., son ejemplos de posibles entidades. A la hora de identificar las entidades, hemos de pensar en nombres que tengan especial importancia dentro del lenguaje propio de la organización o sistema que vaya a utilizar dicha base de datos. Pero no siempre una entidad puede ser concreta, como un camping o un río, en ocasiones puede ser abstracta, como un préstamo, una reserva en un hotel o un concepto.

Un **conjunto de entidades** serán un grupo de entidades que poseen las mismas características o propiedades. Por ejemplo, al conjunto de personas que realizan reservas para un hotel de montaña determinado, se les puede definir como el conjunto de entidades cliente. El conjunto de entidades río, representará todos los ríos existentes en una determinada zona. Por lo general, se suele utilizar el término entidad para identificar conjuntos de entidades. Cada elemento del conjunto de entidades será una ocurrencia de entidad.

Si establecemos un símil con la Programación Orientada a Objetos, podemos decir que el concepto de **entidad** es análogo al de **instancia de objeto** y que el concepto de **conjunto de entidades** lo es al de **clase**.

En el modelo Entidad/Relación, la representación gráfica de las entidades se realiza mediante el nombre de la entidad encerrado dentro de un rectángulo. A continuación se muestra la representación de la entidad CLIENTE.

3.1.- Tipos: fuertes y débiles.

Las entidades pueden ser clasificadas en dos grupos:

a. **Entidades Fuertes o Regulares:**
Son aquellas que tienen existencia por sí mismas, es decir, su existencia no depende de la existencia de otras entidades. Por ejemplo, en una base de datos hospitalaria, la existencia de instancias concretas de la entidad DOCTOR no depende de la existencia de instancias u objetos de la entidad PACIENTE. En el modelo E/R las entidades fuertes se representan como hemos indicado anteriormente, con el nombre de la entidad encerrado dentro de un rectángulo.

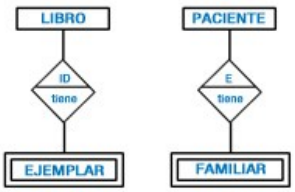
b. **Entidades débiles:**
Son aquellas cuya existencia depende de la existencia de otras instancias de entidad. Por ejemplo, consideremos las entidades EDIFICIO y AULA. Supongamos que puede haber aulas identificadas con la misma numeración pero en edificios diferentes. La numeración de cada aula no identificará completamente cada una de ellas. Para poder identificar completamente un aula es necesario saber también en qué edificio está localizada. Por tanto, la existencia de una instancia de una entidad débil depende de la existencia de una instancia de la entidad fuerte con la que se relaciona.

Entidad Débil: Es un tipo de entidad cuyas propiedades o atributos no la identifican completamente, sino que sólo la identifican de forma parcial. Esta entidad debe participar en una relación que ayude a identificarla.

En el modelo E/R una entidad débil se representa con el nombre de la entidad encerrado en un rectángulo doble. En el gráfico se muestra la representación de la entidad AULA.

Las entidades débiles presentan dos **tipos de dependencia**:

- ✓ **Dependencia en existencia**: entre entidades, si desaparece una instancia de entidad fuerte desaparecerán las instancias de entidad débiles que dependan de la primera. La representación de este tipo de dependencia incluirá una E en el interior de la relación débil.
- ✓ **Dependencia en identificación**: debe darse una dependencia en existencia y además, una ocurrencia de la entidad débil no puede identificarse por sí misma, debiendo hacerse mediante la clave de la entidad fuerte asociada. La representación de este tipo de dependencia incluirá una ID en el interior de la relación débil.



Tanto las entidades fuertes como las débiles se nombran habitualmente con sustantivos en singular.

Puede ser que haya algunos conceptos que aún no hemos desarrollado (relación, atributo y clave) y que se están utilizando para describir los tipos de dependencias, no te preocupes, en los siguientes epígrafes te los describimos claramente.

Identifica cuál de las siguientes entidades no podría ser considerada como entidad débil:

- (•) **PROVEEDOR (perteneciente a una base de datos de gestión de stocks).**
- ( ) PAGO (perteneciente a una base de datos bancaria).
- ( ) FAMILIAR (perteneciente a una base de datos hospitalaria).

Efectivamente, esta entidad puede existir por sí misma sin depender de otras ocurrencias de entidad. Además, posee propiedades o atributos propios que la identifican frente a otras ocurrencias de la misma entidad.

4.- Atributos.

Caso práctico

__Juan__ muestra a María qué atributos han creado para la tabla JUEGOS, pero al aplicar el modelo Entidad-Relación se ha dado cuenta de que le falta algún atributo más.

__María__ esta dibujando la entidad JUEGOS y sus atributos asociados. Ahora va a añadir gráficamente un atributo que recoja la productora de software asociada a cada juego.

¿Cómo guardamos información de cada entidad? A través de sus **atributos**. Las entidades se representan mediante un conjunto de **atributos**. Éstos describen características o propiedades que posee cada miembro de un conjunto de entidades. El mismo atributo establecido para un conjunto de entidades o, lo que es lo mismo, para un tipo de entidad, almacenará información parecida para cada ocurrencia de entidad. Pero, cada ocurrencia de entidad tendrá su propio valor para cada atributo.

Atributo: Cada una de las propiedades o características que tiene un tipo de entidad o un tipo de relación se denomina atributo; los atributos toman valores de uno o varios dominios.

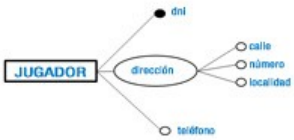
Por tanto, un atributo se utilizará para guardar información sobre alguna característica o propiedad de una entidad o relación. Ejemplos de atributos pueden ser: altura, color, peso, DNI, fecha, etc. todo dependerá de la información que sea necesaria almacenar.

En el modelo Entidad/Relación los atributos de una entidad son representados mediante el nombre del atributo rodeado por una elipse. La elipse se conecta con la entidad mediante una línea recta. Cada atributo debe tener un nombre único que haga referencia al contenido de dicho atributo. Los nombres de los atributos se deben escribir en letra minúscula. En el gráfico se representan algunos de los atributos para la entidad PACIENTE.

Al conjunto de valores permitidos para un atributo se le denomina **dominio**. Todos los posibles valores que puede tomar un atributo deberán estar dentro del dominio. Varios atributos pueden estar definidos dentro del mismo dominio. Por ejemplo, los atributos nombre, apellido primero y apellido segundo de la entidad PACIENTE, están definidos dentro del dominio de cadenas de caracteres de una determinada longitud.

Aunque los dominios suelen ser amplios (números enteros, reales, cadenas de caracteres, etc.), a la hora de llevar a cabo el desarrollo de una base de datos, es mejor establecer unos límites adecuados para que el sistema gestor de la base de datos lleve a cabo las verificaciones oportunas en los datos que se almacenen, garantizando así la integridad de éstos.

4.1.- Tipos de atributos.



¿Todos los atributos son iguales? Claro que no. Existen varias características que hacen que los atributos asociados a una entidad o relación sean diferentes, los clasificaremos según varios criterios.

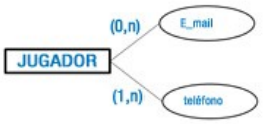
a. **Atributos obligatorios y opcionales. Atributo obligatorio**: es aquél que ha de estar siempre definido para una entidad o relación. Por ejemplo, para la entidad JUGADOR será necesario tener algún atributo que identifique cada ocurrencia de entidad, podría ser su DNI. Una clave o llave es un atributo obligatorio. **Atributo opcional**: es aquél que podría ser definido o no para la entidad. Es decir, puede haber ocurrencias de entidad para las que ese atributo no esté definido o no tenga valor.
b. Atómicos o compuestos.

Atributo simple o atómico: es un atributo que no puede dividirse en otras partes o atributos, presenta un único elemento. No es posible extraer de este atributo partes más pequeñas que puedan tener significado. Un ejemplo de este tipo de atributos podría ser el atributo dni de la entidad JUGADOR del gráfico.

Atributo compuesto: son atributos que pueden ser divididos en subpartes, éstas constituirán otros atributos con significado propio. Por ejemplo, la dirección del jugador podría considerarse como un atributo compuesto por la calle, el número y la localidad.

c. **Atributos monovaluados o multivaluados.**

Atributo monovaluado: es aquél que tiene un único valor para cada ocurrencia de entidad. Un ejemplo de este tipo de atributos es el dni.



Atributo multivaluado: es aquél que puede tomar diferentes valores para cada ocurrencia de entidad. Por ejemplo, la dirección de e-mail de un empleado podría tomar varios valores para alguien que posea varias cuentas de correo. En este tipo de atributos hay que tener en cuenta los siguientes conceptos:

- ✓ La **cardinalidad de un atributo** indica el número mínimo y el número máximo de valores que puede tomar para cada ejemplar de la entidad o relación a la que pertenece.
- ✓ La **cardinalidad mínima** indica la cantidad de valores del atributo que debe existir para que la entidad sea válida. Este número casi siempre es **0** o **1**. Si es **0**, el atributo podría no contener ningún valor y si es **1**, el atributo debe tener un valor.
- ✓ La **cardinalidad máxima** indica la cantidad máxima de valores del atributo que puede tener la entidad. Por lo general es **1** o **n**. Si es **1**, el atributo no puede tener más que un valor, si es **n**, el atributo puede tener múltiples valores y no se especifica la cantidad absoluta.

El atributo E_mail de la figura, puede ser opcional y no contener ningún valor, o bien, almacenar varias cuentas de correo electrónico de un jugador. Como ves, la cardinalidad representada en la imagen es (0,n).

d. **Atributos derivados o almacenados**: el valor de este tipo de atributos puede ser obtenido del valor o valores de otros atributos relacionados. Un ejemplo clásico de atributo derivado es la edad. Si se ha almacenado en algún atributo la fecha de nacimiento, la edad es un valor calculable a partir de dicha fecha.

Rellena los huecos en blanco con los conceptos adecuados.

Si en nuestra base de datos tenemos una entidad USUARIO, los atributos password y login deberán ser atributos obligatorios ya que son imprescindibles para iniciar o jugar partidas. En cambio, un posible atributo ranking que indique en qué posición se encuentra el usuario entre todos los jugadores, podría considerarse un atributo derivado si tenemos en cuenta la puntuación obtenida por cada usuario.

4.2.- Claves.

En el apartado anterior hablábamos de un tipo de atributo especial obligatorio, **las claves o llaves**. Ahora es el momento de abordar con mayor detalle este concepto.

Está claro que es necesario identificar correctamente cada ocurrencia de entidad o relación, de este modo el tratamiento de la información que se almacena podrá realizarse adecuadamente. Esta distinción podría llevarse a cabo tomando todos los valores de todos los atributos de una entidad o relación. Pero, en algunas ocasiones, sabemos que puede no ser necesario utilizar todos, bastando con un subconjunto de ellos. Aunque puede ocurrir que ese subconjunto tenga idénticos valores para varias entidades, por lo que cualquier subconjunto no será válido.

Por tanto, los valores de los atributos de una entidad deben ser tales que permitan **identificar unívocamente** a la entidad. En otras palabras, no se permite que ningún par de entidades tengan exactamente los mismos valores de sus atributos. Teniendo en cuenta esto, presta atención a los siguientes conceptos:

Superclave (Superllave): Es cualquier conjunto de atributos que permite identificar de forma única a una ocurrencia de entidad. Una superclave puede tener atributos no obligatorios, es decir, que no identificarían por si solos una ocurrencia de entidad.

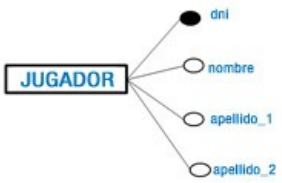
Clave candidata: Si de una superclave no es posible obtener ningún subconjunto que sea a su vez superclave, decimos que dicha superclave es clave candidata.

Clave primaria (Primary Key): También llamada llave primaria o clave principal. De todas las claves candidatas, el diseñador de la base de datos ha de escoger una, que se denominará clave principal o clave primaria. La clave primaria es un atributo o conjunto de ellos, que toman valores únicos y distintos para cada ocurrencia de entidad, identificándola unívocamente. No puede contener valores nulos.

Claves alternativas: son el resto de claves candidatas que no han sido escogidas como clave primaria.

La representación en el modelo Entidad/Relación de las claves primarias puede realizarse de dos formas:

- ✓ Si se utilizan elipses para representar los atributos, se subrayarán aquellos que formen la clave primaria.
- ✓ Si se utilizan círculos para representar los atributos, se utilizará un círculo negro en aquellos que formen la clave primaria.



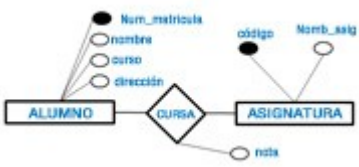
Sea la entidad TRABAJADOR, con los atributos nombre, apellido_1, apellido_2, dni, numero_afiliacion_ss, fecha_nacimiento y código_empresa. ¿Los atributos nombre, apellido_1 y apellido_2 podrían formar una clave candidata?

- ○ Sí, y podrían ser elegidos para ser la clave primaria de TRABAJADOR.
- ○ No, para esta entidad sólo el atributo dni será la clave primaria.
- ◉ **No, si tenemos en cuenta que puede haber varios trabajadores con el mismo nombre y apellidos.**

Efectivamente, los atributos dni y numero_afiliacion_ss, serían dos claves candidatas adecuadas. Si escogemos dni como clave primaria, numero_afiliacion_ss quedaría como clave alternativa.

4.3.- Atributos de una relación.

Una relación puede también tener atributos que la describan. Para ilustrar esta situación, observa el siguiente ejemplo.



Consideremos la relación CURSA entre las entidades ALUMNO y ASIGNATURA. Podríamos asociar a la relación CURSA un atributo nota para especificar la nota que ha obtenido un alumno/a en una determinada asignatura.

Otro ejemplo típico son las relaciones que representan **históricos**. Este tipo de relaciones suele constar de datos como fecha y hora. Cuando se emite una factura a un cliente o se le facilita un

duplicado de la misma, es necesario registrar el momento en el que se ha realizado dicha acción. Para ello, habrá que crear un atributo asociado a la relación entre la entidad CLIENTE y FACTURA que se encargue de guardar la fecha de emisión.

En el modelo Entidad/Relación la representación de atributos asociados a relaciones es exactamente igual a la que utilizábamos para entidades. Podremos utilizar una elipse con el nombre del atributo en su interior, conectada con una línea a la relación, o bien, un círculo blanco conectado con una línea a la relación y junto a él, el nombre del atributo. En el gráfico puedes ver esta segunda representación.

5.- Relaciones.

Caso práctico

__María__ ha identificado claramente las entidades y atributos que van a intervenir en su esquema, pero duda a la hora de representar cómo se van a relacionar dichas entidades.

__Ada__ le indica que es muy importante leer muy bien el documento de especificación de requerimientos del caso real a modelar, ya que de éste se desprenderán las particularidades de las relaciones entre las entidades que acaba de identificar.

—Representar una relación gráficamente en el modelo E/R es sencillo, pero lo interesante es dotar a esa representación de los elementos gráficos adecuados que reflejen fielmente cómo es en realidad: grado, cardinalidad, etc.—comenta __Ada__.

¿Cómo interactúan entre sí las entidades? A través de las relaciones. La relación o interrelación es un elemento del modelo Entidad/Relación que permite relacionar datos entre sí. En una relación se asocia un elemento de una entidad con otro de otra entidad.

Relación: es una asociación entre diferentes entidades. En una relación no pueden aparecer dos veces relacionadas las mismas ocurrencias de entidad.

La representación gráfica en el modelo Entidad/Relación corresponde a un rombo en cuyo interior se encuentra inscrito el nombre de la relación. El rombo estará conectado con las entidades a las que relaciona, mediante líneas rectas, que podrán o no acabar en punta de flecha según el tipo de relación.

Cuando debas dar un nombre a una relación procura que éste haga referencia al objetivo o motivo de la asociación de entidades. Se suelen utilizar verbos en singular. Algunos ejemplos podrían ser: forman, poseen, atiende, contrata, hospeda, supervisa, imparte, etc.

En algunas ocasiones, es interesante que en las líneas que conectan las entidades con la relación, se indique el papel o rol que desempeña cada entidad en la relación. Como se verá más adelante, los papeles o roles son especialmente útiles en relaciones reflexivas.

Para describir y definir adecuadamente las relaciones existentes entre entidades, es imprescindible conocer los siguientes conceptos:

- ✓ **Grado** de la relación.
- ✓ **Cardinalidad de la relación.**
- ✓ **Cardinalidades de las entidades.**

A continuación desarrollamos cada uno de ellos.

5.1.- Grado de una relación.

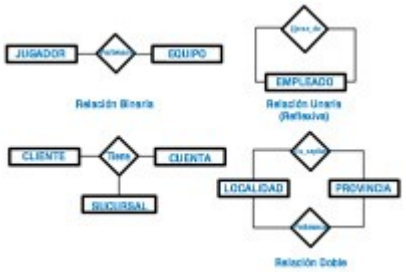
¿Pueden intervenir varias entidades en una misma relación? Claro que sí, en una relación puede intervenir una única entidad o varias.

Grado de una relación: número de entidades que participan en una relación.

En función del grado se pueden establecer diferentes tipos de relaciones:

- ✓ **Relación Unaria o de grado 1**: Es aquella relación en la que participa una única entidad. También llamadas reflexivas o recursivas.

- ✓ **Relación Binaria o de grado 2**: Es aquella relación en la que participan dos entidades. En general, tanto en una primera aproximación, como en los sucesivos refinamientos, el esquema conceptual de la base de datos buscará tener sólo este tipo de relaciones.
- ✓ **Relación Ternaria o de grado 3**: Es aquella relación en la que participan tres entidades al mismo tiempo.
- ✓ **Relación N-aria o de grado n**: Es aquella relación que involucra n entidades. Este tipo de relaciones no son usuales y deben ser simplificadas hacia relaciones de menor grado.
- ✓ **Relación doble**: ocurre cuando dos entidades están relacionadas a través de dos relaciones. Este tipo de relaciones son complejas de manejar.



En este gráfico puedes observar cada uno de los tipos de relaciones en función de su grado y su representación gráfica en el modelo Entidad/Relación.

Rellena los huecos con los conceptos adecuados.

En la relación unaria que puedes ver en el gráfico anterior, un empleado puede ejercer el rol de jefe o el rol de subordinado.

*Al ser una relación reflexiva que relaciona una entidad consigo misma, un empleado puede ejercer el rol de **jefe** sobre uno o varios empleados y, a su vez, podría ejercer el rol de **subordinado** bajo las ordenes de un jefe.*

5.2.- Cardinalidad de relaciones.

¿Qué es eso de la cardinalidad? En matemáticas, el cardinal de un conjunto es el número de elementos que lo forman. Este concepto puede extrapolarse a las relaciones con las que estamos tratando.

Cardinalidad de una relación: Es el número máximo de ocurrencias de cada entidad que pueden intervenir en una ocurrencia de relación. La cardinalidad vendrá expresada siempre para relaciones entre dos entidades. Dependiendo del número de ocurrencias de cada una de las entidades pueden existir relaciones **uno a uno, uno a muchos, muchos a uno y muchos a muchos**.

Observa el siguiente ejemplo, la cardinalidad indicará el número de ocurrencias de la entidad JUGADOR que se relacionan con cada ocurrencia de la entidad EQUIPO y viceversa. Podríamos hacer la siguiente lectura: un jugador pertenece a un equipo y a un equipo pueden pertenecer varios jugadores.

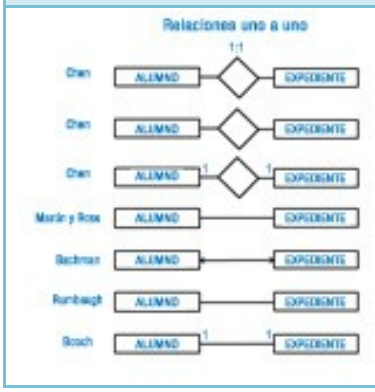
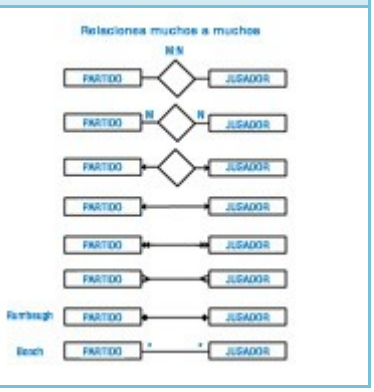
Una posible representación de la cardinalidad de las relaciones es la que hemos visto en el ejemplo anterior. Podríamos representar el resto de cardinalidades mediante las etiquetas **1:1**, **1:N**, **N:1**, **M:N** que se leerían respectivamente: uno a uno, uno a muchos, muchos a uno y muchos a muchos.
Veamos en detalle el significado de cada una de estas cardinalidades:

- ✓ **Relaciones uno a uno (1:1)**. Sean las entidades A y B, una instancia u ocurrencia de la entidad A se relaciona únicamente con otra instancia de la entidad B y viceversa. Por ejemplo, para cada ocurrencia de la entidad ALUMNO sólo habrá una ocurrencia relacionada de la entidad EXPEDIENTE y viceversa. O lo que es lo mismo, un alumno tiene un expediente asociado y un expediente sólo pertenece a un único alumno.
- ✓ **Relaciones uno a muchos (1:N)**. Sean las entidades A y B, una ocurrencia de la entidad A se relaciona con muchas ocurrencias de la entidad B y una ocurrencia de la entidad B sólo estará relacionada con una única ocurrencia de la entidad A. Por ejemplo, para cada ocurrencia de la entidad DOCENTE puede haber varias ocurrencias de la entidad ASIGNATURA y para varias ocurrencias de la entidad ASIGNATURA sólo habrá una ocurrencia relacionada de la entidad DOCENTE (si se establece que una asignatura sólo puede ser impartida por un único docente). O

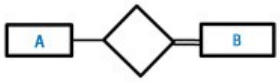
lo que es lo mismo, un docente puede impartir varias asignaturas y una asignatura sólo puede ser impartida por un único docente.

- ✓ **Relaciones muchos a uno (N:1)**. Sean las entidades A y B, una ocurrencia de la entidad A está asociada con una única ocurrencia de la entidad B y un ejemplar de la entidad B está relacionado con muchas ocurrencias de la entidad A. Por ejemplo, Un JUGADOR pertenece a un único EQUIPO y a un EQUIPO pueden pertenecer muchos jugadores.
- ✓ **Relaciones muchos a muchos (M:N)**. Sean las entidades A y B, un ejemplar de la entidad A está relacionado con muchas ocurrencias de la entidad B y viceversa. Por ejemplo, un alumno puede estar matriculado en varias asignaturas y en una asignatura pueden estar matriculados varios alumnos.

La cardinalidad de las relaciones puede representarse de varias maneras en los esquemas del modelo Entidad/Relación. A continuación, te ofrecemos un resumen de las notaciones clasificadas por autores, más empleadas en la representación de cardinalidad de relaciones.

Notaciones para representación de cardinalidad de relaciones.		
Relaciones uno a uno.	**Relaciones uno a muchos.**	**Relaciones muchos a muchos.**
		

5.3.- Cardinalidad de entidades.

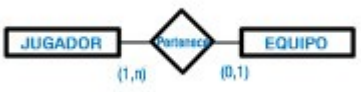


Si existe cardinalidad en las relaciones, supondrás que también existe para las entidades. Estás en lo cierto, la **cardinalidad** con la que una entidad participa en una relación especifica el número mínimo y el número máximo de correspondencias en las que puede tomar parte cada ejemplar de dicha entidad. Indica el número de relaciones en las que una entidad puede aparecer.

Sean las entidades A y B, la participación de la entidad A en una relación es **obligatoria (total)** si la existencia de cada una de sus ocurrencias necesita como mínimo de una ocurrencia de la entidad B (ver figura). En caso contrario, la participación es **opcional (parcial)**.

La cardinalidad de una entidad se representa con el número mínimo y máximo de correspondencias en las que puede tomar parte cada ejemplar de dicha entidad, entre paréntesis. Su representación gráfica será, por tanto, una etiqueta del tipo (0,1), (1,1), (0,N) o (1,N). El significado del primer y segundo elemento del paréntesis corresponde a **(cadinalidad mínima, cardinalidad máxima)**:

- ✓ **Cardinalidad mínima**. Indica el número mínimo de asociaciones en las que aparecerá cada ocurrencia 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). El valor 0 se pondrá cuando la participación de la entidad sea opcional.
- ✓ **Cardinalidad máxima**. Indica el número máximo de relaciones en las que puede aparecer cada ocurrencia de la entidad. Puede ser uno, otro valor concreto mayor que uno (tres por ejemplo) o muchos (se representa con n).



Veámoslo más claro a través del siguiente ejemplo: un JUGADOR pertenece como mínimo a ningún EQUIPO y como máximo a uno (0,1) y, por otra parte, a un EQUIPO pertenece como mínimo un JUGADOR y como máximo varios (1,n). Como puedes ver, la cardinalidad (0,1) de JUGADOR se ha colocado junto a la entidad EQUIPO para representar que un jugador puede no pertenecer a ningún equipo o como máximo a uno. Para la cardinalidad de EQUIPO ocurre igual, se coloca su cardinalidad junto a la entidad JUGADOR para expresar que en un equipo hay mínimo un jugador y máximo varios. Ten en cuenta que cuando se representa la cardinalidad de una entidad, el paréntesis y sus valores han de colocarse junto a la entidad con la que se relaciona. Es decir en el lado opuesto a la relación.
La cardinalidad de entidades también puede representarse en el modelo Entidad/Relación con la notación que se representa en la imagen de la derecha. Por tanto, el anterior ejemplo quedaría representado así:

Supongamos que seguimos diseñando una base de datos para un sitio de juegos online. En un punto del proceso de diseño se ha de modelar el siguiente requisito: cada usuario registrado podrá crear las partidas que desee (a las que otros usuarios pueden unirse), pero una partida solo podrá estar creada por un único usuario. Un usuario podrá o no crear partidas. ¿Cuáles serían las etiquetas del tipo (cardinalidad mínima, cardinalidad máxima) que deberían ponerse junto a las entidades USUARIO y PARTIDA respectivamente, si éstas están asociadas por la relación CREAR (partida)?

- ○ (1,N) y (0,N)
- ○ (1,1) y (1,N)
- ◉ **(1,1) y (0,N)**

Efectivamente, con estas cardinalidades estarías indicando que un usuario puede crear varias partidas, o ninguna. Por otra parte, una partida deberá estar creada exclusivamente por un único usuario.

6.- Simbología del modelo E/R.

Caso práctico

María *acaba de venir de comprar un tablón de anuncios de corcho. Va a colgarlo cerca su puesto de trabajo, junto al de Juan.*

–Mira ***Juan****, voy a imprimir estos gráficos en los que figuran los símbolos más utilizados a la hora de generar diagramas E/R. ¿Sabías que existen diferentes notaciones? – pregunta* ***María****.*

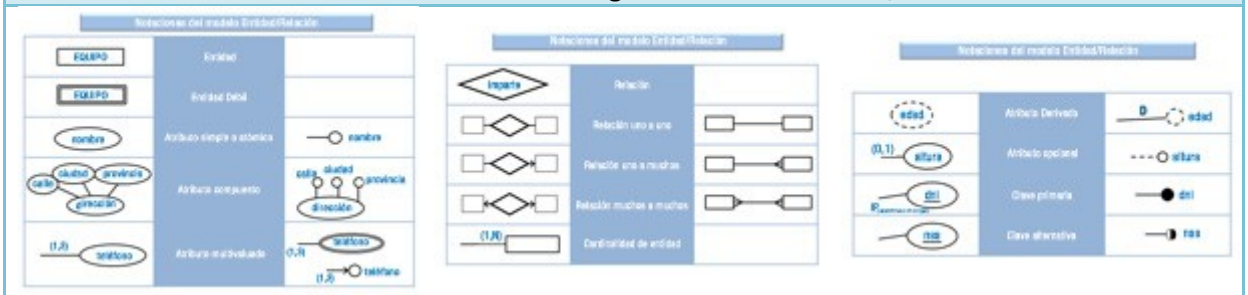
Juan*, que está buscando en su cajón la caja de las chinchetas, añade: –Me parece una idea genial y sí, sí que conocía la existencia de diferentes símbolos. Además, mientras buscaba en Internet algunos ejemplos, he visto que se pueden representar de diferentes maneras los mismos elementos.*

–Estupendo, así tendréis a mano la gran mayoría de símbolos y os será más cómodo interpretar los ejemplos que consultéis –comenta ***Ada****.*

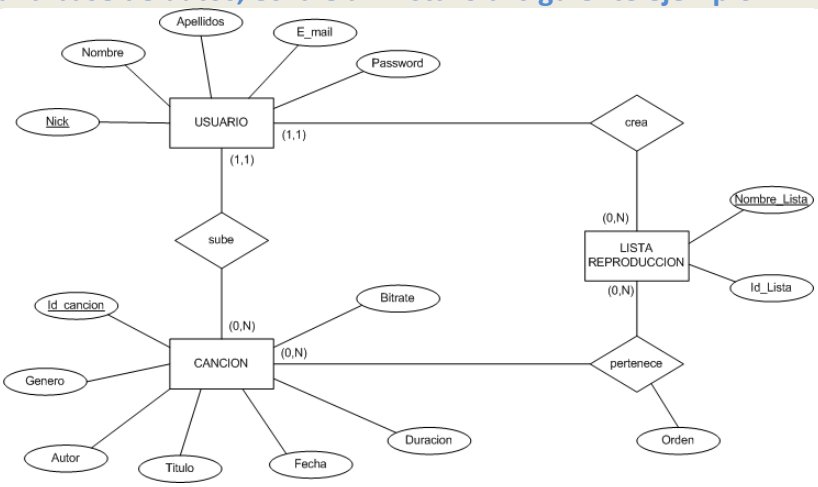
¿Recuerdas todos y cada uno de los símbolos que hemos utilizado a lo largo de esta unidad? Es probable que no. Para facilitar tu aprendizaje, te ofrecemos a continuación un resumen básico de los **símbolos utilizados en el modelo Entidad/Relación**. Verás que existen diferentes maneras de representar los mismos elementos, las que aquí se resumen te servirán para interpretar la gran mayoría de esquemas con los que te puedas encontrar.

Resumen básico de la simbología del modelo Entidad/Relación.

Resumen básico de la simbología del modelo Entidad/Relación



Si quieres ver un ejemplo de cómo se aplican algunas de estas notaciones en un esquema conceptual de una base de datos, échale un vistazo al siguiente ejemplo:



7.- El modelo E/R Extendido.

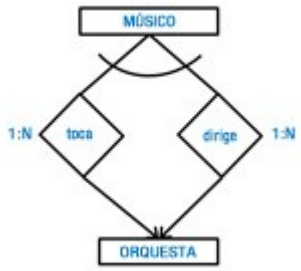
Caso práctico

Cuando la representación de determinadas entidades y relaciones se complique, los miembros de BK Programación necesitarán aplicar alguna técnica adicional que les permita realizar un modelado adecuado del problema. ***Ada****, está preparando una presentación en soporte informático con la que enseñará a* ***Juan*** *y* ***María*** *las nuevas posibilidades que les brinda el modelo Entidad-Relación Extendido.*

Hemos visto que a través del modelo Entidad/Relación se pueden modelar la gran mayoría de los requisitos que una base de datos debe cumplir. Pero existen algunos que ofrecen especial dificultad a la hora de representarlos a través de la simbología tradicional del modelo E/R. Para solucionar este problema, en el modelo Entidad/Relación Extendido se han incorporado nuevas extensiones que permiten mejorar la capacidad para representar circunstancias especiales. Estas extensiones intentan eliminar elementos de difícil o incompleta representación a través de la simbología existente, como por ejemplo relaciones con cardinalidad N:M, o la no identificación clara de entidades.

A continuación, se detallan estas nuevas características que convierten al modelo E/R tradicional en el **modelo Entidad/Relación Extendido**, como son: tipos de restricciones sobre las relaciones, especialización, generalización, conjuntos de entidades de nivel más alto y más bajo, herencia de atributos y agregación.

7.1.- Restricciones en las relaciones.



La primera extensión que el modelo Entidad/Relación Extendido incluye, se centra en la representación de una serie de restricciones sobre las relaciones y sus ejemplares, vamos a describirlas:

a. **Restricción de exclusividad.**
   Cuando existe **una entidad que participa en dos o más relaciones** y cada ocurrencia de dicha entidad sólo puede pertenecer a una de las relaciones únicamente, decimos que existe una restricción de exclusividad. Si la ocurrencia de entidad pertenece a una de las relaciones, no podrá formar parte de la otra. **O se produce una relación o se produce otra pero nunca ambas a la vez.**
   Por ejemplo, supongamos que un músico puede dirigir una orquesta o tocar en en ella, pero no puede hacer las dos cosas simultáneamente. Existirán por tanto, dos relaciones dirige y toca, entre las entidades MUSICO y ORQUESTA, estableciéndose una relación de exclusividad entre ellas.
   La representación gráfica en el modelo Entidad/Relación Extendido de una restricción de exclusividad se realiza **mediante un arco** que engloba a todas aquellas relaciones que son exclusivas.
b. **Restricción de exclusión.**
   Este tipo de restricción se produce **cuando las ocurrencias de las entidades sólo pueden asociarse utilizando una única relación**.
   Pongamos un ejemplo, supongamos que un monitor puede impartir diferentes cursos de perfeccionamiento para monitores, y que éste puede a su vez recibirlos. Pero si un monitor imparte un determinado curso, no podrá estar recibiéndolo simultáneamente y viceversa. Se establecerá, por tanto, una restricción de exclusión que se representa mediante una **línea discontinua entre las dos relaciones**, tal y como se muestra en el ejemplo.
c. **Restricción de inclusividad.**
   Este tipo de restricciones se aplican cuando es necesario modelar **situaciones en las que para que dos ocurrencias de entidad se asocien a través de una relación, tengan que haberlo estado antes a través de otra relación**.

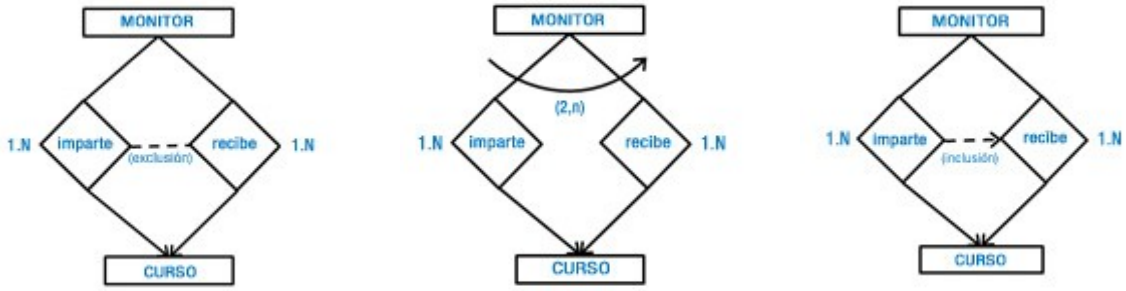
Siguiendo con el ejemplo anterior, supongamos que para que un monitor pueda impartir cursos de cocina sea necesario que reciba previamente dos cursos: nutrición y primeros auxilios. Como puedes ver, es posible que los cursos que el monitor deba recibir no tengan que ser los mismos que luego pueda impartir. Aplicando una restricción de inclusividad entre las relaciones imparte y recibe, estaremos indicando que cualquier ocurrencia de la entidad MONITOR que participa en una de las relaciones (imparte) tiene que participar obligatoriamente en la otra (recibe).

Se representará mediante un **arco acabado en flecha**, que partirá desde la relación que ha de cumplirse primero hacia la otra relación. Se indicará junto al arco la cardinalidad mínima y máxima de dicha restricción de inclusividad. En el ejemplo, (2,n) indica que un monitor ha de recibir 2 cursos antes de poder impartir varios.

d. **Restricción de inclusión.**

En algunas ocasiones aplicar una restricción de inclusividad no representa totalmente la realidad a modelar, entonces se hace necesario aplicar una restricción de inclusión que es aún más fuerte.

En nuestro ejemplo, si hemos de modelar que un monitor pueda impartir un curso, si previamente lo ha recibido, entonces tendremos que aplicar una restricción de inclusión. Con ella toda ocurrencia de la entidad MONITOR que esté asociada a una ocurrencia determinada de la entidad CURSO, a través de la relación imparte, ha de estar unida a la misma ocurrencia de la entidad CURSO a través de la relación recibe.



Supongamos que hemos de modelar mediante el modelo Entidad/Relación Extendido el siguiente requerimiento de una base de datos: Para que un hombre se divorcie de una mujer, primero ha de haber estado casado con ella.
Las entidades participantes son MUJER y HOMBRE, que estarán asociadas a través de dos relaciones: se casa, se divorcia. No tendremos en cuenta la cardinalidad de ambas relaciones.
¿Qué tipo de restricción sobre las relaciones hemos de establecer en nuestro esquema para representar correctamente este requisito?

- ○ Restricción de exclusividad
- ○ Restricción de inclusividad
- ◉ **Restricción de inclusión**

Efectivamente, este tipo de restricción establece la obligatoriedad de haber un casamiento para que pueda haber un divorcio y, además, las entidades que se relacionan a través de la relación se casa, deben ser las mismas que las participantes en se divorcia.

7.2.- Generalización y especialización.

La segunda extensión incorporada en el modelo Entidad/Relación Extendido se centra en nuevos tipos de relaciones que van a permitir modelar la realidad de una manera más fiel. Estos nuevos

tipos de relación reciben el nombre de **jerarquías** y se basan en los conceptos de generalización, especialización y herencia.

Cuando estamos diseñando una base de datos puede que nos encontremos con conjuntos de entidades que posean características comunes, lo que permitiría crear un tipo de entidad de nivel más alto que englobase dichas características. Y a su vez, puede que necesitemos dividir un conjunto de entidades en diferentes subgrupos de entidades por tener éstas, características diferenciadoras. Este proceso de refinamiento ascendente/descendente, permite expresar mediante la generalización la existencia de tipos de entidades de nivel superior que engloban a conjuntos de entidades de nivel inferior. A los conjuntos de entidades de nivel superior también se les denomina **superclase o supertipo**. A los conjuntos de entidades de nivel inferior se les denomina **subclase o subtipo**.

Por tanto, existirá la posibilidad de realizar una **especialización de una superclase en subclases**, y análogamente, **establecer una generalización de las subclases en superclases**. La generalización es la reunión en una superclase o supertivo de entidad de una serie de subclases o subtipos de entidades, que poseen características comunes. Las subclases tendrán otras características que las diferenciarán entre ellas.

¿Cómo detectamos una generalización? Podremos identificar una generalización cuando encontremos una serie de atributos comunes a un conjunto de entidades, y otros atributos que sean específicos. Los atributos comunes conforman la superclase o supertipo y los atributos específicos la subclase o subtipo.

Las jerarquías se caracterizan por un concepto que hemos de tener en cuenta, **la herencia**. A través de la herencia los atributos de una superclase de entidad son heredados por las subclases. Si una superclase interviene en una relación, las subclases también lo harán.

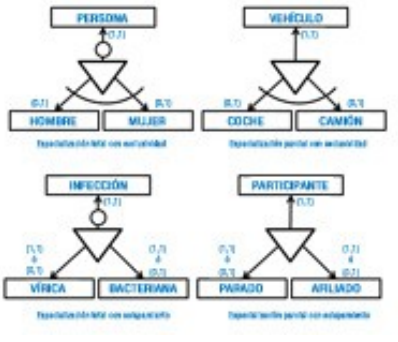
¿Cómo se representa una generalización o especialización? Existen varias notaciones, pero hemos de convenir que la relación que se establece entre una superclase de entidad y todos sus subtipos se expresa a través de las palabras ES UN, o en notación inglesa IS A, que correspondería con ES UN TIPO DE. Partiendo de este punto, una jerarquía se representa mediante un triángulo invertido, sobre él quedará la entidad superclase y conectadas a él a través de líneas rectas, las subclases.

En el ejemplo de la imagen, las subclases INVITADO, REGISTRADO y ADMINISTRADOR constituyen subclases de la superclase USUARIO. Cada una de ellas aporta sus propias características y heredan las pertenecientes a su superclase.

Una generalización/especialización podrá tener las siguientes restricciones semánticas:

- ✓ **Totalidad**: una generalización/especialización será total si todo ejemplar de la superclase pertenece a alguna de las subclases.
- ✓ **Parcialidad**: una generalización/especialización será parcial si no todos los ejemplares de la superclase pertenecen a alguna de las subclases.
- ✓ **Solapamiento**: una generalización/especialización presentará solapamiento si un mismo ejemplar de la superclase puede pertenecer a más de una subclase.
- ✓ **Exclusividad**: una generalización/especialización presentará exclusividad si un mismo ejemplar de la superclase pertenece sólo a una subclase.

Las diferentes restricciones semánticas descritas tienen su representación gráfica, a través del gráfico que a



continuación te mostramos podrás entender mejor su funcionamiento.

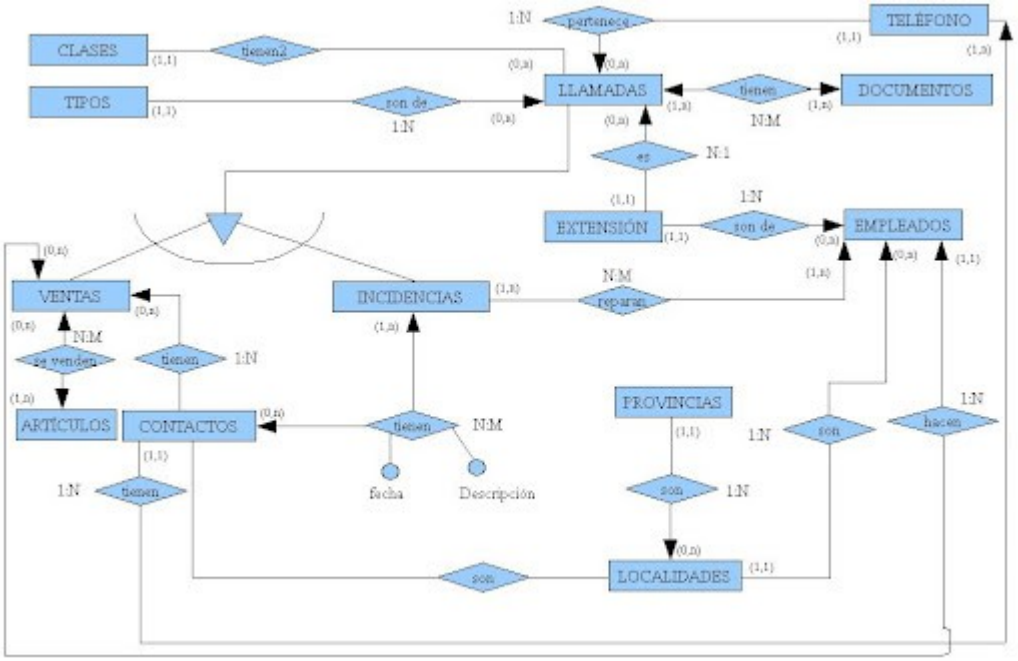
Ejercicio resuelto

Supongamos la existencia de dos entidades TURISMO y CAMION. Los atributos de la entidad TURISMO son: Num_bastidor, Fecha_fab, precio y Num_puertas. Los atributos de la entidad CAMION son: Num_bastidor, Fecha_fab, precio, Num_ejes y Tonelaje.

Si analizamos ambas entidades existen algunos atributos comunes y otros que no. Por tanto, podremos establecer una jerarquía. Para ello, reuniremos los atributos comunes y los asociaremos a una nueva entidad superclase denominada VEHICULO. Las subclases TURISMO y CAMI0N, con sus atributos específicos, quedarán asociadas a la superclase VEHICULO mediante una jerarquía parcial con solapamiento. En el siguiente gráfico puedes apreciar la transformación.

Respuesta:

Si quieres ver cómo se integra la representación de jerarquías dentro de un esquema conceptual completo, te proponemos observes el siguiente esquema:



7.3.- Agregación.

Abordamos ahora la tercera de las extensiones del modelo Entidad/Relación Extendido, la **Agregación**. En el modelo Entidad/Relación no es posible representar relaciones entre relaciones. La agregación es una abstracción a través de la cual las relaciones se tratan como entidades de nivel más alto, siendo utilizada para expresar relaciones entre relaciones o entre entidades y relaciones.

Supongamos un ejemplo en el que hemos de modelar la siguiente situación: una empresa de selección de personal realiza entrevistas a diferentes aspirantes. Puede ser que, de algunas de estas entrevistas a aspirantes, se derive una oferta de empleo, o no. En el siguiente gráfico se representan tres soluciones, las dos primeras erróneas y una tercera correcta, utilizando una agregación.
Como has podido observar, la representación gráfica de una agregación se caracteriza por englobar con un rectángulo las entidades y relación a abstraer. De este modo, se crea una nueva entidad agregada que puede participar en otras relaciones con otras entidades. En este tipo de relación especial de agregación, la cardinalidad máxima y mínima de la entidad agregada siempre será (1,1) no indicándose por ello en el esquema.
Existen dos clases de agregaciones:

- ✓ **Compuesto/componente**: Un todo se obtiene por la unión de diversas partes, que pueden ser objetos distintos y que desempeñan papeles distintos en la agregación. Teniendo esto en cuenta, esta abstracción permite representar que un todo o agregado se obtiene por la unión de diversas partes o componentes que pueden ser tipos de entidades distintas y que juegan diferentes roles en la agregación.
- ✓ **Miembro/Colección**: Un todo se obtiene por la unión de diversas partes del mismo tipo y que desempeñan el mismo papel en la agregación. Teniendo esto en cuenta, esta abstracción permite representar un todo o agregado como una colección de miembros, todos de un mismo tipo de entidad y todos jugando el mismo rol. Esta agregación puede incluir una restricción de orden de los miembros dentro de la colección (indicando el atributo de ordenación). Es decir, permite establecer un orden entre las partes.

En la siguiente figura puedes apreciar los tipos de agregación y su representación gráfica.

Con la agregación hemos terminado de detallar las extensiones más importantes del modelo Entidad/Relación Extendido. A lo largo de tu andadura por el mundo de las bases de datos y, en concreto, en todo lo relacionado con los esquemas conceptuales y diagramas Entidad/Relación, es probable que te encuentres con diferentes notaciones y simbologías. Algunas ya las hemos representado a lo largo de esta unidad y otras podrás encontrarlas en el enlaces que te ofrecemos a continuación. Además, puedes utilizar la información que te proponemos para reforzar y ampliar todo lo visto.
http://www.lsi.us.es/docencia/get.php?id=4564 (páginas 1 a 9). (0.33 MB)

Si hemos de representar a través del modelo E/R Extendido los alumnos pertenecientes a una clase, podríamos utilizar una agregación del tipo Compuesto/Componente.

Verdadero ○ **Falso** ◉

Al ser el alumnado un conjunto de elementos que representan el mismo rol en la relación, el tipo de agregación debería ser Miembro/Colección.

8.- Elaboración de diagramas E/R.

Caso práctico

–La verdad es que a través del esquema que estamos generando, queda más claro cómo es cada entidad y cada relación. Aplicando estas técnicas creo que vamos a ir afianzando un método fiable que podremos aplicar en futuros desarrollos – afirma ***Juan****.*

Ada*, está echando un vistazo a lo que llevan hecho* ***Juan*** *y* ***María****. – Efectivamente* ***Juan****, hay que ser metódicos y no descartar ningún paso, pues podríamos provocar errores en nuestros desarrollos. La confianza de nuestros clientes es vital y para ello hemos de obtener un producto con la mayor calidad posible.*

María *añade: –Supongo que según vayamos realizado proyectos parecidos mejoraremos nuestra técnica.*

Llegados a este punto, te surgirán varias dudas ¿Cómo creo un diagrama E/R? ¿Por dónde empiezo? ¿Y qué puedo hacer con todo lo visto? Son cuestiones totalmente normales cuando se comienza, no te preocupes, vamos a darte una serie de orientaciones para que puedas aplicar todos los conceptos aprendidos hasta ahora en la elaboración de diagramas Entidad/Relación.

Sabemos que en la fase de diseño conceptual de la base de datos, en la que nos encontramos, hemos de generar el diagrama E/R que representará de manera más sencilla el problema real a modelar, independientemente del Sistema Gestor de Base de Datos. Este esquema será como un plano que facilite la comprensión y solución del problema. Este diagrama estará compuesto por la representación gráfica, a través de la simbología vista, de los requisitos o condiciones que se derivan del problema a modelar.

Saltarnos este paso en el proceso de creación e implementación de una base de datos, supondría pérdida de información. Por lo que esta fase, requerirá de la creación de uno o varios esquemas previos más cercanos al mundo real, antes del paso a tablas del modelo relacional.

Te darás cuenta que, como en la programación, **la práctica es fundamental**. Los diagramas no siempre se crean del mismo modo y, en ocasiones, hay que retocarlos e incluso rehacerlos. A través de la resolución de diferentes problemas y la elaboración de múltiples diagramas, obtendrás la destreza necesaria para generar esquemas que garanticen una posterior y correcta conversión del modelo Entidad/Relación al modelo Relacional.

8.1.- Identificación de entidades y relaciones.

¡Manos a la obra! Lo primero que hemos de tener a nuestra disposición para poder generar un diagrama E/R adecuado es el conjunto de requerimientos, requisitos o condiciones que nuestra base de datos ha de cumplir. Es lo que se denomina el documento de especificación de requerimientos. En otras palabras, el enunciado del problema a modelar. Cuanto más completa y detallada sea la información de la que dispongamos, mucho mejor.

Suponiendo que conocemos la simbología del modelo Entidad/Relación y que entendemos su significado ¿Cómo empezamos? Las etapas para la creación del diagrama E/R se detallan a continuación:

a. **Identificación de entidades**: Es un proceso bastante intuitivo. Para localizar aquellos elementos que serán las entidades de nuestro esquema, analizaremos la especificación de requerimientos en busca de nombres o sustantivos. Si estos nombres se refieren a objetos importantes dentro del problema probablemente serán entidades. Tendremos en cuenta que nombres referidos a características, cualidades o propiedades no se convertirán en entidades.

Otra forma de identificar entidades es localizando objetos o elementos que existen por sí mismos. Por ejemplo: VEHICULO, PIEZA, etc. En otras ocasiones, la localización de varias características o propiedades puede dejar ver la existencia de una entidad.

¿Esto puede ser una entidad o no? Es una pregunta que se repite mucho cuando estamos en esta etapa. Algunos autores indican que para poder considerarse como entidad se deben cumplir tres reglas:

- ✓ Existencia propia.
- ✓ Cada ejemplar de un tipo de entidad debe poder ser diferenciado del resto de ejemplares.
- ✓ Todos los ejemplares de un tipo de entidad deben tener las mismas propiedades.

El número de entidades obtenidas debe ser manejable y según se vayan identificando se les otorgará **nombres, preferiblemente en mayúsculas, representativos** de su significado o función. De esta manera el diagrama será cada vez más legible.

b. **Identificación de relaciones**: Localizadas las entidades, debemos establecer qué relación existe entre ellas. Para ello, analizaremos de nuevo el documento de especificación de requerimientos en busca de **verbos o expresiones verbales que conecten unas entidades con otras**. En la gran mayoría de ocasiones encontraremos que las relaciones se establecen entre dos entidades (relaciones binarias), pero prestaremos especial atención a las relaciones entre más entidades y a las relaciones recursivas o relaciones unarias.

c. Cada una de las relaciones establecidas deberá tener asignado un **nombre, preferiblemente en minúsculas, representativo** de su significado o acción.

En ocasiones, el identificador de una relación está compuesto por varias palabras, como por ejemplo: es supervisado, trabaja para, etc. Es recomendable que utilices guiones bajos para unir las palabras que forman el identificador.

Dependiendo de la notación elegida, el siguiente paso será la representación de la cardinalidad (mínima y máxima) de las entidades participantes en cada relación y del tipo de correspondencia de la relación (1 a 1, 1 a muchos o muchos a muchos).

Si hemos encontrado alguna relación recursiva, reflexiva o unaria, hemos de representar en nuestro esquema los roles desempeñados por la entidad en dicha relación.

Rellena los huecos con los conceptos adecuados.

Las entidades suelen localizarse en el documento de especificación de requerimientos a través de sustantivos y las relaciones a través de verbos. Pero hemos de tener cuidado, no siempre los sustantivos representarán entidades, pues podría tratarse de atributos.

8.2.- Identificación de atributos, claves y jerarquías.

Sólo con la localización de entidades y relaciones no está todo hecho. Hemos de completar el proceso realizando las siguientes tareas:

a. **Identificación de atributos**: Volvemos sobre el documento de especificación de requerimientos para buscar nombres relativos a características, propiedades, identificadores o cualidades de entidades o relaciones. Resulta más sencillo si nos preguntamos ¿Qué información es necesario tener en cuenta de una u otra entidad o relación? Quizás no todos los atributos estén reflejados directamente en el documento de especificación de

requerimientos, aplicando el sentido común el diseñador podrá establecerlos en algunos casos y en otros, será necesario consultar e indagar en el problema.

Tendremos en cuenta si los atributos localizados son simples o compuestos, derivados o calculados y si algún atributo o conjunto de ellos se repite en varias entidades. Si se da este último caso, deberemos detenernos y plantear la posibilidad de establecer una jerarquía de especialización, o bien, dejar las entidades tal y como han sido identificadas.

Cada atributo deberá tener asignado un nombre, preferiblemente en minúsculas, representativo de su contenido o función. Además, siempre es recomendable recopilar la siguiente información de cada atributo:

- ✓ Nombre y descripción.
- ✓ Atributos simples que lo componen, si es atributo compuesto.
- ✓ Método de cálculo, si es atributo derivado o calculado.

En el caso de encontrar atributos asociados a relaciones con cardinalidad uno a muchos, se valorará asignar ese atributo o atributos a la entidad con mayor cardinalidad participante en la relación.

b. **Identificación de claves**: Del conjunto de atributos de una entidad se establecerán una o varias claves candidatas, escogiéndose una de ellas como clave o llave primaria de la entidad. Esta clave estará formada por uno o varios atributos que identificarán de manera unívoca cada ocurrencia de entidad. El proceso de identificación de claves permitirá determinar la fortaleza (al menos una clave candidata) o debilidad (ninguna clave candidata) de las entidades encontradas.

   Se representará la existencia de esta clave primaria mediante la notación elegida para la elaboración el diagrama E/R. Del mismo modo, se deberán representar adecuadamente las entidades fuertes o débiles.

c. **Determinación de jerarquías**: Como se ha comentado anteriormente, es probable que existan entidades con características comunes que puedan ser generalizadas en una entidad de nivel superior o superclase (jerarquía de generalización). Pero también, puede ser necesario expresar en el esquema las particularidades de diferentes ejemplares de un tipo de entidad, por lo que se crearán subclases o subtipos de una superclase o supertipo (jerarquía de especialización). Para ello, habrá que analizar con detenimiento el documento de especificación de requerimientos.

   Si se identifica algún tipo de jerarquía, se deberá representar adecuadamente según el tipo de notación elegida, determinando si la jerarquía es total/parcial o exclusiva/con solapamiento.

8.3.- Metodologías.

Hasta aquí, tenemos identificados los elementos necesarios para construir nuestro diagrama, pero ¿Existe alguna metodología para llevarlo a cabo? Sí, y además podremos utilizar varias. Partiremos de una versión preliminar del esquema conceptual o diagrama E/R que, tras sucesivos refinamientos, será modificado para obtener el diagrama E/R definitivo. Las metodologías o estrategias disponibles para la elaboración del esquema conceptual son las siguientes:

a. **Metodología Descendente (Top-Down)**: Se trata de partir de un esquema general e ir descomponiendo éste en niveles, cada uno de ellos con mayor número de detalles. Se parte de objetos muy abstractos, que se refinan paso a paso hasta llegar al esquema final.

b. **Metodología Ascendente (Bottom-Up)**: Inicialmente, se parte del nivel más bajo, los atributos. Se irán agrupando en entidades, para después ir creando las relaciones entre éstas y las posibles jerarquías hasta obtener un diagrama completo. Se parte de objetos atómicos que no pueden ser descompuestos y a continuación se obtienen abstracciones u objetos de mayor nivel de abstracción que forman el esquema.
c. **Metodología Dentro-fuera (Inside-Out)**: Inicialmente se comienza a desarrollar el esquema en una parte del papel y a medida que se analiza la especificación de requerimientos, se va completando con entidades y relaciones hasta ocupar todo el documento.
d. **Metodología Mixta**: Es empleada en problemas complejos. Se dividen los requerimientos en subconjuntos que serán analizados independientemente. Se crea un esquema que servirá como estructura en la que irán interconectando los conceptos importantes con el resultado del análisis de los subconjuntos creados. Esta metodología utiliza las técnicas ascendente y descendente. Se aplicará la técnica descendente para dividir los requerimientos y en cada subconjunto de ellos, se aplicará la técnica ascendente.

¿Cuál de estas metodologías utilizar? Cualquiera de ellas puede ser válida, todo dependerá de lo fácil y útil que te resulte aplicarlas. Probablemente y, casi sin ser consciente de ello, tú mismo crearás tu propia metodología combinando las existentes. Pero, como decíamos hace algunos epígrafes, **la práctica es fundamental**. Realizando gran cantidad de esquemas, analizándolos y llevando a cabo modificaciones en ellos es como irás refinando tu técnica de elaboración de diagramas E/R. Llegará un momento en que sólo con leer el documento de especificación de requerimientos serás capaz de ir construyendo en tu mente cómo será su representación sobre el papel, pero paciencia y ve paso a paso.

"Vísteme despacio, que tengo prisa". *Napoleón Bonaparte.*

Rellena los huecos con los conceptos adecuados.
La metodología en la que se parte de un alto nivel de abstracción y que, tras un proceso de refinamiento sucesivo, se obtiene el esquema final se denomina: Metodología descendente.

8.4.- Redundancia en diagramas E/R.

Una de las principales razones por las que las bases de datos aparecieron fue la eliminación de la redundancia en los datos ¿Y qué es la redundancia?

Redundancia: reproducción, reiteración, insistencia, reincidencia, reanudación. En bases de datos hace referencia al almacenamiento de los mismos datos varias veces en diferentes lugares.

La redundancia de datos puede provocar problemas como:

- ✓ **Aumento de la carga de trabajo**: al estar almacenado un dato en varios lugares, las operaciones de grabación o actualización de datos necesitan realizarse en varias ocasiones.
- ✓ **Gasto extra de espacio de almacenamiento**: al estar repetidos, los datos ocupan mayor cantidad de espacio en el medio de almacenamiento. Cuanto mayor sea la base de datos, más patente se hará esta problema.
- ✓ **Inconsistencia**: se produce cuando los datos que están repetidos, no contienen los mismos valores. Es decir, se ha actualizado su valor en un lugar y en otro no, por lo que no se sabría qué dato es válido y cual erróneo.

Para que una base de datos funcione óptimamente, hay que empezar realizando un buen diseño de ella. Es imprescindible que nuestros diagramas E/R controlen la redundancia y, para ello, debemos analizar el esquema y valorar qué elementos pueden estar incorporando redundancia a nuestra solución.

¿Dónde buscamos indicios de redundancia en nuestros esquemas? Existen lugares y elementos que podrían presentar redundancia, por ejemplo:

- ✓ **Atributos redundantes** cuyo contenido se calcula en función de otros. Un atributo derivado puede ser origen de redundancia.
- ✓ Varias entidades unidas circularmente o cíclica a través de varias relaciones, es lo que se conoce como **un ciclo**. En caso de existir un ciclo, deberemos tener en cuenta las siguientes condiciones, antes de poder eliminar dicha relación redundante:
  - ➔ Que el significado de las relaciones que componen el ciclo sea el mismo.
  - ➔ Que si eliminamos la relación redundante, el significado del resto de relaciones es el mismo.
  - ➔ Que si la relación eliminada tenía atributos asociados, éstos puedan ser asignados a alguna entidad participante en el esquema, sin que se pierda su significado.

Pero hay que tener en cuenta que **no siempre que exista un ciclo estaremos ante una redundancia**. Es necesario analizar detenidamente dicho ciclo para determinar si realmente existe o no redundancia.

Para finalizar, una apreciación. No toda redundancia es perjudicial. Existen ciertas circunstancias y condiciones en las que es conveniente (sobre todo a efectos de rendimiento) introducir cierta **redundancia controlada** en una base de datos. Por ejemplo, si el método de cálculo del valor de un determinado atributo derivado es complejo (varias operaciones matemáticas o de cadenas de caracteres, varios atributos implicados, etc.) y ralentiza el funcionamiento de la base de datos, quizá sea conveniente definir dicho atributo desde el principio y no considerarlo como un atributo redundante. La incorporación o no de redundancia controlada dependerá de la elección que haga el diseñador.

8.5.- Propiedades deseables de un diagrama E/R.

Cuando construimos un diagrama Entidad/Relación existen una serie de propiedades o características que éste debería cumplir. Quizá no se materialicen todas, pero hemos de intentar cubrir la gran mayoría de ellas. De este modo, conseguiremos que nuestros diagramas o esquemas conceptuales tengan mayor calidad.

Estas características o propiedades deseables se desglosan a continuación:

- ✓ **Completitud**: Un diagrama E/R será completo si es posible verificar que cada uno de los requerimientos está representado en dicho diagrama y viceversa, cada representación del diagrama tiene su equivalente en los requerimientos.
- ✓ **Corrección**: Un diagrama E/R será correcto si emplea de manera adecuada todos los elementos del modelo Entidad/Relación. La corrección de un diagrama puede analizarse desde dos vertientes:
  - ➔ **Corrección sintáctica**: Se producirá cuando no se produzcan representaciones erróneas en el diagrama.
  - ➔ **Corrección semántica**: Se producirá cuando las representaciones signifiquen exactamente lo que está estipulado en los requerimientos. Posibles errores semánticos serían: la utilización de un atributo en lugar de una entidad, el uso de una entidad en lugar de una relación, utilizar el mismo identificador para dos entidades o dos relaciones, indicar erróneamente alguna cardinalidad u omitirla, etc.
- ✓ **Minimalidad**: Un diagrama E/R será mínimo si se puede verificar que al eliminar algún concepto presente en el diagrama, se pierde información. Si un diagrama es redundante, no será mínimo.
- ✓ **Sencillez**: Un diagrama E/R será sencillo si representa los requerimientos de manera fácil de comprender, sin artificios complejos.

- ✓ **Legibilidad**: Un diagrama E/R será legible si puede interpretarse fácilmente. La legibilidad de un diagrama dependerá en gran medida del modo en que se disponen los diferentes elementos e interconexiones. Esta propiedad tiene mucho que ver con aspectos estéticos del diagrama.
- ✓ **Escalabilidad**: Un diagrama E/R será escalable si es capaz de incorporar posibles cambios derivados de nuevos requerimientos.

Si en un diagrama E/R asociamos un atributo a una entidad, pero este atributo debe asociarse realmente a una relación en la que interviene dicha entidad, estaríamos incumpliendo la propiedad de:

- ○ Completitud
- ◉ **Corrección semántica**
- ○ Corrección sintáctica

9.- Paso del diagrama E/R al modelo relacional.

Caso práctico

__Juan__ y __María__ ya han terminado de elaborar el diagrama E/R, con la ayuda de __Ada__. Las últimas modificaciones hechas en éste garantizan que todas las condiciones establecidas en el documento de especificación de requerimientos han sido representadas adecuadamente.

–¿Y ahora cómo se pasa este diagrama a una base de datos real? –pregunta __María__.

–Aún hay que obtener el "paso a tablas" de lo representado en el diagrama. En cuanto realicemos esa transformación tendremos los elementos necesarios para implementar nuestra base de datos en cualquier SGBD relacional –le aclara __Ada__.

Si analizamos todo el proceso descrito hasta el momento, la fase de diseño conceptual desarrollada, y que se materializa en el diagrama E/R, permite una gran independencia de las cuestiones relativas a la implementación física de la base de datos. El tipo de SGBD, las herramientas software, las aplicaciones, lenguajes de programación o hardware disponible no afectarán, al menos hasta el momento, a los resultados de esta fase.

Nuestro esquema conceptual habrá sido revisado, modificado y probado para verificar que se cumplen adecuadamente todos y cada uno de los requerimientos del problema a modelar. Este esquema representará el punto de partida para la siguiente fase, el diseño lógico de la base de datos.

El diseño lógico consistirá en la construcción de un esquema de la información relativa al problema, basado en un modelo de base de datos concreto. El esquema conceptual se transformará en un esquema lógico que utilizará los elementos y características del modelo de datos en el que esté basado el SGBD, para implementar nuestra base de datos. Como pudimos ver anteriormente, estos modelos podrán ser: el modelo en red, el modelo jerárquico y, sobre todo, el modelo relacional y el modelo orientado a objetos.

Para esta transformación será necesario realizar una serie de pasos preparatorios sobre el esquema conceptual obtenido en la fase de diseño conceptual. Nos centraremos en la simplificación y transformación del esquema para que el paso hacia el modelo de datos elegido (en este caso el modelo relacional) sea mucho más sencilla y efectiva.

Seguidamente, tomando como referencia el esquema modificado/simplificado, se realizará el paso de éste al modelo de datos relacional. Esta transformación requerirá de la aplicación de determinadas reglas y condiciones que garanticen la equivalencia entre el esquema conceptual y el esquema lógico.

Como paso posterior, sobre la información del esquema lógico obtenido, será necesario llevar a cabo un proceso que permitirá diseñar de forma correcta la estructura lógica de los datos. Este proceso recibe el nombre de normalización, que se conforma como un conjunto de técnicas que permiten validar esquemas lógicos basados en el modelo relacional.

Entonces, ¿qué pasos son los siguientes a dar? Resumiendo un poco, simplificaremos nuestro diagrama E/R, lo transformaremos al modelo relacional, aplicaremos normalización y obtendremos lo que se conoce en el argot como el paso a tablas del esquema conceptual o, lo que es lo mismo, el esquema lógico. Desde ese momento, basándonos en este esquema, podremos llevarnos nuestra base de datos a cualquier SGBD basado en el modelo relacional e implementarla físicamente. Esta implementación física será totalmente dependiente de las características del SGBD elegido.

9.1.- Simplificación previa de diagramas.

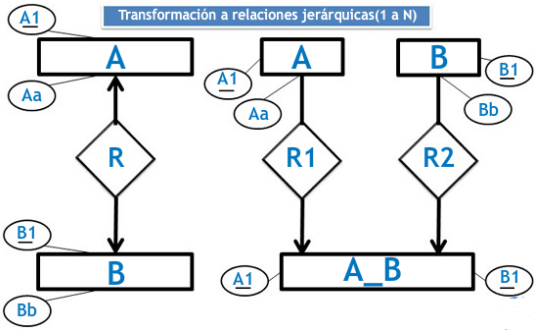
Existe un conjunto de procedimientos y normas que es necesario aplicar a nuestros diagramas E/R para que su transformación al modelo lógico basado en el modelo relacional, sea correcta y casi automática. Si aplicas correctamente estas pautas, conseguirás que el proceso de transformación sea fácil y fiable. Las transformaciones de las que estamos hablando son las siguientes:

- ✓ Transformación de relaciones n-arias en binarias.
- ✓ Eliminación de relaciones cíclicas.
- ✓ Reducción a relaciones jerárquicas (uno a muchos).
- ✓ Conversión de entidades débiles en fuertes.

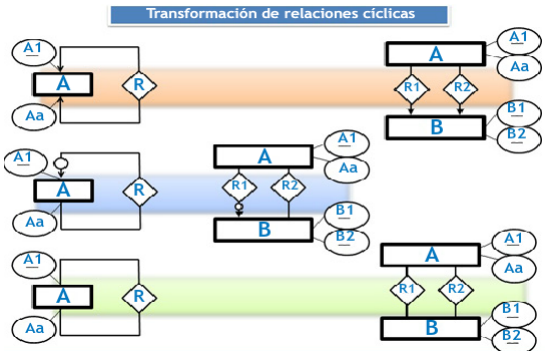
Veamos detalladamente cómo llevar a cabo las transformaciones de las que hemos estado hablando:

- ✓ **Transformación de atributos compuestos:** Los atributos compuestos de una entidad han de ser descompuestos en los atributos simples por los que están formados.
- ✓ **Transformación de atributos multivaluados:** Si nuestro diagrama incluye la existencia de un atributo multivaluado, este se ha de convertir en una entidad relacionada con la entidad de la que procede. Para esta nueva entidad se elegirá un nombre adecuado y tendrá un único atributo (el correspondiente al antiguo atributo múltiple). Este atributo es posible que funcione correctamente como clave primaria de la entidad pero a veces es posible que no. En este caso, la entidad que hemos creado puede que sea débil. Deberemos ajustar en cualquier caso correctamente las claves primarias.
- ✓ **Transformación a relaciones jerárquicas:** Se trata de transformar las relaciones con cardinalidad muchos a muchos (M a N) en relaciones con cardinalidad uno a muchos (1 a N). Observa la animación para comprender cómo se realiza la transformación. Si existiese algún atributo asociado a la relación n-aria, quedaría asociado a la nueva entidad que se crea.

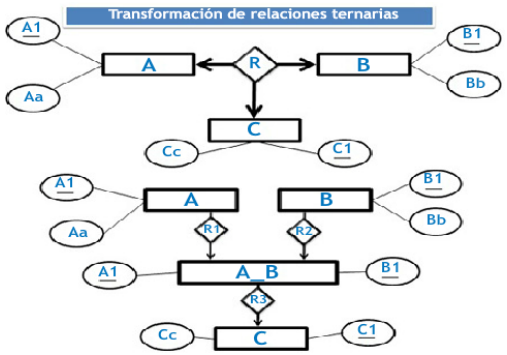
La relación R, que asocia la entidad A con la entidad B, tiene cardinalidad M a N (muchos a muchos). Para eliminarla, se crea una nueva entidad A_B cuya clave estará formada por las claves de A y B. Posteriormente se relacionan A y B mediante dos relaciones 1 a N (R1 y R2) con la nueva entidad A_B.



- ✓ **Transformación de relaciones cíclicas:** De forma general, si tenemos una entidad sobre la que existe una relación cíclica, para eliminar dicha relación, se crea una nueva entidad cuya clave estará formada por dos atributos, que contendrán las claves de las ocurrencias relacionadas. Entre ambas entidades se establecen dos relaciones, cuya cardinalidad dependerá de la cardinalidad que tuviera la relación cíclica en un principio.



- ✓ **Transformación de relaciones ternarias:** El tratamiento de las relaciones ternarias es similar al realizado para atributos asociados a relaciones, ya que una relación ternaria puede considerarse como una relación binaria a a la que se le asocia una entidad. Por consiguiente, si en lugar de ser un conjunto de atributos los asociados a la relación es una entidad, se asociaría ésta mediante una nueva



relación a la entidad resultante de eliminar la relación binaria.

- ✓ **Transformación de entidades débiles en fuertes:** Para esta transformación sólo es necesario añadir a la entidad débil los atributos clave de la entidad que hace posible la identificación de las ocurrencias. La clave de esta nueva entidad fuerte estará formada por los atributos clave de la que fuera entidad débil más los atributos adicionales.

Sea la entidad ALUMNADO que participa en la relación COLABORA con otra entidad llamada GRUPO_TRABAJO. Un alumno o alumna puede colaborar en varios grupos de trabajo simultáneamente y, a su vez, en un grupo de trabajo pueden colaborar un número indeterminado de alumnos. Se necesita registrar los días en los que el alumnado colabora con cada grupo de trabajo, para ello se asocia a la relación COLABORA un atributo denominado fecha_colaboración. Este atributo registrará en qué fecha un determinado alumno/a ha colaborado en un determinado grupo de trabajo.
¿Si tuvieras que hacer la transformación de esta parte del esquema conceptual para eliminar la relación M a N COLABORA, dónde colocarías el atributo fecha_colaboración?

- ○ En la entidad ALUMNADO, ya que en esta entidad es donde se almacenan todos los datos asociados al alumnado. Si consultamos el alumno o alumna, sabremos cuándo a colaborado en un grupo
- ◉ **En una nueva entidad que es combinación de ALUMNADO y GRUPO_TRABAJO, a la que podríamos llamar ALUMNADO_GRUPO**
- ○ En la entidad GRUPO_TRABAJO

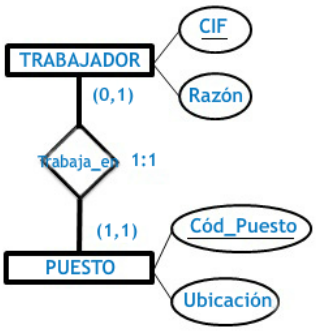
al transformar la relación M a N, se crean dos relaciones 1 a N entre ALUMNADO-ALUMNADO_GRUPO Y GRUPO_TRABAJO-ALUMNADO_GRUPO, siendo ALUMNADO_GRUPO una nueva entidad que tendrá por claves las claves primarias de ALUMNADO y GRUPO_TRABAJO, recibiendo como atributo el atributo que estaba asociado a la relación COLABORA. Para cada par ALUMNADO/GRUPO_TRABAJO tendremos registrado cuándo se realizó la colaboración.

10.- Paso del diagrama E/R al Modelo Relacional.

Si se ha llevado a cabo el proceso preparatorio de nuestro esquema conceptual o diagrama E/R, según se ha indicado en epígrafes anteriores, dispondremos de un Esquema Conceptual Modificado (ECM) en el que sólo existirán exclusivamente entidades fuertes con sus atributos y relaciones jerárquicas (1 a N). Pues bien, la aplicación del modelo de datos Relacional es automática, para ello se deben tener en cuenta las siguientes cuestiones:

- ✓ Toda entidad se transforma en una tabla.
- ✓ Todo atributo se transforma en columna dentro de una tabla.
- ✓ El atributo clave de la entidad se convierte en clave primaria de la tabla y se representará subrayado en la tabla.
- ✓ Cada entidad débil generará una tabla que incluirá todos sus atributos, añadiéndose a ésta los atributos que son clave primaria de la entidad fuerte con la que esté relacionada. Estos atributos añadidos se constituyen como clave foránea que referencia a la entidad fuerte. Seguidamente, se escogerá una clave primaria para la tabla creada.
- ✓ Las relaciones Uno a Uno podrán generar una nueva tabla o propagar la clave en función de la cardinalidad de las entidades.

Paso de relaciones 1 a 1 del Esquema E/R al Esquema Relacional



Se ha de tener en cuenta las cadinalidades de las entidades que intervienen. Existen dos soluciones:

Si las entidades poseen cardinalidades (0,1), la relación se convertirá en una tabla.
Si una de las entidades posee cardinalidad (0,1) y la otra (1,1), se propagará la clave de la entidad con cardinalidad (1,1) a la tabla resultante de la entidad de cardinalidad (0,1).
Si ambas entidades tienen cadinalidad (1,1) se puede propagar la clave de cualquiera de ellas a la tabla de la otra entidad.

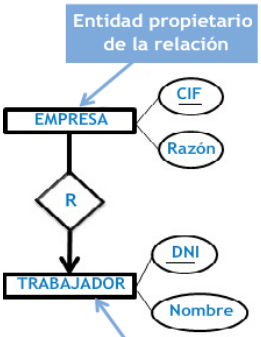
Nuestro ejemplo quedaría:

PUESTO(Cód_Puesto, Ubicación, DNI)
TRABAJADOR(DNI, Nombre)

DNI quedaría como clave foránea que referencia a la tabla TRABAJADOR

- ✓ Las relaciones Uno a Muchos podrán generar una nueva tabla o propagar la clave.

Paso de relaciones 1 a N del Esquema E/R al Esquema Relacional



Como norma general:

a) Se convierten a tablas las dos entidades
b) Se propaga la clave de la entidad propietario a la entidad miembro

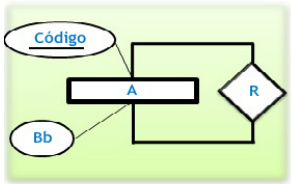
El paso a tablas quedaría:

EMPRESA(CIF, Razón)
TRABAJADOR(DNI,Nombre,CIF)

CIF quedaría como clave foránea que referencia a la tabla EMPRESA

- ✓ Las relaciones reflexivas o cíclicas podrán generar una o varias tablas en función de la cardinalidad de la relación.

Paso de relaciones Reflexivas del Esquema E/R al Esquema Relacional



Para este tipo de relaciones hemos de tener muy en cuenta la cardinalidad. Podremos tener los siguientes casos:

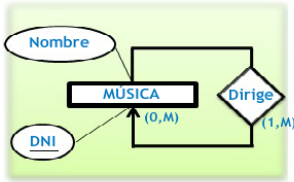
a) Si la relación es de 1 a 1, la clave de la entidad se repetirá, aunque debe ser con identificadores diferentes. Un identificador actuará como clave primaria y el otro, como clave foránea de ella misma

La clave primaria Código pasa a tener dos identificadores diferentes. Uno de ellos actuará como clave primaria y el otro como foránea

Para nuestro ejemplo, el paso a tablas quedaría:

A(Código1, Código2, Bb)

Paso de relaciones Reflexivas del Esquema E/R al Esquema Relacional



b) Si la relación es 1 a M, podremos tener dos casos:

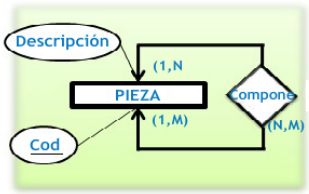
- Si la entidad muchos es siempre obligatoria, actuaremos como para el caso 1 a 1
- Si la entidad muchos no es obligatoria, se creará una nueva tabla cuya clave será la de la entidad del lado muchos, y se propaga la clave hacia esta nueva tabla como clave foránea

Para nuestro ejemplo, el paso a tablas quedaría:

MÚSICO(DNI, Nombre)
DIRIGE(DNI, DNI_Director)

En la entidad DIRIGE, se añade de nuevo el DNI pero jugando el rol de director, siendo clave foránea de la entidad MÚSICO

Paso de relaciones Reflexivas del Esquema E/R al Esquema Relacional



c) Si la relación es M a N, se trataría como una relación binaria entre dos entidades. Se crearía una nueva tabla que tendrá dos veces la clave primaria de la entidad. Si hubiese atributos asociados a la relación, se asociarían a la nueva tabla. La clave de esta tabla será la combinación de ambas claves primarias.

Para nuestro ejemplo, el paso a tablas quedaría:

PIEZA(Cod, Descripción)
PIEZA_COMPUESTA(Cod,Cod_com ponente)

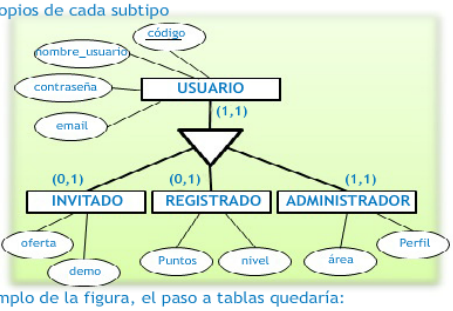
En la nueva tabla PIEZA_COMPUESTA aparecerá dos veces la clave primaria de PIEZA, pero jugando dos roles. Por un lado el de clave de la pieza en cuestión y, por otro, el de la/s pieza/s que la componen

✓ Las jerarquías generarán la reunión, eliminación o creación de relaciones 1 a 1.

Paso de Jerarquías del Esquema E/R al Esquema Relacional

Existen tres formas de tratar las relaciones jerárquicas, son las siguientes:

c) Crear una única entidad que aglutine todos los subtipos. Esta nueva entidad tendrá todos los atributos del supertipo y de los subtipos. Esta unión permite una mayor simplicidad, aunque puede provocar valores nulos en atributos propios de cada subtipo

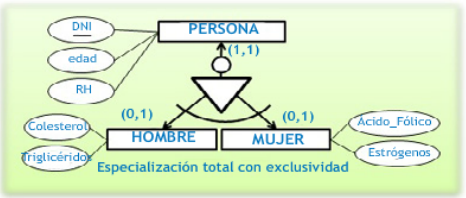


Para el ejemplo de la figura, el paso a tablas quedaría:

USUARIO(Código, nombre_usuario, contraseña, email, oferta, demo, Puntos, nivel, área, Perfil)

Paso de Jerarquías del Esquema E/R al Esquema Relacional

b) Anulación del supertipo. Al suprimir el supertipo, sus atributos pasan directamente a todos los subtipos y las relaciones del supertipo se han de reproducir en cada uno de los subtipos. La clave del supertipo, pasa a los subtipos. Este tratamiento suele aplicarse en jerarquías totales y exclusivas

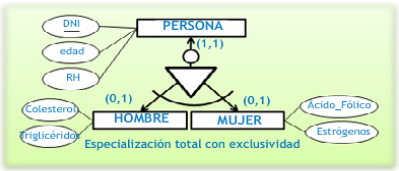


Para el ejemplo de la figura, el paso a tablas quedaría:

HOMBRE(DNI, edad, RH, Colesterol, Triglicéridos)
MUJER (DNI, edad, RH, Ácido_Fólico, Estrógenos)

Paso de Jerarquías del Esquema E/R al Esquema Relacional

c) Añadir relaciones 1 a 1 entre el supertipo y los subtipos. Los atributos del supertipo se mantendrán y cada uno de los subtipos tendrá una clave foránea proveniente del supertipo, con la que podrán identificarse. El supertipo se relaciona con los subtipos mediante relaciones 1 a 1



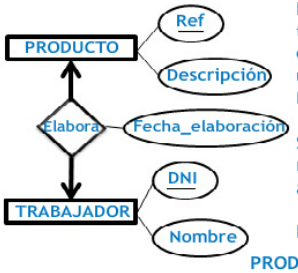
Para el ejemplo de la figura, el paso a tablas quedaría:

PERSONA (DNI, edad, RH)
HOMBRE(DNI, Colesterol, Triglicéridos)
MUJER (DNI, Ácido_Fólico, Estrógenos)

✓ Las relaciones Muchos a Muchos se transforman en una tabla que tendrá como clave primaria las claves primarias de las entidades que asocia.

No obstante, si en el proceso de generación del diagrama E/R o esquema conceptual hemos aplicado correctamente las reglas de simplificación de diagramas, nuestro Esquema Conceptual Modificado nos permitirá el paso a tablas teniendo en cuenta sólo las transformaciones asociadas a entidades, relaciones 1 a N, 1 a 1 y Jerarquías.

Paso de relaciones M a N del Esquema E/R al Esquema Relacional



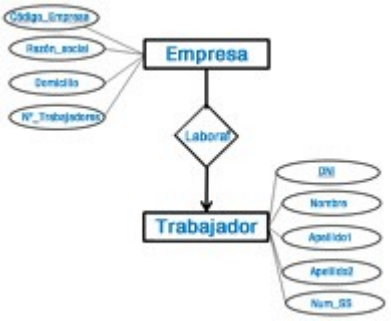
Las relaciones M a N se transforman en una nueva tabla que tendrá como clave primaria la unión de las claves primarias de las entidades que asocia

Si hubiese atributos asociados a la relación, éstos se incluirían como atributos de la nueva tabla

Nuestro ejemplo quedaría:

PRODUCTO(Ref, Descripción)
TRABAJADOR(DNI, Nombre)
ELABORA(Ref, DNI,Fecha_elaboración)

Las claves de las entidades relacionadas, quedan en la nueva tabla



Ejercicio resuelto

Sea la siguiente representación a través del modelo E/R de una relación entre dos entidades, obtén el paso a tablas de dicho esquema:

Respuesta:

El paso a tablas de dicho esquema sería el siguiente:

```
EMPRESA (Código empresa, razón_social, domicilio, N°_Trabajadores)
TRABAJADOR(DNI, Nombre, Apellido1, Apellido2, Num_SS)
```

Para materializar la relación de uno a muchos LABORAL, se incluye una clave foránea en la entidad TRABAJADOR, que referencia a la entidad EMPRESA, quedando:

```
EMPRESA (Código empresa, razón_social, domicilio, N°_Trabajadores)
TRABAJADOR(DNI, Nombre, Apellido1, Apellido2, Num_SS, Código_empresa)
```

11.- Normalización de modelos relacionales.

Caso práctico

*En estos primeros desarrollos **Ada** debe estar muy pendiente del trabajo que están realizando **Juan** y **María.** El proceso de transformación del Esquema Conceptual Modificado al modelo Relacional, requiere cierta experiencia y concentración. Dada su importancia y dificultad, este paso deben llevarlo a cabo de manera tranquila y comentando en grupo las diferentes operaciones que van a ir realizando.*

¿Crees que tu base de datos ya podría construirse directamente sobre el SGBD relacional que hayas elegido? La respuesta podría ser afirmativa, pero si queremos que nuestra base de datos funcione con plena fiabilidad, es necesario antes llevar a cabo un proceso de normalización de las tablas que la componen.

¿Y qué es eso de la normalización?

Normalización: Proceso que consiste en imponer a las tablas del modelo Relacional una serie de restricciones a través de un conjunto de transformaciones consecutivas. Este proceso garantizará que las tablas contienen los atributos necesarios y suficientes para describir la realidad de la entidad que representan, permitiendo separar aquellos atributos que por su contenido podrían generar la creación de otra tabla.

A veces uno se pregunta ¿Quién habrá sido el ideante de estos conceptos? En el siguiente enlace que te proponemos, puedes conocer quién fue.

http://es.wikipedia.org/wiki/Edgar_Frank_Codd

A principios de la década de los setenta, concretamente en 1972, Codd establece una técnica para llevar a cabo el diseño de la estructura lógica de los datos representados a través del modelo relacional, a la que denominó **normalización**. Pero esta técnica no ha de utilizarse para el diseño de la base de datos, sino como un proceso de refinamiento que debe aplicarse después de lo que conocemos como "paso a tablas", o lo que formalmente se denomina traducción del esquema conceptual al esquema lógico. Este proceso de refinamiento conseguirá los siguientes objetivos:

- ✓ Suprimir dependencias erróneas entre atributos.
- ✓ Optimizar los procesos de inserción, modificación y borrado en la base de datos.

El proceso de normalización se basa en el análisis de las dependencias entre atributos. Para ello tendrá en cuenta los conceptos de: **dependencia funcional, dependencia funcional completa y dependencia transitiva**. Estos conceptos se desarrollan seguidamente.

¿Y cómo se aplica la normalización? Es un proceso que se realiza en varias etapas secuenciales. Cada etapa está asociada a una **forma normal**, que establece unos requisitos a cumplir por la tabla sobre la que se aplica. Existen varias formas normales: **Primera, Segunda, Tercera, Boyce-Codd, Cuarta, Quinta y Dominio-Clave**. Como hemos indicado, el paso de una forma normal a otra es consecutivo, si no se satisface una determinada forma normal no puede pasarse al análisis de la siguiente. Según vamos avanzando en la normalización, los requisitos a cumplir serán cada vez más restrictivos, lo que hará que nuestro esquema relacional sea cada vez más robusto.

Como norma general, para garantizar que no existan problemas en la actualización de datos, **es recomendable aplicar el proceso de normalización hasta Tercera Forma Normal o incluso hasta Forma Normal de Boyce-Codd**. En los siguientes epígrafes se describen las características y requisitos de cada una de las formas normales.

11.1.- Tipos de dependencias.

Vamos a desarrollar aquí los conceptos sobre los que se basa el análisis de dependencias entre atributos, que se lleva a cabo en el proceso de normalización antes indicado, son los siguientes:

- ✓ **Dependencia Funcional:** Dados los atributos A y B, se dice que B depende funcionalmente de A, sí, y solo sí, para cada valor de A sólo puede existir un valor de B. La dependencia funcional siempre se establece entre atributos de una misma tabla. El atributo A se denomina determinante, ya que A determina el valor de B. Para representar esta dependencia funcional utilizamos la siguiente notación: **A → B**. Hay que indicar que A y B podrían ser un solo atributo o un conjunto de ellos.
- ✓ **Dependencia Funcional Completa:** Dados los atributos A1, A2, ...Ak y B, se dice que B depende funcionalmente de forma completa de A1, A2, ...Ak, si y solo si B depende funcionalmente del conjunto de atributos A1, A2, ...Ak, pero no de ninguno de sus posibles subconjuntos.
- ✓ **Dependencia Transitiva:** Dados tres atributos A, B y C, se dice que existe una dependencia transitiva entre A y C, si B depende funcionalmente de A y C depende funcionalmente de B. A, B y C podrían ser un solo atributo o un conjunto de ellos.

Para ilustrar los tipos de dependencias descritas, analiza el siguiente ejercicio resuelto.

Ejercicio resuelto

Dadas las siguientes tablas:

```
EMPLEADO( DNI, Nombre, Dirección, Localidad, Cod_Localidad, Nombre_hijo, Edad_hijo)
LIBRO (Título_libro, Num_ejemplar, Autor, Editorial, Precio)
```

Resuelve las siguientes cuestiones:

a. Indica qué atributos presentan una dependencia funcional de la clave primaria de la tabla EMPLEADO.
b. Indica qué atributos presentan una dependencia funcional completa en la tabla LIBRO.
c. Indica qué atributos presentan una dependencia transitiva en la tabla EMPLEADO.

Resultado:

Apartado a)

Los atributos Nombre, y Dirección dependen funcionalmente de DNI, ya que para un DNI específico sólo podrá haber un nombre y una dirección. Pero los atributos Nombre_hijo y Edad_hijo no presentan esa dependencia funcional de DNI, ya que para un DNI específico podríamos tener varios valores diferentes en esos atributos. (Consideraremos para este ejemplo que todos los empleados registrados en esta base de datos tienen nombres distintos). Expresemos estas dependencias funcionales mediante su notación:

```
DNI → Nombre
DNI → Dirección
```

Apartado b)

Los atributos Editorial y Precio dependen funcionalmente del conjunto de atributos que forman la clave primaria de la tabla, pero no dependen de Título_libro o de Num_ejemplar por separado, por lo que presentan una dependencia funcional completa de la clave. El atributo Autor depende funcionalmente sólo y exclusivamente de Titulo_libro, por lo que no presenta una dependencia funcional completa de los atributos que forman la clave.

Apartado c)

Los atributos Cod_Localidad y Localidad dependen funcionalmente de DNI, pero entre Cod_Localidad y Localidad existe otra dependencia funcional. Por tanto, se establece que Localidad depende funcionalmente de Cod_Localidad, y a su vez, Cod_Localidad depende funcionalmente de DNI. Con lo que podemos afirmar que existe una dependencia transitiva entre Localidad y DNI. Si lo representamos con la notación asociada a las dependencias funcionales, quedaría:

```
DNI → Cod_Localidad → Localidad.
```

11.2.- Formas Normales.

Una vez conocidos los conceptos sobre los que se basa el proceso de normalización, se han de llevar a cabo una serie de etapas consecutivas en las que se aplicarán las propiedades de cada una de las formas normales definidas por Codd. A continuación se exponen los requisitos a cumplir por las tablas de nuestra base de datos según la forma normal que apliquemos.

1ª Forma Normal

Una tabla está en Primera Forma Normal (1FN o FN1) sí, y sólo sí, todos los atributos de la misma contienen valores atómicos, es decir, no hay grupos repetitivos. Dicho de otra forma, estará en 1FN si los atributos no clave, dependen funcionalmente de la clave. ¿Cómo se normaliza a Primera Forma Normal?

a. Se crea, a partir de la tabla inicial, una nueva tabla cuyos atributos son los que presentan dependencia funcional de la clave primaria. La clave de esta tabla será la misma clave primaria de la tabla inicial. Esta tabla ya estará en 1FN.
b. Con los atributos restantes se crea otra tabla y se elije entre ellos uno que será la clave primaria de dicha tabla. Comprobaremos si esta segunda tabla está en 1FN. Si es así, la tabla inicial ya estará normalizada a 1FN y el proceso termina. Si no está en 1FN, tomaremos la segunda tabla como tabla inicial y repetiremos el proceso.

2ª Forma Normal

Una tabla está en Segunda Forma Normal (2FN o FN2) sí, y sólo sí, está en 1FN y, además, todos los atributos que no pertenecen a la clave dependen funcionalmente de forma completa de ella. Es obvio que una tabla que esté en 1FN y cuya clave esté compuesta por un único atributo, estará en 2FN. ¿Cómo se normaliza a Segunda Forma Normal?

a. Se crea, a partir de la tabla inicial, una nueva tabla con los atributos que dependen funcionalmente de forma completa de la clave. La clave de esta tabla será la misma clave primaria de la tabla inicial. Esta tabla ya estará en 2FN.
b. Con los atributos restantes, se crea otra tabla que tendrá por clave el subconjunto de atributos de la clave inicial de los que dependen de forma completa. Se comprueba si esta tabla está en 2FN. Si es así, la tabla inicial ya está normalizada y el proceso termina. Si no está en 2FN, tomamos esta segunda tabla como tabla inicial y repetiremos el proceso.

3ª Forma Normal

Una tabla está en Tercera Forma Normal (3FN o FN3) sí, y sólo sí, está en 2FN y, además, cada atributo que no está en la clave primaria no depende transitivamente de la clave primaria. ¿Cómo se normaliza a Tercera Forma Normal?

a. Se crea, a partir de la tabla inicial, una nueva tabla con los atributos que no poseen dependencias transitivas de la clave primaria. Esta tabla ya estará en 3FN.
b. Con los atributos restantes, se crea otra tabla con los dos atributos no clave que intervienen en la dependencia transitiva, y se elije uno de ellos como clave primaria, si cumple los requisitos para ello. Se comprueba si esta tabla está en 3FN. Si es así, la tabla inicial ya está

normalizada y el proceso termina. Si no está en 3FN, tomamos esta segunda tabla como tabla inicial y repetiremos el proceso.

Forma Normal de Boyce Codd

Una tabla está en Forma Normal de Boyce-Codd (FNBC o BCFN) sí, y sólo sí, está en 3FN y todo determinante es una clave candidata. Un determinante será todo atributo simple o compuesto del que depende funcionalmente de forma completa algún otro atributo de la tabla. Aquellas tablas en la que todos sus atributos forman parte de la clave primaria, estarán en FNBC. Por tanto, si encontramos un determinante que no es clave candidata, la tabla no estará en FNBC. Esta redundancia suele ocurrir por una mala elección de la clave. Para normalizar a FNBC tendremos que descomponer la tabla inicial en dos, siendo cuidadosos para evitar la pérdida de información en dicha descomposición.

Otras formas normales

Existen también la Cuarta Forma Normal (4FN o FN4), Quinta Forma Normal (5FN o FN5) y Forma Normal de Dominio-Clave (DKFN), aunque se ha recomendado normalizar hasta 3FN o FNBC. La 4FN se basa en el concepto de Dependencias Multivaluadas, la 5FN en las Dependencias de Join o de reunión y la DKFN en las restricciones impuestas sobre los dominios y las claves.

Si deseas conocer cuáles son las propiedades y requisitos a cumplir establecidos en las formas normales 4ª, 5ª y DKFN, te proponemos los siguientes enlaces:
http://conclase.net/mysql/curso/?cap=004c#NOR_4FN
http://es.wikipedia.org/wiki/5NF
http://es.wikipedia.org/wiki/DKNF

Ejercicio resuelto

Sea la siguiente tabla:

```
COMPRAS (cod_compra, cod_prod, nomb_prod, fecha, cantidad, precio, fecha_rec, cod_prov, nomb_prov, tfno).
```

Se pide normalizarla hasta FNBC.

Resultado:

Comprobamos 1FN:

La tabla COMPRAS está en 1FN ya que todos sus atributos son atómicos y todos los atributos no clave dependen funcionalmente de la clave.

Comprobamos 2FN:

Nos preguntaremos ¿Todo atributo depende de todo el conjunto de atributos que forman la clave primaria, o sólo de parte?. Como vemos, existen atributos que dependen sólo de una parte de la clave, por lo que esta tabla no está en 2FN.

Veamos las dependencias:

```
cod_prod → nomb_prod, y cod_prod es parte de la clave primaria.
```

Al no estar en 2FN, hemos de descomponer la tabla COMPRAS en:

```
COMPRA1 (cod_compra, cod_prod, fecha, cantidad, precio, fecha_rec, cod_prov, nomb_prov, tfno).
```

```
PRODUCTO (cod_prod, nomb_prod).
```

Una vez hecha esta descomposición, ambas tablas están en 2FN. Todos los atributos no clave dependen de toda la clave primaria.

Comprobamos 3FN:
PRODUCTO está en 3FN, ya que por el número de atributos que tiene no puede tener dependencias transitivas.
¿COMPRA1 está en 3FN? Hemos de preguntarnos si existen dependencias transitivas entre atributos no clave.

Veamos las dependencias:

```
cod_prov → nomb_prov
cod_prov → tfno
```

(siendo cod_prov el código del proveedor y nomb_prov el nombre del proveedor)

COMPRA1 no está en 3FN porque existen dependencias transitivas entre atributos no clave, por tanto hemos de descomponer:

```
COMPRA2 (cod_compra, cod_prod, fecha, cantidad, precio, fecha_rec, cod_prov)
PROVEEDOR (cod_prov, nomb_prov, tfno)
```

Comprobamos FNBC:

PRODUCTO está en FNBC, ya que está en 3FN y todo determinante es clave candidata.
COMPRA2 está en FNBC, ya que está en 3FN y todo determinante es clave candidata.
PROVEEDOR está en FNBC, ya que está en 3FN y todo determinante es clave candidata.

La tabla inicial COMPRAS queda normalizada hasta FNBC del siguiente modo:

```
PRODUCTO (cod_prod, nomb_prod)
COMPRA2 (cod_compra, cod_prod, fecha, cantidad, precio, fecha_rec, cod_prov)
PROVEEDOR (cod_prov, nomb_prov, tfno)
```

TEMA 3

INDICE

1. Análisis y diseño de bases de datos. 3
2.- ¿Qué es el Modelo E/R? 4
3.- Entidades. 5
3.1.- Tipos: fuertes y débiles. 5
4.- Atributos. 7
4.1.- Tipos de atributos 7
4.2.- Claves. 8
4.3.- Atributos de una relación. 9
5.- Relaciones. 11
5.1.- Grado de una relación. 11
5.2.- Cardinalidad de relaciones. 12
5.3.- Cardinalidad de entidades. 13
6.- Simbología del modelo E/R. 15
7.- El modelo E/R Extendido. 16
7.1.- Restricciones en las relaciones. 16
7.2.- Generalización y especialización. 17
Ejercicio resuelto 19
7.3.- Agregación. 19
8.- Elaboración de diagramas E/R. 21
8.1.- Identificación de entidades y relaciones. 21
8.2.- Identificación de atributos, claves y jerarquías. 22
8.3.- Metodologías. 23
8.4.- Redundancia en diagramas E/R. 24
8.5.- Propiedades deseables de un diagrama E/R. 25
9.- Paso del diagrama E/R al modelo relacional. 27
9.1.- Simplificación previa de diagramas. 28
10.- Paso del diagrama E/R al Modelo Relacional. 30
Ejercicio resuelto 32
11.- Normalización de modelos relacionales. 33
11.1.- Tipos de dependencias. 34
Ejercicio resuelto 34
11.2.- Formas Normales. 35
1ª Forma Normal 35
2ª Forma Normal 35
3ª Forma Normal 35
Forma Normal de Boyce Codd 36
Otras formas normales 36
Ejercicio resuelto 36

Interpretación de diagramas entidad/relación.

Caso práctico

__Ada__ está analizando la manera en la que __Juan__ y __María__ han comenzando a construir la base de datos que sustentará el sitio web de juegos online. Parece que la aplicación del modelo relacional está marchando correctamente, aunque le interesa que el proceso se realice siguiendo un método lo más estandarizado posible y que les ofrezca independencia del SGBD que escojan.

De este modo, podrán planificar el desarrollo de cada una de las fases y ajustar mejor los tiempos dedicados a cada una de ellas.

Como se ha descrito en unidades anteriores, un modelo de datos es una colección de herramientas conceptuales que permiten llevar a cabo la descripción de los datos, sus relaciones, su semántica o significado y las restricciones que se les pueden aplicar. Sabemos que los SGBD cuentan con una arquitectura que simplifica, a los diferentes usuarios de la base de datos, su labor. El objetivo fundamental de esta arquitectura es separar los programas de aplicación de la base de datos física, proponiendo tres niveles de abstracción: **nivel interno o físico, nivel lógico o conceptual y nivel externo o de visión del usuario**.

1. Análisis y diseño de bases de datos.

El **Nivel lógico o conceptual** describe la estructura completa de la base de datos a través de lo que llamamos **Esquema Conceptual,** que se encarga de representar la información de una manera totalmente independiente del Sistema Gestor de Base de Datos.

Cuando hemos de desarrollar una base de datos se distinguen claramente dos fases de trabajo: **Análisis** y **Diseño**. En la siguiente tabla te describimos las etapas que forman parte de cada fase.

Pasos de las fases de Análisis y de Dsiseño	
Fase de Análisis	**Fase de Diseño**
Análisis de entidades: Se trata de localizar y definir las entidades y sus atributos.	Diseño de tablas.
Análisis de relaciones: Se definirán las relaciones existentes entre entidades.	Normalización.
Obtención del Esquema Conceptual a través del modelo E-R.	Aplicación de retrodiseño, si fuese necesario.
Fusión de vistas: Se reúnen en un único esquema todos los esquemas existentes en función de las diferentes vistas de cada perfil de usuario.	Diseño de transacciones: localización del conjunto de operaciones o transacciones que operarán sobre el esquema conceptual.
Aplicación del enfoque de datos relacional.	Diseño de sendas de acceso: se formalizan los métodos de acceso dentro de la estructura de datos.

Llevando a cabo una correcta fase de análisis estaremos dando un paso determinante en el desarrollo de nuestras bases de datos. El hecho de saltarse el esquema conceptual conlleva un problema de pérdida de información respecto al problema real a solucionar. El esquema conceptual debe reflejar todos los aspectos relevantes del mundo real que se va a modelar.

Para la realización de esquemas que ofrezcan una visión global de los datos, Peter Chen en 1976 y 1977 presenta dos artículos en los que se describe el **modelo Entidad/Relación** (entity/relationship). Con el paso del tiempo, este modelo ha sufrido modificaciones y mejoras. Actualmente, el modelo **entidad/relación extendido (ERE)** es el más aceptado, aunque existen variaciones que hacen que este modelo no sea totalmente un estándar. Ambos modelos serán estudiados a lo largo de esta unidad.

2.- ¿Qué es el Modelo E/R?

Caso práctico

*—**María**, ¿Si un carpintero recibe un encargo de un mueble, en qué crees que se basa para fabricarlo? —Pregunta **Ada**.*

*Levantando la vista de la pantalla de su ordenador, **María** contesta: —Supongo que un esquema o croquis, a veces hay detalles que es necesario consultar en la documentación, porque todo no es posible memorizarlo.*

*Juan interviene: —Me temo que ya sé por dónde van los tiros, **Ada**. ¿Con esa pregunta te estás refiriendo a los esquemas gráficos que se deben crear para la construcción de bases de datos?*
***Ada** sonríe y hace un gesto para que ambos se acerquen: —¿Sabéis lo qué es el modelo Entidad – Relación?*

Es una herramienta de referencia para la representación conceptual de problemas del mundo real. Su objetivo principal, facilitar el diseño de bases de datos permitiendo la especificación de un esquema que representa la estructura lógica completa de una base de datos. Este esquema partirá de las descripciones textuales de la realidad, que establecen los requerimientos del sistema, buscando ser lo más fiel posible al comportamiento del mundo real para modelarlo.

El modelo de datos E-R representa el significado de los datos, es un modelo semántico. De ahí que no esté orientado a ningún sistema físico concreto y tampoco tiene un ámbito informático puro de aplicación, ya que podría utilizarse para describir procesos de producción, estructuras de empresa, etc. Además, las características actuales de este modelo favorecen la representación de cualquier tipo de sistema y a cualquier nivel de abstracción o refinamiento, lo cual da lugar a que se aplique tanto a la representación de problemas que vayan a ser tratados mediante un sistema informatizado, como manual.

Gracias al modelo Entidad-Relación, creado por **Peter Chen** en los años setenta, se puede representar el mundo real mediante una serie de símbolos y expresiones determinados. El modelo de datos Entidad/Relación (E/R ó E-R) está basado en una percepción consistente en objetos básicos llamados **entidades** y **relaciones** entre estos objetos, estos y otros conceptos se desarrollan a continuación.

La información con la que el modelo Entidad-Relación trabaja ha de ser lo más detallada y fiel posible a la realidad del problema a representar.

Verdadero ◉ Falso ○

3.- Entidades.

Caso práctico

*—¿Cada una de las tablas que hemos estado generando equivale a una entidad en el modelo E/R? —Pregunta **Juan**.*

*—Algunas de ellas corresponden a entidades y otras a relaciones, depende del problema a resolver. Por ejemplo, la tabla USUARIO sí se correspondería con una entidad. Además, hay que tener cuidado a la hora de identificar entidades porque algunas veces podemos confundir entidades con atributos y viceversa —responde **Ada**.*

Para los miembros de BK Programación va a ser necesario que conozcan bien cómo se aplica este modelo si quieren que el proceso de creación de bases de datos sea correcto.

Si utilizamos las bases de datos para guardar información sobre cosas que nos interesan o que interesan a una organización, ¿No crees que hay que identificar esas cosas primero para poder guardar información sobre ellas? Para ello, vamos a describir un primer concepto, el de **Entidad**.

Una **entidad** puede ser un objeto físico, un concepto o cualquier elemento que queramos modelar, que tenga importancia para la organización y del que se desee guardar información. Cada entidad debe poseer alguna característica, o conjunto de ellas, que lo haga único frente al resto de objetos. Por ejemplo, podemos establecer una entidad llamada ALUMNO que tendrá una serie de características. El alumnado podría ser distinguido mediante su número de identificación escolar (NIE), por ejemplo.

Entidad: objeto real o abstracto, con características diferenciadoras capaces de hacerse distinguir de otros objetos.

¿Ponemos otro ejemplo? Supongamos que tienes que desarrollar el esquema conceptual para una base de datos de mapas de montaña, los elementos: camping, pista forestal, valle, río, pico, refugio, etc., son ejemplos de posibles entidades. A la hora de identificar las entidades, hemos de pensar en nombres que tengan especial importancia dentro del lenguaje propio de la organización o sistema que vaya a utilizar dicha base de datos. Pero no siempre una entidad puede ser concreta, como un camping o un río, en ocasiones puede ser abstracta, como un préstamo, una reserva en un hotel o un concepto.

Un **conjunto de entidades** serán un grupo de entidades que poseen las mismas características o propiedades. Por ejemplo, al conjunto de personas que realizan reservas para un hotel de montaña determinado, se les puede definir como el conjunto de entidades cliente. El conjunto de entidades río, representará todos los ríos existentes en una determinada zona. Por lo general, se suele utilizar el término entidad para identificar conjuntos de entidades. Cada elemento del conjunto de entidades será una ocurrencia de entidad.

Si establecemos un símil con la Programación Orientada a Objetos, podemos decir que el concepto de **entidad** es análogo al de **instancia de objeto** y que el concepto de **conjunto de entidades** lo es al de **clase**.

En el modelo Entidad/Relación, la representación gráfica de las entidades se realiza mediante el nombre de la entidad encerrado dentro de un rectángulo. A continuación se muestra la representación de la entidad CLIENTE.

3.1.- Tipos: fuertes y débiles.

Las entidades pueden ser clasificadas en dos grupos:

a. **Entidades Fuertes o Regulares:**
Son aquellas que tienen existencia por sí mismas, es decir, su existencia no depende de la existencia de otras entidades. Por ejemplo, en una base de datos hospitalaria, la existencia de instancias concretas de la entidad DOCTOR no depende de la existencia de instancias u objetos de la entidad PACIENTE. En el modelo E/R las entidades fuertes se representan como hemos indicado anteriormente, con el nombre de la entidad encerrado dentro de un rectángulo.

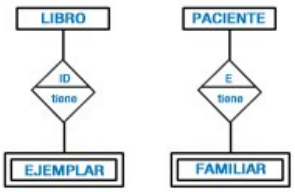
b. **Entidades débiles:**
Son aquellas cuya existencia depende de la existencia de otras instancias de entidad. Por ejemplo, consideremos las entidades EDIFICIO y AULA. Supongamos que puede haber aulas identificadas con la misma numeración pero en edificios diferentes. La numeración de cada aula no identificará completamente cada una de ellas. Para poder identificar completamente un aula es necesario saber también en qué edificio está localizada. Por tanto, la existencia de una instancia de una entidad débil depende de la existencia de una instancia de la entidad fuerte con la que se relaciona.

Entidad Débil: Es un tipo de entidad cuyas propiedades o atributos no la identifican completamente, sino que sólo la identifican de forma parcial. Esta entidad debe participar en una relación que ayude a identificarla.

En el modelo E/R una entidad débil se representa con el nombre de la entidad encerrado en un rectángulo doble. En el gráfico se muestra la representación de la entidad AULA.

Las entidades débiles presentan dos **tipos de dependencia**:

- ✓ **Dependencia en existencia**: entre entidades, si desaparece una instancia de entidad fuerte desaparecerán las instancias de entidad débiles que dependan de la primera. La representación de este tipo de dependencia incluirá una E en el interior de la relación débil.
- ✓ **Dependencia en identificación**: debe darse una dependencia en existencia y además, una ocurrencia de la entidad débil no puede identificarse por sí misma, debiendo hacerse mediante la clave de la entidad fuerte asociada. La representación de este tipo de dependencia incluirá una ID en el interior de la relación débil.



Tanto las entidades fuertes como las débiles se nombran habitualmente con sustantivos en singular.

Puede ser que haya algunos conceptos que aún no hemos desarrollado (relación, atributo y clave) y que se están utilizando para describir los tipos de dependencias, no te preocupes, en los siguientes epígrafes te los describimos claramente.

Identifica cuál de las siguientes entidades no podría ser considerada como entidad débil:

- (•) **PROVEEDOR (perteneciente a una base de datos de gestión de stocks).**
- ( ) PAGO (perteneciente a una base de datos bancaria).
- ( ) FAMILIAR (perteneciente a una base de datos hospitalaria).

Efectivamente, esta entidad puede existir por sí misma sin depender de otras ocurrencias de entidad. Además, posee propiedades o atributos propios que la identifican frente a otras ocurrencias de la misma entidad.

4.- Atributos.

Caso práctico

__Juan__ muestra a María qué atributos han creado para la tabla JUEGOS, pero al aplicar el modelo Entidad-Relación se ha dado cuenta de que le falta algún atributo más.

__María__ esta dibujando la entidad JUEGOS y sus atributos asociados. Ahora va a añadir gráficamente un atributo que recoja la productora de software asociada a cada juego.

¿Cómo guardamos información de cada entidad? A través de sus **atributos**. Las entidades se representan mediante un conjunto de **atributos**. Éstos describen características o propiedades que posee cada miembro de un conjunto de entidades. El mismo atributo establecido para un conjunto de entidades o, lo que es lo mismo, para un tipo de entidad, almacenará información parecida para cada ocurrencia de entidad. Pero, cada ocurrencia de entidad tendrá su propio valor para cada atributo.

Atributo: Cada una de las propiedades o características que tiene un tipo de entidad o un tipo de relación se denomina atributo; los atributos toman valores de uno o varios dominios.

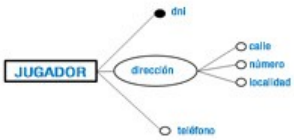
Por tanto, un atributo se utilizará para guardar información sobre alguna característica o propiedad de una entidad o relación. Ejemplos de atributos pueden ser: altura, color, peso, DNI, fecha, etc. todo dependerá de la información que sea necesaria almacenar.

En el modelo Entidad/Relación los atributos de una entidad son representados mediante el nombre del atributo rodeado por una elipse. La elipse se conecta con la entidad mediante una línea recta. Cada atributo debe tener un nombre único que haga referencia al contenido de dicho atributo. Los nombres de los atributos se deben escribir en letra minúscula. En el gráfico se representan algunos de los atributos para la entidad PACIENTE.

Al conjunto de valores permitidos para un atributo se le denomina **dominio**. Todos los posibles valores que puede tomar un atributo deberán estar dentro del dominio. Varios atributos pueden estar definidos dentro del mismo dominio. Por ejemplo, los atributos nombre, apellido primero y apellido segundo de la entidad PACIENTE, están definidos dentro del dominio de cadenas de caracteres de una determinada longitud.

Aunque los dominios suelen ser amplios (números enteros, reales, cadenas de caracteres, etc.), a la hora de llevar a cabo el desarrollo de una base de datos, es mejor establecer unos límites adecuados para que el sistema gestor de la base de datos lleve a cabo las verificaciones oportunas en los datos que se almacenen, garantizando así la integridad de éstos.

4.1.- Tipos de atributos.



¿Todos los atributos son iguales? Claro que no. Existen varias características que hacen que los atributos asociados a una entidad o relación sean diferentes, los clasificaremos según varios criterios.

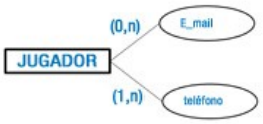
a. **Atributos obligatorios y opcionales. Atributo obligatorio**: es aquél que ha de estar siempre definido para una entidad o relación. Por ejemplo, para la entidad JUGADOR será necesario tener algún atributo que identifique cada ocurrencia de entidad, podría ser su DNI. Una clave o llave es un atributo obligatorio. **Atributo opcional**: es aquél que podría ser definido o no para la entidad. Es decir, puede haber ocurrencias de entidad para las que ese atributo no esté definido o no tenga valor.
b. Atómicos o compuestos.

Atributo simple o atómico: es un atributo que no puede dividirse en otras partes o atributos, presenta un único elemento. No es posible extraer de este atributo partes más pequeñas que puedan tener significado. Un ejemplo de este tipo de atributos podría ser el atributo dni de la entidad JUGADOR del gráfico.

Atributo compuesto: son atributos que pueden ser divididos en subpartes, éstas constituirán otros atributos con significado propio. Por ejemplo, la dirección del jugador podría considerarse como un atributo compuesto por la calle, el número y la localidad.

c. **Atributos monovaluados o multivaluados.**

Atributo monovaluado: es aquél que tiene un único valor para cada ocurrencia de entidad. Un ejemplo de este tipo de atributos es el dni.



Atributo multivaluado: es aquél que puede tomar diferentes valores para cada ocurrencia de entidad. Por ejemplo, la dirección de e-mail de un empleado podría tomar varios valores para alguien que posea varias cuentas de correo. En este tipo de atributos hay que tener en cuenta los siguientes conceptos:

- ✓ La **cardinalidad de un atributo** indica el número mínimo y el número máximo de valores que puede tomar para cada ejemplar de la entidad o relación a la que pertenece.
- ✓ La **cardinalidad mínima** indica la cantidad de valores del atributo que debe existir para que la entidad sea válida. Este número casi siempre es **0** o **1**. Si es **0**, el atributo podría no contener ningún valor y si es **1**, el atributo debe tener un valor.
- ✓ La **cardinalidad máxima** indica la cantidad máxima de valores del atributo que puede tener la entidad. Por lo general es **1** o **n**. Si es **1**, el atributo no puede tener más que un valor, si es **n**, el atributo puede tener múltiples valores y no se especifica la cantidad absoluta.

El atributo E_mail de la figura, puede ser opcional y no contener ningún valor, o bien, almacenar varias cuentas de correo electrónico de un jugador. Como ves, la cardinalidad representada en la imagen es (0,n).

d. **Atributos derivados o almacenados**: el valor de este tipo de atributos puede ser obtenido del valor o valores de otros atributos relacionados. Un ejemplo clásico de atributo derivado es la edad. Si se ha almacenado en algún atributo la fecha de nacimiento, la edad es un valor calculable a partir de dicha fecha.

Rellena los huecos en blanco con los conceptos adecuados.

Si en nuestra base de datos tenemos una entidad USUARIO, los atributos password y login deberán ser atributos obligatorios ya que son imprescindibles para iniciar o jugar partidas. En cambio, un posible atributo ranking que indique en qué posición se encuentra el usuario entre todos los jugadores, podría considerarse un atributo derivado si tenemos en cuenta la puntuación obtenida por cada usuario.

4.2.- Claves.

En el apartado anterior hablábamos de un tipo de atributo especial obligatorio, **las claves o llaves**. Ahora es el momento de abordar con mayor detalle este concepto.

Está claro que es necesario identificar correctamente cada ocurrencia de entidad o relación, de este modo el tratamiento de la información que se almacena podrá realizarse adecuadamente. Esta distinción podría llevarse a cabo tomando todos los valores de todos los atributos de una entidad o relación. Pero, en algunas ocasiones, sabemos que puede no ser necesario utilizar todos, bastando con un subconjunto de ellos. Aunque puede ocurrir que ese subconjunto tenga idénticos valores para varias entidades, por lo que cualquier subconjunto no será válido.

Por tanto, los valores de los atributos de una entidad deben ser tales que permitan **identificar unívocamente** a la entidad. En otras palabras, no se permite que ningún par de entidades tengan exactamente los mismos valores de sus atributos. Teniendo en cuenta esto, presta atención a los siguientes conceptos:

Superclave (Superllave): Es cualquier conjunto de atributos que permite identificar de forma única a una ocurrencia de entidad. Una superclave puede tener atributos no obligatorios, es decir, que no identificarían por si solos una ocurrencia de entidad.

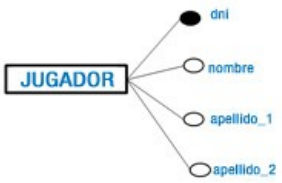
Clave candidata: Si de una superclave no es posible obtener ningún subconjunto que sea a su vez superclave, decimos que dicha superclave es clave candidata.

Clave primaria (Primary Key): También llamada llave primaria o clave principal. De todas las claves candidatas, el diseñador de la base de datos ha de escoger una, que se denominará clave principal o clave primaria. La clave primaria es un atributo o conjunto de ellos, que toman valores únicos y distintos para cada ocurrencia de entidad, identificándola unívocamente. No puede contener valores nulos.

Claves alternativas: son el resto de claves candidatas que no han sido escogidas como clave primaria.

La representación en el modelo Entidad/Relación de las claves primarias puede realizarse de dos formas:

- ✓ Si se utilizan elipses para representar los atributos, se subrayarán aquellos que formen la clave primaria.
- ✓ Si se utilizan círculos para representar los atributos, se utilizará un círculo negro en aquellos que formen la clave primaria.



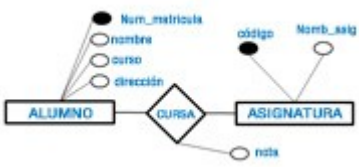
Sea la entidad TRABAJADOR, con los atributos nombre, apellido_1, apellido_2, dni, numero_afiliacion_ss, fecha_nacimiento y código_empresa. ¿Los atributos nombre, apellido_1 y apellido_2 podrían formar una clave candidata?

- ○ Sí, y podrían ser elegidos para ser la clave primaria de TRABAJADOR.
- ○ No, para esta entidad sólo el atributo dni será la clave primaria.
- ◉ **No, si tenemos en cuenta que puede haber varios trabajadores con el mismo nombre y apellidos.**

Efectivamente, los atributos dni y numero_afiliacion_ss, serían dos claves candidatas adecuadas. Si escogemos dni como clave primaria, numero_afiliacion_ss quedaría como clave alternativa.

4.3.- Atributos de una relación.

Una relación puede también tener atributos que la describan. Para ilustrar esta situación, observa el siguiente ejemplo.



Consideremos la relación CURSA entre las entidades ALUMNO y ASIGNATURA. Podríamos asociar a la relación CURSA un atributo nota para especificar la nota que ha obtenido un alumno/a en una determinada asignatura.

Otro ejemplo típico son las relaciones que representan **históricos**. Este tipo de relaciones suele constar de datos como fecha y hora. Cuando se emite una factura a un cliente o se le facilita un

duplicado de la misma, es necesario registrar el momento en el que se ha realizado dicha acción. Para ello, habrá que crear un atributo asociado a la relación entre la entidad CLIENTE y FACTURA que se encargue de guardar la fecha de emisión.

En el modelo Entidad/Relación la representación de atributos asociados a relaciones es exactamente igual a la que utilizábamos para entidades. Podremos utilizar una elipse con el nombre del atributo en su interior, conectada con una línea a la relación, o bien, un círculo blanco conectado con una línea a la relación y junto a él, el nombre del atributo. En el gráfico puedes ver esta segunda representación.

5.- Relaciones.

Caso práctico

__María__ ha identificado claramente las entidades y atributos que van a intervenir en su esquema, pero duda a la hora de representar cómo se van a relacionar dichas entidades.

__Ada__ le indica que es muy importante leer muy bien el documento de especificación de requerimientos del caso real a modelar, ya que de éste se desprenderán las particularidades de las relaciones entre las entidades que acaba de identificar.

—Representar una relación gráficamente en el modelo E/R es sencillo, pero lo interesante es dotar a esa representación de los elementos gráficos adecuados que reflejen fielmente cómo es en realidad: grado, cardinalidad, etc.—comenta __Ada__.

¿Cómo interactúan entre sí las entidades? A través de las relaciones. La relación o interrelación es un elemento del modelo Entidad/Relación que permite relacionar datos entre sí. En una relación se asocia un elemento de una entidad con otro de otra entidad.

Relación: es una asociación entre diferentes entidades. En una relación no pueden aparecer dos veces relacionadas las mismas ocurrencias de entidad.

La representación gráfica en el modelo Entidad/Relación corresponde a un rombo en cuyo interior se encuentra inscrito el nombre de la relación. El rombo estará conectado con las entidades a las que relaciona, mediante líneas rectas, que podrán o no acabar en punta de flecha según el tipo de relación.

Cuando debas dar un nombre a una relación procura que éste haga referencia al objetivo o motivo de la asociación de entidades. Se suelen utilizar verbos en singular. Algunos ejemplos podrían ser: forman, poseen, atiende, contrata, hospeda, supervisa, imparte, etc.

En algunas ocasiones, es interesante que en las líneas que conectan las entidades con la relación, se indique el papel o rol que desempeña cada entidad en la relación. Como se verá más adelante, los papeles o roles son especialmente útiles en relaciones reflexivas.

Para describir y definir adecuadamente las relaciones existentes entre entidades, es imprescindible conocer los siguientes conceptos:

- ✓ **Grado** de la relación.
- ✓ **Cardinalidad de la relación.**
- ✓ **Cardinalidades de las entidades.**

A continuación desarrollamos cada uno de ellos.

5.1.- Grado de una relación.

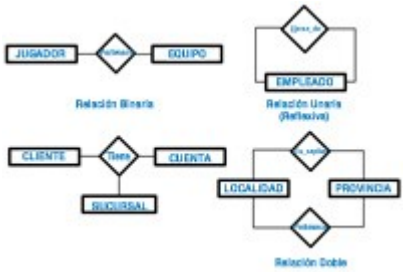
¿Pueden intervenir varias entidades en una misma relación? Claro que sí, en una relación puede intervenir una única entidad o varias.

Grado de una relación: número de entidades que participan en una relación.

En función del grado se pueden establecer diferentes tipos de relaciones:

- ✓ **Relación Unaria o de grado 1**: Es aquella relación en la que participa una única entidad. También llamadas reflexivas o recursivas.

- ✓ **Relación Binaria o de grado 2**: Es aquella relación en la que participan dos entidades. En general, tanto en una primera aproximación, como en los sucesivos refinamientos, el esquema conceptual de la base de datos buscará tener sólo este tipo de relaciones.
- ✓ **Relación Ternaria o de grado 3**: Es aquella relación en la que participan tres entidades al mismo tiempo.
- ✓ **Relación N-aria o de grado n**: Es aquella relación que involucra n entidades. Este tipo de relaciones no son usuales y deben ser simplificadas hacia relaciones de menor grado.
- ✓ **Relación doble**: ocurre cuando dos entidades están relacionadas a través de dos relaciones. Este tipo de relaciones son complejas de manejar.



En este gráfico puedes observar cada uno de los tipos de relaciones en función de su grado y su representación gráfica en el modelo Entidad/Relación.

Rellena los huecos con los conceptos adecuados.

En la relación unaria que puedes ver en el gráfico anterior, un empleado puede ejercer el rol de jefe o el rol de subordinado.

*Al ser una relación reflexiva que relaciona una entidad consigo misma, un empleado puede ejercer el rol de **jefe** sobre uno o varios empleados y, a su vez, podría ejercer el rol de **subordinado** bajo las ordenes de un jefe.*

5.2.- Cardinalidad de relaciones.

¿Qué es eso de la cardinalidad? En matemáticas, el cardinal de un conjunto es el número de elementos que lo forman. Este concepto puede extrapolarse a las relaciones con las que estamos tratando.

Cardinalidad de una relación: Es el número máximo de ocurrencias de cada entidad que pueden intervenir en una ocurrencia de relación. La cardinalidad vendrá expresada siempre para relaciones entre dos entidades. Dependiendo del número de ocurrencias de cada una de las entidades pueden existir relaciones **uno a uno, uno a muchos, muchos a uno y muchos a muchos**.

Observa el siguiente ejemplo, la cardinalidad indicará el número de ocurrencias de la entidad JUGADOR que se relacionan con cada ocurrencia de la entidad EQUIPO y viceversa. Podríamos hacer la siguiente lectura: un jugador pertenece a un equipo y a un equipo pueden pertenecer varios jugadores.

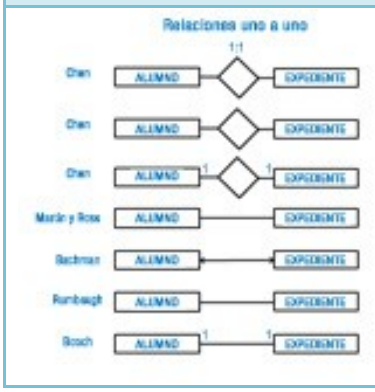
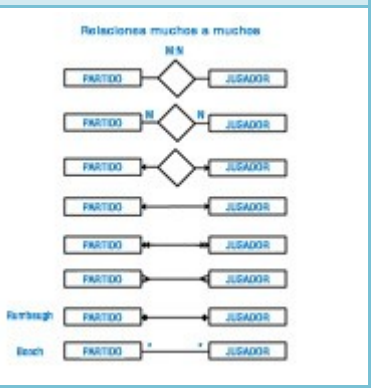
Una posible representación de la cardinalidad de las relaciones es la que hemos visto en el ejemplo anterior. Podríamos representar el resto de cardinalidades mediante las etiquetas **1:1**, **1:N**, **N:1**, **M:N** que se leerían respectivamente: uno a uno, uno a muchos, muchos a uno y muchos a muchos.
Veamos en detalle el significado de cada una de estas cardinalidades:

- ✓ **Relaciones uno a uno (1:1)**. Sean las entidades A y B, una instancia u ocurrencia de la entidad A se relaciona únicamente con otra instancia de la entidad B y viceversa. Por ejemplo, para cada ocurrencia de la entidad ALUMNO sólo habrá una ocurrencia relacionada de la entidad EXPEDIENTE y viceversa. O lo que es lo mismo, un alumno tiene un expediente asociado y un expediente sólo pertenece a un único alumno.
- ✓ **Relaciones uno a muchos (1:N)**. Sean las entidades A y B, una ocurrencia de la entidad A se relaciona con muchas ocurrencias de la entidad B y una ocurrencia de la entidad B sólo estará relacionada con una única ocurrencia de la entidad A. Por ejemplo, para cada ocurrencia de la entidad DOCENTE puede haber varias ocurrencias de la entidad ASIGNATURA y para varias ocurrencias de la entidad ASIGNATURA sólo habrá una ocurrencia relacionada de la entidad DOCENTE (si se establece que una asignatura sólo puede ser impartida por un único docente). O

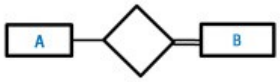
lo que es lo mismo, un docente puede impartir varias asignaturas y una asignatura sólo puede ser impartida por un único docente.

- ✓ **Relaciones muchos a uno (N:1)**. Sean las entidades A y B, una ocurrencia de la entidad A está asociada con una única ocurrencia de la entidad B y un ejemplar de la entidad B está relacionado con muchas ocurrencias de la entidad A. Por ejemplo, Un JUGADOR pertenece a un único EQUIPO y a un EQUIPO pueden pertenecer muchos jugadores.
- ✓ **Relaciones muchos a muchos (M:N)**. Sean las entidades A y B, un ejemplar de la entidad A está relacionado con muchas ocurrencias de la entidad B y viceversa. Por ejemplo, un alumno puede estar matriculado en varias asignaturas y en una asignatura pueden estar matriculados varios alumnos.

La cardinalidad de las relaciones puede representarse de varias maneras en los esquemas del modelo Entidad/Relación. A continuación, te ofrecemos un resumen de las notaciones clasificadas por autores, más empleadas en la representación de cardinalidad de relaciones.

Notaciones para representación de cardinalidad de relaciones.		
Relaciones uno a uno.	**Relaciones uno a muchos.**	**Relaciones muchos a muchos.**
		

5.3.- Cardinalidad de entidades.

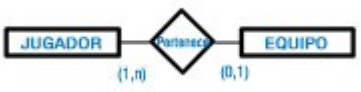


Si existe cardinalidad en las relaciones, supondrás que también existe para las entidades. Estás en lo cierto, la **cardinalidad** con la que una entidad participa en una relación especifica el número mínimo y el número máximo de correspondencias en las que puede tomar parte cada ejemplar de dicha entidad. Indica el número de relaciones en las que una entidad puede aparecer.

Sean las entidades A y B, la participación de la entidad A en una relación es **obligatoria (total)** si la existencia de cada una de sus ocurrencias necesita como mínimo de una ocurrencia de la entidad B (ver figura). En caso contrario, la participación es **opcional (parcial)**.

La cardinalidad de una entidad se representa con el número mínimo y máximo de correspondencias en las que puede tomar parte cada ejemplar de dicha entidad, entre paréntesis. Su representación gráfica será, por tanto, una etiqueta del tipo (0,1), (1,1), (0,N) o (1,N). El significado del primer y segundo elemento del paréntesis corresponde a **(cadinalidad mínima, cardinalidad máxima)**:

- ✓ **Cardinalidad mínima**. Indica el número mínimo de asociaciones en las que aparecerá cada ocurrencia 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). El valor 0 se pondrá cuando la participación de la entidad sea opcional.
- ✓ **Cardinalidad máxima**. Indica el número máximo de relaciones en las que puede aparecer cada ocurrencia de la entidad. Puede ser uno, otro valor concreto mayor que uno (tres por ejemplo) o muchos (se representa con n).



Veámoslo más claro a través del siguiente ejemplo: un JUGADOR pertenece como mínimo a ningún EQUIPO y como máximo a uno (0,1) y, por otra parte, a un EQUIPO pertenece como mínimo un JUGADOR y como máximo varios (1,n). Como puedes ver, la cardinalidad (0,1) de JUGADOR se ha colocado junto a la entidad EQUIPO para representar que un jugador puede no pertenecer a ningún equipo o como máximo a uno. Para la cardinalidad de EQUIPO ocurre igual, se coloca su cardinalidad junto a la entidad JUGADOR para expresar que en un equipo hay mínimo un jugador y máximo varios. Ten en cuenta que cuando se representa la cardinalidad de una entidad, el paréntesis y sus valores han de colocarse junto a la entidad con la que se relaciona. Es decir en el lado opuesto a la relación.
La cardinalidad de entidades también puede representarse en el modelo Entidad/Relación con la notación que se representa en la imagen de la derecha. Por tanto, el anterior ejemplo quedaría representado así:

Supongamos que seguimos diseñando una base de datos para un sitio de juegos online. En un punto del proceso de diseño se ha de modelar el siguiente requisito: cada usuario registrado podrá crear las partidas que desee (a las que otros usuarios pueden unirse), pero una partida solo podrá estar creada por un único usuario. Un usuario podrá o no crear partidas. ¿Cuáles serían las etiquetas del tipo (cardinalidad mínima, cardinalidad máxima) que deberían ponerse junto a las entidades USUARIO y PARTIDA respectivamente, si éstas están asociadas por la relación CREAR (partida)?

- ○ (1,N) y (0,N)
- ○ (1,1) y (1,N)
- ◉ **(1,1) y (0,N)**

Efectivamente, con estas cardinalidades estarías indicando que un usuario puede crear varias partidas, o ninguna. Por otra parte, una partida deberá estar creada exclusivamente por un único usuario.

6.- Simbología del modelo E/R.

Caso práctico

María *acaba de venir de comprar un tablón de anuncios de corcho. Va a colgarlo cerca su puesto de trabajo, junto al de Juan.*

–Mira ***Juan****, voy a imprimir estos gráficos en los que figuran los símbolos más utilizados a la hora de generar diagramas E/R. ¿Sabías que existen diferentes notaciones? – pregunta* ***María****.*

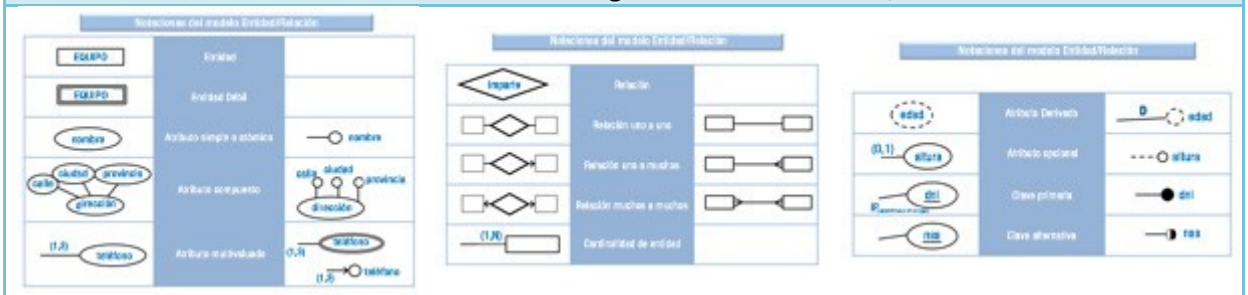
Juan*, que está buscando en su cajón la caja de las chinchetas, añade: –Me parece una idea genial y sí, sí que conocía la existencia de diferentes símbolos. Además, mientras buscaba en Internet algunos ejemplos, he visto que se pueden representar de diferentes maneras los mismos elementos.*

–Estupendo, así tendréis a mano la gran mayoría de símbolos y os será más cómodo interpretar los ejemplos que consultéis –comenta ***Ada****.*

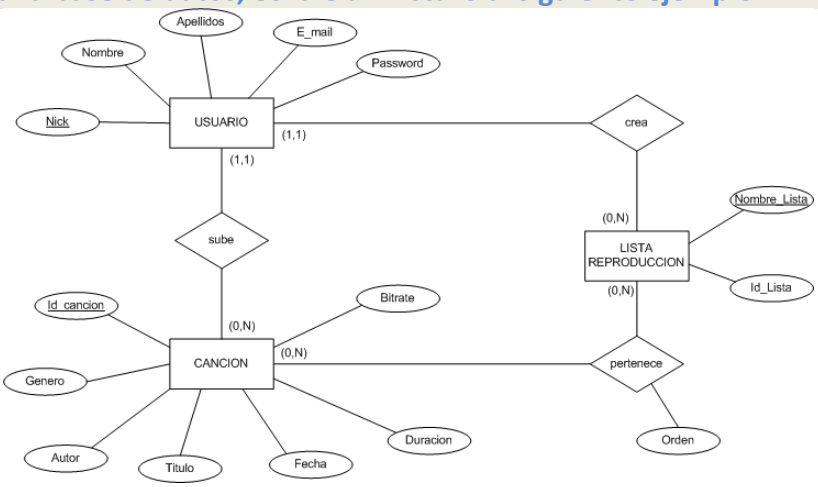
¿Recuerdas todos y cada uno de los símbolos que hemos utilizado a lo largo de esta unidad? Es probable que no. Para facilitar tu aprendizaje, te ofrecemos a continuación un resumen básico de los **símbolos utilizados en el modelo Entidad/Relación**. Verás que existen diferentes maneras de representar los mismos elementos, las que aquí se resumen te servirán para interpretar la gran mayoría de esquemas con los que te puedas encontrar.

Resumen básico de la simbología del modelo Entidad/Relación.

Resumen básico de la simbología del modelo Entidad/Relación



Si quieres ver un ejemplo de cómo se aplican algunas de estas notaciones en un esquema conceptual de una base de datos, échale un vistazo al siguiente ejemplo:



7.- El modelo E/R Extendido.

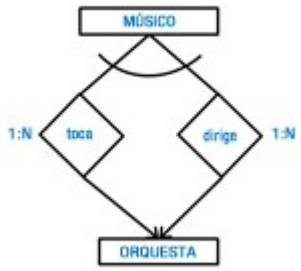
Caso práctico

Cuando la representación de determinadas entidades y relaciones se complique, los miembros de BK Programación necesitarán aplicar alguna técnica adicional que les permita realizar un modelado adecuado del problema. ***Ada****, está preparando una presentación en soporte informático con la que enseñará a* ***Juan*** *y* ***María*** *las nuevas posibilidades que les brinda el modelo Entidad-Relación Extendido.*

Hemos visto que a través del modelo Entidad/Relación se pueden modelar la gran mayoría de los requisitos que una base de datos debe cumplir. Pero existen algunos que ofrecen especial dificultad a la hora de representarlos a través de la simbología tradicional del modelo E/R. Para solucionar este problema, en el modelo Entidad/Relación Extendido se han incorporado nuevas extensiones que permiten mejorar la capacidad para representar circunstancias especiales. Estas extensiones intentan eliminar elementos de difícil o incompleta representación a través de la simbología existente, como por ejemplo relaciones con cardinalidad N:M, o la no identificación clara de entidades.

A continuación, se detallan estas nuevas características que convierten al modelo E/R tradicional en el **modelo Entidad/Relación Extendido**, como son: tipos de restricciones sobre las relaciones, especialización, generalización, conjuntos de entidades de nivel más alto y más bajo, herencia de atributos y agregación.

7.1.- Restricciones en las relaciones.



La primera extensión que el modelo Entidad/Relación Extendido incluye, se centra en la representación de una serie de restricciones sobre las relaciones y sus ejemplares, vamos a describirlas:

a. **Restricción de exclusividad.**
   Cuando existe **una entidad que participa en dos o más relaciones** y cada ocurrencia de dicha entidad sólo puede pertenecer a una de las relaciones únicamente, decimos que existe una restricción de exclusividad. Si la ocurrencia de entidad pertenece a una de las relaciones, no podrá formar parte de la otra. **O se produce una relación o se produce otra pero nunca ambas a la vez.**
   Por ejemplo, supongamos que un músico puede dirigir una orquesta o tocar en en ella, pero no puede hacer las dos cosas simultáneamente. Existirán por tanto, dos relaciones dirige y toca, entre las entidades MUSICO y ORQUESTA, estableciéndose una relación de exclusividad entre ellas.
   La representación gráfica en el modelo Entidad/Relación Extendido de una restricción de exclusividad se realiza **mediante un arco** que engloba a todas aquellas relaciones que son exclusivas.
b. **Restricción de exclusión.**
   Este tipo de restricción se produce **cuando las ocurrencias de las entidades sólo pueden asociarse utilizando una única relación**.
   Pongamos un ejemplo, supongamos que un monitor puede impartir diferentes cursos de perfeccionamiento para monitores, y que éste puede a su vez recibirlos. Pero si un monitor imparte un determinado curso, no podrá estar recibiéndolo simultáneamente y viceversa. Se establecerá, por tanto, una restricción de exclusión que se representa mediante una **línea discontinua entre las dos relaciones**, tal y como se muestra en el ejemplo.
c. **Restricción de inclusividad.**
   Este tipo de restricciones se aplican cuando es necesario modelar **situaciones en las que para que dos ocurrencias de entidad se asocien a través de una relación, tengan que haberlo estado antes a través de otra relación**.

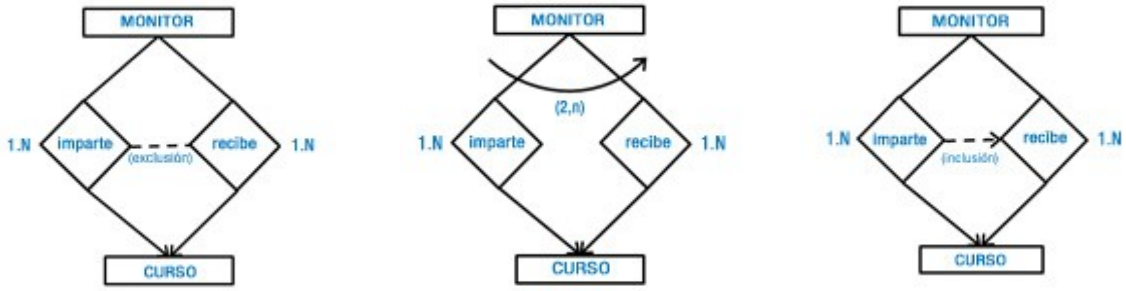
Siguiendo con el ejemplo anterior, supongamos que para que un monitor pueda impartir cursos de cocina sea necesario que reciba previamente dos cursos: nutrición y primeros auxilios. Como puedes ver, es posible que los cursos que el monitor deba recibir no tengan que ser los mismos que luego pueda impartir. Aplicando una restricción de inclusividad entre las relaciones imparte y recibe, estaremos indicando que cualquier ocurrencia de la entidad MONITOR que participa en una de las relaciones (imparte) tiene que participar obligatoriamente en la otra (recibe).

Se representará mediante un **arco acabado en flecha**, que partirá desde la relación que ha de cumplirse primero hacia la otra relación. Se indicará junto al arco la cardinalidad mínima y máxima de dicha restricción de inclusividad. En el ejemplo, (2,n) indica que un monitor ha de recibir 2 cursos antes de poder impartir varios.

d. **Restricción de inclusión.**

En algunas ocasiones aplicar una restricción de inclusividad no representa totalmente la realidad a modelar, entonces se hace necesario aplicar una restricción de inclusión que es aún más fuerte.

En nuestro ejemplo, si hemos de modelar que un monitor pueda impartir un curso, si previamente lo ha recibido, entonces tendremos que aplicar una restricción de inclusión. Con ella toda ocurrencia de la entidad MONITOR que esté asociada a una ocurrencia determinada de la entidad CURSO, a través de la relación imparte, ha de estar unida a la misma ocurrencia de la entidad CURSO a través de la relación recibe.



Supongamos que hemos de modelar mediante el modelo Entidad/Relación Extendido el siguiente requerimiento de una base de datos: Para que un hombre se divorcie de una mujer, primero ha de haber estado casado con ella.
Las entidades participantes son MUJER y HOMBRE, que estarán asociadas a través de dos relaciones: se casa, se divorcia. No tendremos en cuenta la cardinalidad de ambas relaciones.
¿Qué tipo de restricción sobre las relaciones hemos de establecer en nuestro esquema para representar correctamente este requisito?

- ○ Restricción de exclusividad
- ○ Restricción de inclusividad
- ◉ **Restricción de inclusión**

Efectivamente, este tipo de restricción establece la obligatoriedad de haber un casamiento para que pueda haber un divorcio y, además, las entidades que se relacionan a través de la relación se casa, deben ser las mismas que las participantes en se divorcia.

7.2.- Generalización y especialización.

La segunda extensión incorporada en el modelo Entidad/Relación Extendido se centra en nuevos tipos de relaciones que van a permitir modelar la realidad de una manera más fiel. Estos nuevos

tipos de relación reciben el nombre de **jerarquías** y se basan en los conceptos de generalización, especialización y herencia.

Cuando estamos diseñando una base de datos puede que nos encontremos con conjuntos de entidades que posean características comunes, lo que permitiría crear un tipo de entidad de nivel más alto que englobase dichas características. Y a su vez, puede que necesitemos dividir un conjunto de entidades en diferentes subgrupos de entidades por tener éstas, características diferenciadoras. Este proceso de refinamiento ascendente/descendente, permite expresar mediante la generalización la existencia de tipos de entidades de nivel superior que engloban a conjuntos de entidades de nivel inferior. A los conjuntos de entidades de nivel superior también se les denomina **superclase o supertipo**. A los conjuntos de entidades de nivel inferior se les denomina **subclase o subtipo**.

Por tanto, existirá la posibilidad de realizar una **especialización de una superclase en subclases**, y análogamente, **establecer una generalización de las subclases en superclases**. La generalización es la reunión en una superclase o supertivo de entidad de una serie de subclases o subtipos de entidades, que poseen características comunes. Las subclases tendrán otras características que las diferenciarán entre ellas.

¿Cómo detectamos una generalización? Podremos identificar una generalización cuando encontremos una serie de atributos comunes a un conjunto de entidades, y otros atributos que sean específicos. Los atributos comunes conforman la superclase o supertipo y los atributos específicos la subclase o subtipo.

Las jerarquías se caracterizan por un concepto que hemos de tener en cuenta, **la herencia**. A través de la herencia los atributos de una superclase de entidad son heredados por las subclases. Si una superclase interviene en una relación, las subclases también lo harán.

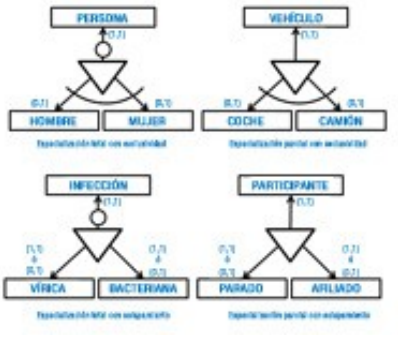
¿Cómo se representa una generalización o especialización? Existen varias notaciones, pero hemos de convenir que la relación que se establece entre una superclase de entidad y todos sus subtipos se expresa a través de las palabras ES UN, o en notación inglesa IS A, que correspondería con ES UN TIPO DE. Partiendo de este punto, una jerarquía se representa mediante un triángulo invertido, sobre él quedará la entidad superclase y conectadas a él a través de líneas rectas, las subclases.

En el ejemplo de la imagen, las subclases INVITADO, REGISTRADO y ADMINISTRADOR constituyen subclases de la superclase USUARIO. Cada una de ellas aporta sus propias características y heredan las pertenecientes a su superclase.

Una generalización/especialización podrá tener las siguientes restricciones semánticas:

- ✓ **Totalidad**: una generalización/especialización será total si todo ejemplar de la superclase pertenece a alguna de las subclases.
- ✓ **Parcialidad**: una generalización/especialización será parcial si no todos los ejemplares de la superclase pertenecen a alguna de las subclases.
- ✓ **Solapamiento**: una generalización/especialización presentará solapamiento si un mismo ejemplar de la superclase puede pertenecer a más de una subclase.
- ✓ **Exclusividad**: una generalización/especialización presentará exclusividad si un mismo ejemplar de la superclase pertenece sólo a una subclase.

Las diferentes restricciones semánticas descritas tienen su representación gráfica, a través del gráfico que a



continuación te mostramos podrás entender mejor su funcionamiento.

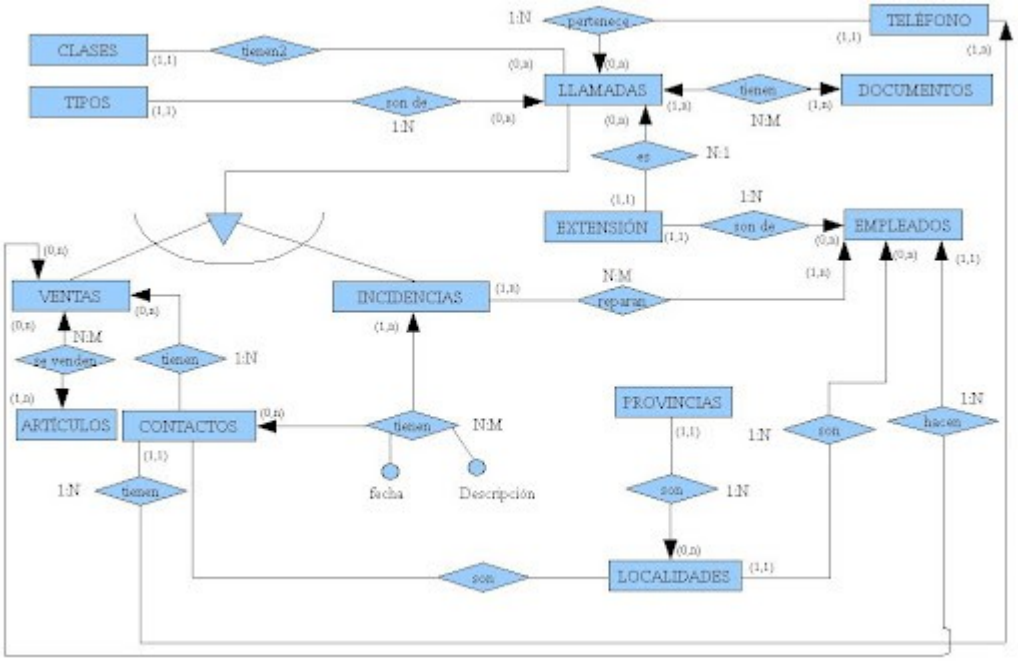
Ejercicio resuelto

Supongamos la existencia de dos entidades TURISMO y CAMION. Los atributos de la entidad TURISMO son: Num_bastidor, Fecha_fab, precio y Num_puertas. Los atributos de la entidad CAMION son: Num_bastidor, Fecha_fab, precio, Num_ejes y Tonelaje.

Si analizamos ambas entidades existen algunos atributos comunes y otros que no. Por tanto, podremos establecer una jerarquía. Para ello, reuniremos los atributos comunes y los asociaremos a una nueva entidad superclase denominada VEHICULO. Las subclases TURISMO y CAMI0N, con sus atributos específicos, quedarán asociadas a la superclase VEHICULO mediante una jerarquía parcial con solapamiento. En el siguiente gráfico puedes apreciar la transformación.

Respuesta:

Si quieres ver cómo se integra la representación de jerarquías dentro de un esquema conceptual completo, te proponemos observes el siguiente esquema:



7.3.- Agregación.

Abordamos ahora la tercera de las extensiones del modelo Entidad/Relación Extendido, la **Agregación**. En el modelo Entidad/Relación no es posible representar relaciones entre relaciones. La agregación es una abstracción a través de la cual las relaciones se tratan como entidades de nivel más alto, siendo utilizada para expresar relaciones entre relaciones o entre entidades y relaciones.

Supongamos un ejemplo en el que hemos de modelar la siguiente situación: una empresa de selección de personal realiza entrevistas a diferentes aspirantes. Puede ser que, de algunas de estas entrevistas a aspirantes, se derive una oferta de empleo, o no. En el siguiente gráfico se representan tres soluciones, las dos primeras erróneas y una tercera correcta, utilizando una agregación.
Como has podido observar, la representación gráfica de una agregación se caracteriza por englobar con un rectángulo las entidades y relación a abstraer. De este modo, se crea una nueva entidad agregada que puede participar en otras relaciones con otras entidades. En este tipo de relación especial de agregación, la cardinalidad máxima y mínima de la entidad agregada siempre será (1,1) no indicándose por ello en el esquema.
Existen dos clases de agregaciones:

- ✓ **Compuesto/componente**: Un todo se obtiene por la unión de diversas partes, que pueden ser objetos distintos y que desempeñan papeles distintos en la agregación. Teniendo esto en cuenta, esta abstracción permite representar que un todo o agregado se obtiene por la unión de diversas partes o componentes que pueden ser tipos de entidades distintas y que juegan diferentes roles en la agregación.
- ✓ **Miembro/Colección**: Un todo se obtiene por la unión de diversas partes del mismo tipo y que desempeñan el mismo papel en la agregación. Teniendo esto en cuenta, esta abstracción permite representar un todo o agregado como una colección de miembros, todos de un mismo tipo de entidad y todos jugando el mismo rol. Esta agregación puede incluir una restricción de orden de los miembros dentro de la colección (indicando el atributo de ordenación). Es decir, permite establecer un orden entre las partes.

En la siguiente figura puedes apreciar los tipos de agregación y su representación gráfica.

Con la agregación hemos terminado de detallar las extensiones más importantes del modelo Entidad/Relación Extendido. A lo largo de tu andadura por el mundo de las bases de datos y, en concreto, en todo lo relacionado con los esquemas conceptuales y diagramas Entidad/Relación, es probable que te encuentres con diferentes notaciones y simbologías. Algunas ya las hemos representado a lo largo de esta unidad y otras podrás encontrarlas en el enlaces que te ofrecemos a continuación. Además, puedes utilizar la información que te proponemos para reforzar y ampliar todo lo visto.
http://www.lsi.us.es/docencia/get.php?id=4564 (páginas 1 a 9). (0.33 MB)

Si hemos de representar a través del modelo E/R Extendido los alumnos pertenecientes a una clase, podríamos utilizar una agregación del tipo Compuesto/Componente.

Verdadero ○ **Falso** ◉

Al ser el alumnado un conjunto de elementos que representan el mismo rol en la relación, el tipo de agregación debería ser Miembro/Colección.

8.- Elaboración de diagramas E/R.

Caso práctico

–La verdad es que a través del esquema que estamos generando, queda más claro cómo es cada entidad y cada relación. Aplicando estas técnicas creo que vamos a ir afianzando un método fiable que podremos aplicar en futuros desarrollos – afirma ***Juan****.*

Ada*, está echando un vistazo a lo que llevan hecho* ***Juan*** *y* ***María****. – Efectivamente* ***Juan****, hay que ser metódicos y no descartar ningún paso, pues podríamos provocar errores en nuestros desarrollos. La confianza de nuestros clientes es vital y para ello hemos de obtener un producto con la mayor calidad posible.*

María *añade: –Supongo que según vayamos realizado proyectos parecidos mejoraremos nuestra técnica.*

Llegados a este punto, te surgirán varias dudas ¿Cómo creo un diagrama E/R? ¿Por dónde empiezo? ¿Y qué puedo hacer con todo lo visto? Son cuestiones totalmente normales cuando se comienza, no te preocupes, vamos a darte una serie de orientaciones para que puedas aplicar todos los conceptos aprendidos hasta ahora en la elaboración de diagramas Entidad/Relación.

Sabemos que en la fase de diseño conceptual de la base de datos, en la que nos encontramos, hemos de generar el diagrama E/R que representará de manera más sencilla el problema real a modelar, independientemente del Sistema Gestor de Base de Datos. Este esquema será como un plano que facilite la comprensión y solución del problema. Este diagrama estará compuesto por la representación gráfica, a través de la simbología vista, de los requisitos o condiciones que se derivan del problema a modelar.

Saltarnos este paso en el proceso de creación e implementación de una base de datos, supondría pérdida de información. Por lo que esta fase, requerirá de la creación de uno o varios esquemas previos más cercanos al mundo real, antes del paso a tablas del modelo relacional.

Te darás cuenta que, como en la programación, **la práctica es fundamental**. Los diagramas no siempre se crean del mismo modo y, en ocasiones, hay que retocarlos e incluso rehacerlos. A través de la resolución de diferentes problemas y la elaboración de múltiples diagramas, obtendrás la destreza necesaria para generar esquemas que garanticen una posterior y correcta conversión del modelo Entidad/Relación al modelo Relacional.

8.1.- Identificación de entidades y relaciones.

¡Manos a la obra! Lo primero que hemos de tener a nuestra disposición para poder generar un diagrama E/R adecuado es el conjunto de requerimientos, requisitos o condiciones que nuestra base de datos ha de cumplir. Es lo que se denomina el documento de especificación de requerimientos. En otras palabras, el enunciado del problema a modelar. Cuanto más completa y detallada sea la información de la que dispongamos, mucho mejor.

Suponiendo que conocemos la simbología del modelo Entidad/Relación y que entendemos su significado ¿Cómo empezamos? Las etapas para la creación del diagrama E/R se detallan a continuación:

a. **Identificación de entidades**: Es un proceso bastante intuitivo. Para localizar aquellos elementos que serán las entidades de nuestro esquema, analizaremos la especificación de requerimientos en busca de nombres o sustantivos. Si estos nombres se refieren a objetos importantes dentro del problema probablemente serán entidades. Tendremos en cuenta que nombres referidos a características, cualidades o propiedades no se convertirán en entidades.

Otra forma de identificar entidades es localizando objetos o elementos que existen por sí mismos. Por ejemplo: VEHICULO, PIEZA, etc. En otras ocasiones, la localización de varias características o propiedades puede dejar ver la existencia de una entidad.

¿Esto puede ser una entidad o no? Es una pregunta que se repite mucho cuando estamos en esta etapa. Algunos autores indican que para poder considerarse como entidad se deben cumplir tres reglas:

- ✓ Existencia propia.
- ✓ Cada ejemplar de un tipo de entidad debe poder ser diferenciado del resto de ejemplares.
- ✓ Todos los ejemplares de un tipo de entidad deben tener las mismas propiedades.

El número de entidades obtenidas debe ser manejable y según se vayan identificando se les otorgará **nombres, preferiblemente en mayúsculas, representativos** de su significado o función. De esta manera el diagrama será cada vez más legible.

b. **Identificación de relaciones**: Localizadas las entidades, debemos establecer qué relación existe entre ellas. Para ello, analizaremos de nuevo el documento de especificación de requerimientos en busca de **verbos o expresiones verbales que conecten unas entidades con otras**. En la gran mayoría de ocasiones encontraremos que las relaciones se establecen entre dos entidades (relaciones binarias), pero prestaremos especial atención a las relaciones entre más entidades y a las relaciones recursivas o relaciones unarias.

c. Cada una de las relaciones establecidas deberá tener asignado un **nombre, preferiblemente en minúsculas, representativo** de su significado o acción.

En ocasiones, el identificador de una relación está compuesto por varias palabras, como por ejemplo: es supervisado, trabaja para, etc. Es recomendable que utilices guiones bajos para unir las palabras que forman el identificador.

Dependiendo de la notación elegida, el siguiente paso será la representación de la cardinalidad (mínima y máxima) de las entidades participantes en cada relación y del tipo de correspondencia de la relación (1 a 1, 1 a muchos o muchos a muchos).

Si hemos encontrado alguna relación recursiva, reflexiva o unaria, hemos de representar en nuestro esquema los roles desempeñados por la entidad en dicha relación.

Rellena los huecos con los conceptos adecuados.

Las entidades suelen localizarse en el documento de especificación de requerimientos a través de sustantivos y las relaciones a través de verbos. Pero hemos de tener cuidado, no siempre los sustantivos representarán entidades, pues podría tratarse de atributos.

8.2.- Identificación de atributos, claves y jerarquías.

Sólo con la localización de entidades y relaciones no está todo hecho. Hemos de completar el proceso realizando las siguientes tareas:

a. **Identificación de atributos**: Volvemos sobre el documento de especificación de requerimientos para buscar nombres relativos a características, propiedades, identificadores o cualidades de entidades o relaciones. Resulta más sencillo si nos preguntamos ¿Qué información es necesario tener en cuenta de una u otra entidad o relación? Quizás no todos los atributos estén reflejados directamente en el documento de especificación de

requerimientos, aplicando el sentido común el diseñador podrá establecerlos en algunos casos y en otros, será necesario consultar e indagar en el problema.

Tendremos en cuenta si los atributos localizados son simples o compuestos, derivados o calculados y si algún atributo o conjunto de ellos se repite en varias entidades. Si se da este último caso, deberemos detenernos y plantear la posibilidad de establecer una jerarquía de especialización, o bien, dejar las entidades tal y como han sido identificadas.

Cada atributo deberá tener asignado un nombre, preferiblemente en minúsculas, representativo de su contenido o función. Además, siempre es recomendable recopilar la siguiente información de cada atributo:

- ✓ Nombre y descripción.
- ✓ Atributos simples que lo componen, si es atributo compuesto.
- ✓ Método de cálculo, si es atributo derivado o calculado.

En el caso de encontrar atributos asociados a relaciones con cardinalidad uno a muchos, se valorará asignar ese atributo o atributos a la entidad con mayor cardinalidad participante en la relación.

b. **Identificación de claves**: Del conjunto de atributos de una entidad se establecerán una o varias claves candidatas, escogiéndose una de ellas como clave o llave primaria de la entidad. Esta clave estará formada por uno o varios atributos que identificarán de manera unívoca cada ocurrencia de entidad. El proceso de identificación de claves permitirá determinar la fortaleza (al menos una clave candidata) o debilidad (ninguna clave candidata) de las entidades encontradas.

   Se representará la existencia de esta clave primaria mediante la notación elegida para la elaboración el diagrama E/R. Del mismo modo, se deberán representar adecuadamente las entidades fuertes o débiles.

c. **Determinación de jerarquías**: Como se ha comentado anteriormente, es probable que existan entidades con características comunes que puedan ser generalizadas en una entidad de nivel superior o superclase (jerarquía de generalización). Pero también, puede ser necesario expresar en el esquema las particularidades de diferentes ejemplares de un tipo de entidad, por lo que se crearán subclases o subtipos de una superclase o supertipo (jerarquía de especialización). Para ello, habrá que analizar con detenimiento el documento de especificación de requerimientos.

   Si se identifica algún tipo de jerarquía, se deberá representar adecuadamente según el tipo de notación elegida, determinando si la jerarquía es total/parcial o exclusiva/con solapamiento.

8.3.- Metodologías.

Hasta aquí, tenemos identificados los elementos necesarios para construir nuestro diagrama, pero ¿Existe alguna metodología para llevarlo a cabo? Sí, y además podremos utilizar varias. Partiremos de una versión preliminar del esquema conceptual o diagrama E/R que, tras sucesivos refinamientos, será modificado para obtener el diagrama E/R definitivo. Las metodologías o estrategias disponibles para la elaboración del esquema conceptual son las siguientes:

a. **Metodología Descendente (Top-Down)**: Se trata de partir de un esquema general e ir descomponiendo éste en niveles, cada uno de ellos con mayor número de detalles. Se parte de objetos muy abstractos, que se refinan paso a paso hasta llegar al esquema final.

b. **Metodología Ascendente (Bottom-Up)**: Inicialmente, se parte del nivel más bajo, los atributos. Se irán agrupando en entidades, para después ir creando las relaciones entre éstas y las posibles jerarquías hasta obtener un diagrama completo. Se parte de objetos atómicos que no pueden ser descompuestos y a continuación se obtienen abstracciones u objetos de mayor nivel de abstracción que forman el esquema.
c. **Metodología Dentro-fuera (Inside-Out)**: Inicialmente se comienza a desarrollar el esquema en una parte del papel y a medida que se analiza la especificación de requerimientos, se va completando con entidades y relaciones hasta ocupar todo el documento.
d. **Metodología Mixta**: Es empleada en problemas complejos. Se dividen los requerimientos en subconjuntos que serán analizados independientemente. Se crea un esquema que servirá como estructura en la que irán interconectando los conceptos importantes con el resultado del análisis de los subconjuntos creados. Esta metodología utiliza las técnicas ascendente y descendente. Se aplicará la técnica descendente para dividir los requerimientos y en cada subconjunto de ellos, se aplicará la técnica ascendente.

¿Cuál de estas metodologías utilizar? Cualquiera de ellas puede ser válida, todo dependerá de lo fácil y útil que te resulte aplicarlas. Probablemente y, casi sin ser consciente de ello, tú mismo crearás tu propia metodología combinando las existentes. Pero, como decíamos hace algunos epígrafes, **la práctica es fundamental**. Realizando gran cantidad de esquemas, analizándolos y llevando a cabo modificaciones en ellos es como irás refinando tu técnica de elaboración de diagramas E/R. Llegará un momento en que sólo con leer el documento de especificación de requerimientos serás capaz de ir construyendo en tu mente cómo será su representación sobre el papel, pero paciencia y ve paso a paso.

"Vísteme despacio, que tengo prisa". *Napoleón Bonaparte.*

Rellena los huecos con los conceptos adecuados.
La metodología en la que se parte de un alto nivel de abstracción y que, tras un proceso de refinamiento sucesivo, se obtiene el esquema final se denomina: Metodología descendente.

8.4.- Redundancia en diagramas E/R.

Una de las principales razones por las que las bases de datos aparecieron fue la eliminación de la redundancia en los datos ¿Y qué es la redundancia?

Redundancia: reproducción, reiteración, insistencia, reincidencia, reanudación. En bases de datos hace referencia al almacenamiento de los mismos datos varias veces en diferentes lugares.

La redundancia de datos puede provocar problemas como:

- ✓ **Aumento de la carga de trabajo**: al estar almacenado un dato en varios lugares, las operaciones de grabación o actualización de datos necesitan realizarse en varias ocasiones.
- ✓ **Gasto extra de espacio de almacenamiento**: al estar repetidos, los datos ocupan mayor cantidad de espacio en el medio de almacenamiento. Cuanto mayor sea la base de datos, más patente se hará esta problema.
- ✓ **Inconsistencia**: se produce cuando los datos que están repetidos, no contienen los mismos valores. Es decir, se ha actualizado su valor en un lugar y en otro no, por lo que no se sabría qué dato es válido y cual erróneo.

Para que una base de datos funcione óptimamente, hay que empezar realizando un buen diseño de ella. Es imprescindible que nuestros diagramas E/R controlen la redundancia y, para ello, debemos analizar el esquema y valorar qué elementos pueden estar incorporando redundancia a nuestra solución.

¿Dónde buscamos indicios de redundancia en nuestros esquemas? Existen lugares y elementos que podrían presentar redundancia, por ejemplo:

- ✓ **Atributos redundantes** cuyo contenido se calcula en función de otros. Un atributo derivado puede ser origen de redundancia.
- ✓ Varias entidades unidas circularmente o cíclica a través de varias relaciones, es lo que se conoce como **un ciclo**. En caso de existir un ciclo, deberemos tener en cuenta las siguientes condiciones, antes de poder eliminar dicha relación redundante:
  - ➔ Que el significado de las relaciones que componen el ciclo sea el mismo.
  - ➔ Que si eliminamos la relación redundante, el significado del resto de relaciones es el mismo.
  - ➔ Que si la relación eliminada tenía atributos asociados, éstos puedan ser asignados a alguna entidad participante en el esquema, sin que se pierda su significado.

Pero hay que tener en cuenta que **no siempre que exista un ciclo estaremos ante una redundancia**. Es necesario analizar detenidamente dicho ciclo para determinar si realmente existe o no redundancia.

Para finalizar, una apreciación. No toda redundancia es perjudicial. Existen ciertas circunstancias y condiciones en las que es conveniente (sobre todo a efectos de rendimiento) introducir cierta **redundancia controlada** en una base de datos. Por ejemplo, si el método de cálculo del valor de un determinado atributo derivado es complejo (varias operaciones matemáticas o de cadenas de caracteres, varios atributos implicados, etc.) y ralentiza el funcionamiento de la base de datos, quizá sea conveniente definir dicho atributo desde el principio y no considerarlo como un atributo redundante. La incorporación o no de redundancia controlada dependerá de la elección que haga el diseñador.

8.5.- Propiedades deseables de un diagrama E/R.

Cuando construimos un diagrama Entidad/Relación existen una serie de propiedades o características que éste debería cumplir. Quizá no se materialicen todas, pero hemos de intentar cubrir la gran mayoría de ellas. De este modo, conseguiremos que nuestros diagramas o esquemas conceptuales tengan mayor calidad.

Estas características o propiedades deseables se desglosan a continuación:

- ✓ **Completitud**: Un diagrama E/R será completo si es posible verificar que cada uno de los requerimientos está representado en dicho diagrama y viceversa, cada representación del diagrama tiene su equivalente en los requerimientos.
- ✓ **Corrección**: Un diagrama E/R será correcto si emplea de manera adecuada todos los elementos del modelo Entidad/Relación. La corrección de un diagrama puede analizarse desde dos vertientes:
  - ➔ **Corrección sintáctica**: Se producirá cuando no se produzcan representaciones erróneas en el diagrama.
  - ➔ **Corrección semántica**: Se producirá cuando las representaciones signifiquen exactamente lo que está estipulado en los requerimientos. Posibles errores semánticos serían: la utilización de un atributo en lugar de una entidad, el uso de una entidad en lugar de una relación, utilizar el mismo identificador para dos entidades o dos relaciones, indicar erróneamente alguna cardinalidad u omitirla, etc.
- ✓ **Minimalidad**: Un diagrama E/R será mínimo si se puede verificar que al eliminar algún concepto presente en el diagrama, se pierde información. Si un diagrama es redundante, no será mínimo.
- ✓ **Sencillez**: Un diagrama E/R será sencillo si representa los requerimientos de manera fácil de comprender, sin artificios complejos.

- ✓ **Legibilidad**: Un diagrama E/R será legible si puede interpretarse fácilmente. La legibilidad de un diagrama dependerá en gran medida del modo en que se disponen los diferentes elementos e interconexiones. Esta propiedad tiene mucho que ver con aspectos estéticos del diagrama.
- ✓ **Escalabilidad**: Un diagrama E/R será escalable si es capaz de incorporar posibles cambios derivados de nuevos requerimientos.

Si en un diagrama E/R asociamos un atributo a una entidad, pero este atributo debe asociarse realmente a una relación en la que interviene dicha entidad, estaríamos incumpliendo la propiedad de:

- ○ Completitud
- ◉ **Corrección semántica**
- ○ Corrección sintáctica

9.- Paso del diagrama E/R al modelo relacional.

Caso práctico

__Juan__ y __María__ ya han terminado de elaborar el diagrama E/R, con la ayuda de __Ada__. Las últimas modificaciones hechas en éste garantizan que todas las condiciones establecidas en el documento de especificación de requerimientos han sido representadas adecuadamente.

–¿Y ahora cómo se pasa este diagrama a una base de datos real? –pregunta __María__.

–Aún hay que obtener el "paso a tablas" de lo representado en el diagrama. En cuanto realicemos esa transformación tendremos los elementos necesarios para implementar nuestra base de datos en cualquier SGBD relacional –le aclara __Ada__.

Si analizamos todo el proceso descrito hasta el momento, la fase de diseño conceptual desarrollada, y que se materializa en el diagrama E/R, permite una gran independencia de las cuestiones relativas a la implementación física de la base de datos. El tipo de SGBD, las herramientas software, las aplicaciones, lenguajes de programación o hardware disponible no afectarán, al menos hasta el momento, a los resultados de esta fase.

Nuestro esquema conceptual habrá sido revisado, modificado y probado para verificar que se cumplen adecuadamente todos y cada uno de los requerimientos del problema a modelar. Este esquema representará el punto de partida para la siguiente fase, el diseño lógico de la base de datos.

El diseño lógico consistirá en la construcción de un esquema de la información relativa al problema, basado en un modelo de base de datos concreto. El esquema conceptual se transformará en un esquema lógico que utilizará los elementos y características del modelo de datos en el que esté basado el SGBD, para implementar nuestra base de datos. Como pudimos ver anteriormente, estos modelos podrán ser: el modelo en red, el modelo jerárquico y, sobre todo, el modelo relacional y el modelo orientado a objetos.

Para esta transformación será necesario realizar una serie de pasos preparatorios sobre el esquema conceptual obtenido en la fase de diseño conceptual. Nos centraremos en la simplificación y transformación del esquema para que el paso hacia el modelo de datos elegido (en este caso el modelo relacional) sea mucho más sencilla y efectiva.

Seguidamente, tomando como referencia el esquema modificado/simplificado, se realizará el paso de éste al modelo de datos relacional. Esta transformación requerirá de la aplicación de determinadas reglas y condiciones que garanticen la equivalencia entre el esquema conceptual y el esquema lógico.

Como paso posterior, sobre la información del esquema lógico obtenido, será necesario llevar a cabo un proceso que permitirá diseñar de forma correcta la estructura lógica de los datos. Este proceso recibe el nombre de normalización, que se conforma como un conjunto de técnicas que permiten validar esquemas lógicos basados en el modelo relacional.

Entonces, ¿qué pasos son los siguientes a dar? Resumiendo un poco, simplificaremos nuestro diagrama E/R, lo transformaremos al modelo relacional, aplicaremos normalización y obtendremos lo que se conoce en el argot como el paso a tablas del esquema conceptual o, lo que es lo mismo, el esquema lógico. Desde ese momento, basándonos en este esquema, podremos llevarnos nuestra base de datos a cualquier SGBD basado en el modelo relacional e implementarla físicamente. Esta implementación física será totalmente dependiente de las características del SGBD elegido.

9.1.- Simplificación previa de diagramas.

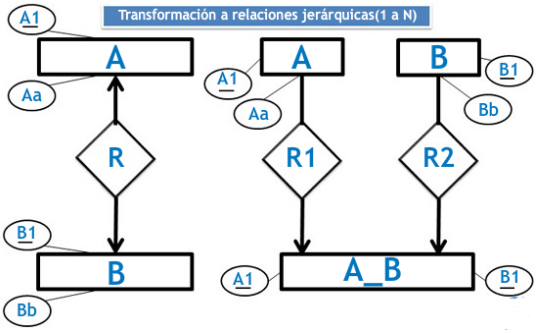
Existe un conjunto de procedimientos y normas que es necesario aplicar a nuestros diagramas E/R para que su transformación al modelo lógico basado en el modelo relacional, sea correcta y casi automática. Si aplicas correctamente estas pautas, conseguirás que el proceso de transformación sea fácil y fiable. Las transformaciones de las que estamos hablando son las siguientes:

- ✓ Transformación de relaciones n-arias en binarias.
- ✓ Eliminación de relaciones cíclicas.
- ✓ Reducción a relaciones jerárquicas (uno a muchos).
- ✓ Conversión de entidades débiles en fuertes.

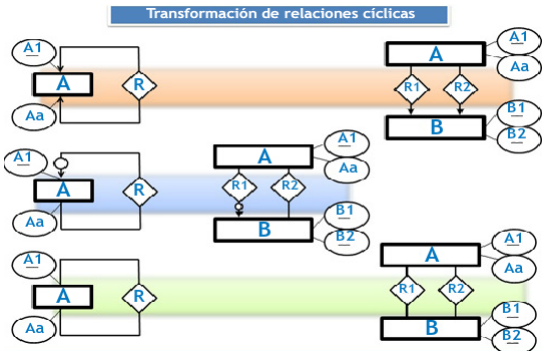
Veamos detalladamente cómo llevar a cabo las transformaciones de las que hemos estado hablando:

- ✓ **Transformación de atributos compuestos:** Los atributos compuestos de una entidad han de ser descompuestos en los atributos simples por los que están formados.
- ✓ **Transformación de atributos multivaluados:** Si nuestro diagrama incluye la existencia de un atributo multivaluado, este se ha de convertir en una entidad relacionada con la entidad de la que procede. Para esta nueva entidad se elegirá un nombre adecuado y tendrá un único atributo (el correspondiente al antiguo atributo múltiple). Este atributo es posible que funcione correctamente como clave primaria de la entidad pero a veces es posible que no. En este caso, la entidad que hemos creado puede que sea débil. Deberemos ajustar en cualquier caso correctamente las claves primarias.
- ✓ **Transformación a relaciones jerárquicas:** Se trata de transformar las relaciones con cardinalidad muchos a muchos (M a N) en relaciones con cardinalidad uno a muchos (1 a N). Observa la animación para comprender cómo se realiza la transformación. Si existiese algún atributo asociado a la relación n-aria, quedaría asociado a la nueva entidad que se crea.

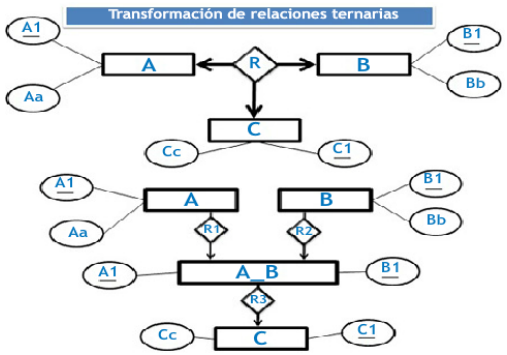
La relación R, que asocia la entidad A con la entidad B, tiene cardinalidad M a N (muchos a muchos). Para eliminarla, se crea una nueva entidad A_B cuya clave estará formada por las claves de A y B. Posteriormente se relacionan A y B mediante dos relaciones 1 a N (R1 y R2) con la nueva entidad A_B.



- ✓ **Transformación de relaciones cíclicas:** De forma general, si tenemos una entidad sobre la que existe una relación cíclica, para eliminar dicha relación, se crea una nueva entidad cuya clave estará formada por dos atributos, que contendrán las claves de las ocurrencias relacionadas. Entre ambas entidades se establecen dos relaciones, cuya cardinalidad dependerá de la cardinalidad que tuviera la relación cíclica en un principio.



- ✓ **Transformación de relaciones ternarias:** El tratamiento de las relaciones ternarias es similar al realizado para atributos asociados a relaciones, ya que una relación ternaria puede considerarse como una relación binaria a a la que se le asocia una entidad. Por consiguiente, si en lugar de ser un conjunto de atributos los asociados a la relación es una entidad, se asociaría ésta mediante una nueva



relación a la entidad resultante de eliminar la relación binaria.

- ✓ **Transformación de entidades débiles en fuertes:** Para esta transformación sólo es necesario añadir a la entidad débil los atributos clave de la entidad que hace posible la identificación de las ocurrencias. La clave de esta nueva entidad fuerte estará formada por los atributos clave de la que fuera entidad débil más los atributos adicionales.

Sea la entidad ALUMNADO que participa en la relación COLABORA con otra entidad llamada GRUPO_TRABAJO. Un alumno o alumna puede colaborar en varios grupos de trabajo simultáneamente y, a su vez, en un grupo de trabajo pueden colaborar un número indeterminado de alumnos. Se necesita registrar los días en los que el alumnado colabora con cada grupo de trabajo, para ello se asocia a la relación COLABORA un atributo denominado fecha_colaboración. Este atributo registrará en qué fecha un determinado alumno/a ha colaborado en un determinado grupo de trabajo.
¿Si tuvieras que hacer la transformación de esta parte del esquema conceptual para eliminar la relación M a N COLABORA, dónde colocarías el atributo fecha_colaboración?

- ○ En la entidad ALUMNADO, ya que en esta entidad es donde se almacenan todos los datos asociados al alumnado. Si consultamos el alumno o alumna, sabremos cuándo a colaborado en un grupo
- ◉ **En una nueva entidad que es combinación de ALUMNADO y GRUPO_TRABAJO, a la que podríamos llamar ALUMNADO_GRUPO**
- ○ En la entidad GRUPO_TRABAJO

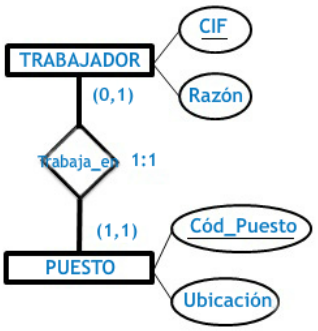
al transformar la relación M a N, se crean dos relaciones 1 a N entre ALUMNADO-ALUMNADO_GRUPO Y GRUPO_TRABAJO-ALUMNADO_GRUPO, siendo ALUMNADO_GRUPO una nueva entidad que tendrá por claves las claves primarias de ALUMNADO y GRUPO_TRABAJO, recibiendo como atributo el atributo que estaba asociado a la relación COLABORA. Para cada par ALUMNADO/GRUPO_TRABAJO tendremos registrado cuándo se realizó la colaboración.

10.- Paso del diagrama E/R al Modelo Relacional.

Si se ha llevado a cabo el proceso preparatorio de nuestro esquema conceptual o diagrama E/R, según se ha indicado en epígrafes anteriores, dispondremos de un Esquema Conceptual Modificado (ECM) en el que sólo existirán exclusivamente entidades fuertes con sus atributos y relaciones jerárquicas (1 a N). Pues bien, la aplicación del modelo de datos Relacional es automática, para ello se deben tener en cuenta las siguientes cuestiones:

- ✓ Toda entidad se transforma en una tabla.
- ✓ Todo atributo se transforma en columna dentro de una tabla.
- ✓ El atributo clave de la entidad se convierte en clave primaria de la tabla y se representará subrayado en la tabla.
- ✓ Cada entidad débil generará una tabla que incluirá todos sus atributos, añadiéndose a ésta los atributos que son clave primaria de la entidad fuerte con la que esté relacionada. Estos atributos añadidos se constituyen como clave foránea que referencia a la entidad fuerte. Seguidamente, se escogerá una clave primaria para la tabla creada.
- ✓ Las relaciones Uno a Uno podrán generar una nueva tabla o propagar la clave en función de la cardinalidad de las entidades.

Paso de relaciones 1 a 1 del Esquema E/R al Esquema Relacional



Se ha de tener en cuenta las cadinalidades de las entidades que intervienen. Existen dos soluciones:

Si las entidades poseen cardinalidades (0,1), la relación se convertirá en una tabla.
Si una de las entidades posee cardinalidad (0,1) y la otra (1,1), se propagará la clave de la entidad con cardinalidad (1,1) a la tabla resultante de la entidad de cardinalidad (0,1).
Si ambas entidades tienen cadinalidad (1,1) se puede propagar la clave de cualquiera de ellas a la tabla de la otra entidad.

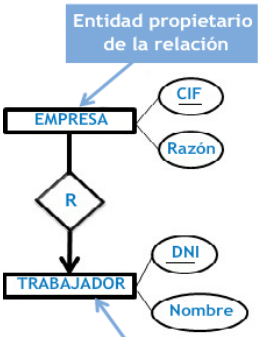
Nuestro ejemplo quedaría:

PUESTO(Cód_Puesto, Ubicación, DNI)
TRABAJADOR(DNI, Nombre)

DNI quedaría como clave foránea que referencia a la tabla TRABAJADOR

- ✓ Las relaciones Uno a Muchos podrán generar una nueva tabla o propagar la clave.

Paso de relaciones 1 a N del Esquema E/R al Esquema Relacional



Como norma general:

a) Se convierten a tablas las dos entidades
b) Se propaga la clave de la entidad propietario a la entidad miembro

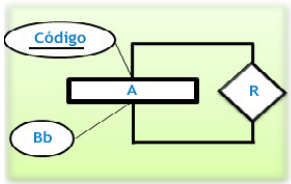
El paso a tablas quedaría:

EMPRESA(CIF, Razón)
TRABAJADOR(DNI,Nombre,CIF)

CIF quedaría como clave foránea que referencia a la tabla EMPRESA

- ✓ Las relaciones reflexivas o cíclicas podrán generar una o varias tablas en función de la cardinalidad de la relación.

Paso de relaciones Reflexivas del Esquema E/R al Esquema Relacional



Para este tipo de relaciones hemos de tener muy en cuenta la cardinalidad. Podremos tener los siguientes casos:

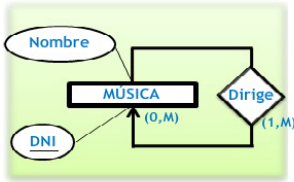
a) Si la relación es de 1 a 1, la clave de la entidad se repetirá, aunque debe ser con identificadores diferentes. Un identificador actuará como clave primaria y el otro, como clave foránea de ella misma

La clave primaria Código pasa a tener dos identificadores diferentes. Uno de ellos actuará como clave primaria y el otro como foránea

Para nuestro ejemplo, el paso a tablas quedaría:

A(Código1, Código2, Bb)

Paso de relaciones Reflexivas del Esquema E/R al Esquema Relacional



b) Si la relación es 1 a M, podremos tener dos casos:

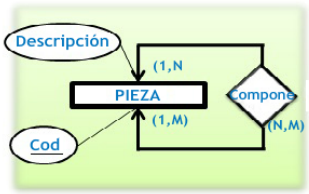
- Si la entidad muchos es siempre obligatoria, actuaremos como para el caso 1 a 1
- Si la entidad muchos no es obligatoria, se creará una nueva tabla cuya clave será la de la entidad del lado muchos, y se propaga la clave hacia esta nueva tabla como clave foránea

Para nuestro ejemplo, el paso a tablas quedaría:

MÚSICO(DNI, Nombre)
DIRIGE(DNI, DNI_Director)

En la entidad DIRIGE, se añade de nuevo el DNI pero jugando el rol de director, siendo clave foránea de la entidad MÚSICO

Paso de relaciones Reflexivas del Esquema E/R al Esquema Relacional



c) Si la relación es M a N, se trataría como una relación binaria entre dos entidades. Se crearía una nueva tabla que tendrá dos veces la clave primaria de la entidad. Si hubiese atributos asociados a la relación, se asociarían a la nueva tabla. La clave de esta tabla será la combinación de ambas claves primarias.

Para nuestro ejemplo, el paso a tablas quedaría:

PIEZA(Cod, Descripción)
PIEZA_COMPUESTA(Cod,Cod_com ponente)

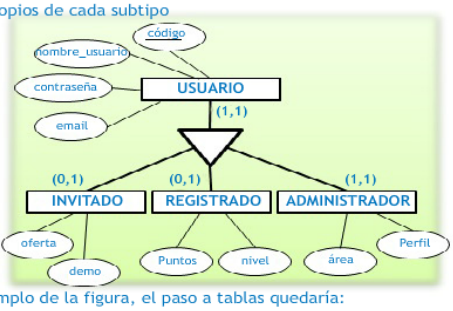
En la nueva tabla PIEZA_COMPUESTA aparecerá dos veces la clave primaria de PIEZA, pero jugando dos roles. Por un lado el de clave de la pieza en cuestión y, por otro, el de la/s pieza/s que la componen

✓ Las jerarquías generarán la reunión, eliminación o creación de relaciones 1 a 1.

Paso de Jerarquías del Esquema E/R al Esquema Relacional

Existen tres formas de tratar las relaciones jerárquicas, son las siguientes:

c) Crear una única entidad que aglutine todos los subtipos. Esta nueva entidad tendrá todos los atributos del supertipo y de los subtipos. Esta unión permite una mayor simplicidad, aunque puede provocar valores nulos en atributos propios de cada subtipo

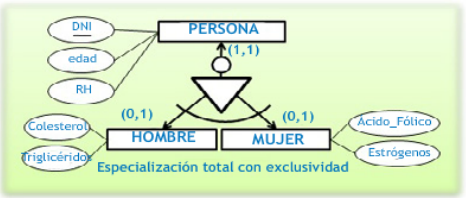


Para el ejemplo de la figura, el paso a tablas quedaría:

USUARIO(Código, nombre_usuario, contraseña, email, oferta, demo, Puntos, nivel, área, Perfil)

Paso de Jerarquías del Esquema E/R al Esquema Relacional

b) Anulación del supertipo. Al suprimir el supertipo, sus atributos pasan directamente a todos los subtipos y las relaciones del supertipo se han de reproducir en cada uno de los subtipos. La clave del supertipo, pasa a los subtipos. Este tratamiento suele aplicarse en jerarquías totales y exclusivas

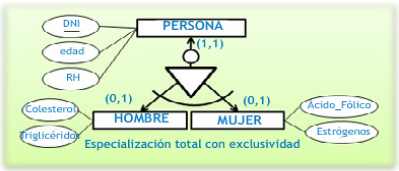


Para el ejemplo de la figura, el paso a tablas quedaría:

HOMBRE(DNI, edad, RH, Colesterol, Triglicéridos)
MUJER (DNI, edad, RH, Ácido_Fólico, Estrógenos)

Paso de Jerarquías del Esquema E/R al Esquema Relacional

c) Añadir relaciones 1 a 1 entre el supertipo y los subtipos. Los atributos del supertipo se mantendrán y cada uno de los subtipos tendrá una clave foránea proveniente del supertipo, con la que podrán identificarse. El supertipo se relaciona con los subtipos mediante relaciones 1 a 1



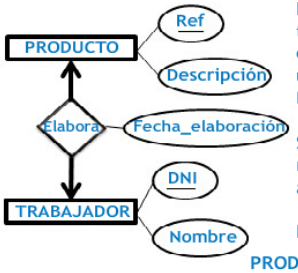
Para el ejemplo de la figura, el paso a tablas quedaría:

PERSONA (DNI, edad, RH)
HOMBRE(DNI, Colesterol, Triglicéridos)
MUJER (DNI, Ácido_Fólico, Estrógenos)

✓ Las relaciones Muchos a Muchos se transforman en una tabla que tendrá como clave primaria las claves primarias de las entidades que asocia.

No obstante, si en el proceso de generación del diagrama E/R o esquema conceptual hemos aplicado correctamente las reglas de simplificación de diagramas, nuestro Esquema Conceptual Modificado nos permitirá el paso a tablas teniendo en cuenta sólo las transformaciones asociadas a entidades, relaciones 1 a N, 1 a 1 y Jerarquías.

Paso de relaciones M a N del Esquema E/R al Esquema Relacional



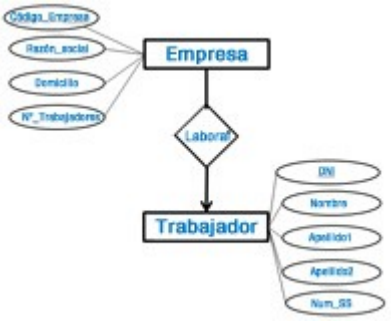
Las relaciones M a N se transforman en una nueva tabla que tendrá como clave primaria la unión de las claves primarias de las entidades que asocia

Si hubiese atributos asociados a la relación, éstos se incluirían como atributos de la nueva tabla

Nuestro ejemplo quedaría:

PRODUCTO(Ref, Descripción)
TRABAJADOR(DNI, Nombre)
ELABORA(Ref, DNI,Fecha_elaboración)

Las claves de las entidades relacionadas, quedan en la nueva tabla



Ejercicio resuelto

Sea la siguiente representación a través del modelo E/R de una relación entre dos entidades, obtén el paso a tablas de dicho esquema:

Respuesta:

El paso a tablas de dicho esquema sería el siguiente:

```
EMPRESA (Código empresa, razón_social, domicilio, N°_Trabajadores)
TRABAJADOR(DNI, Nombre, Apellido1, Apellido2, Num_SS)
```

Para materializar la relación de uno a muchos LABORAL, se incluye una clave foránea en la entidad TRABAJADOR, que referencia a la entidad EMPRESA, quedando:

```
EMPRESA (Código empresa, razón_social, domicilio, N°_Trabajadores)
TRABAJADOR(DNI, Nombre, Apellido1, Apellido2, Num_SS, Código_empresa)
```

11.- Normalización de modelos relacionales.

Caso práctico

*En estos primeros desarrollos **Ada** debe estar muy pendiente del trabajo que están realizando **Juan** y **María.** El proceso de transformación del Esquema Conceptual Modificado al modelo Relacional, requiere cierta experiencia y concentración. Dada su importancia y dificultad, este paso deben llevarlo a cabo de manera tranquila y comentando en grupo las diferentes operaciones que van a ir realizando.*

¿Crees que tu base de datos ya podría construirse directamente sobre el SGBD relacional que hayas elegido? La respuesta podría ser afirmativa, pero si queremos que nuestra base de datos funcione con plena fiabilidad, es necesario antes llevar a cabo un proceso de normalización de las tablas que la componen.

¿Y qué es eso de la normalización?

Normalización: Proceso que consiste en imponer a las tablas del modelo Relacional una serie de restricciones a través de un conjunto de transformaciones consecutivas. Este proceso garantizará que las tablas contienen los atributos necesarios y suficientes para describir la realidad de la entidad que representan, permitiendo separar aquellos atributos que por su contenido podrían generar la creación de otra tabla.

A veces uno se pregunta ¿Quién habrá sido el ideante de estos conceptos? En el siguiente enlace que te proponemos, puedes conocer quién fue.

http://es.wikipedia.org/wiki/Edgar_Frank_Codd

A principios de la década de los setenta, concretamente en 1972, Codd establece una técnica para llevar a cabo el diseño de la estructura lógica de los datos representados a través del modelo relacional, a la que denominó **normalización**. Pero esta técnica no ha de utilizarse para el diseño de la base de datos, sino como un proceso de refinamiento que debe aplicarse después de lo que conocemos como "paso a tablas", o lo que formalmente se denomina traducción del esquema conceptual al esquema lógico. Este proceso de refinamiento conseguirá los siguientes objetivos:

- ✓ Suprimir dependencias erróneas entre atributos.
- ✓ Optimizar los procesos de inserción, modificación y borrado en la base de datos.

El proceso de normalización se basa en el análisis de las dependencias entre atributos. Para ello tendrá en cuenta los conceptos de: **dependencia funcional, dependencia funcional completa y dependencia transitiva**. Estos conceptos se desarrollan seguidamente.

¿Y cómo se aplica la normalización? Es un proceso que se realiza en varias etapas secuenciales. Cada etapa está asociada a una **forma normal**, que establece unos requisitos a cumplir por la tabla sobre la que se aplica. Existen varias formas normales: **Primera, Segunda, Tercera, Boyce-Codd, Cuarta, Quinta y Dominio-Clave**. Como hemos indicado, el paso de una forma normal a otra es consecutivo, si no se satisface una determinada forma normal no puede pasarse al análisis de la siguiente. Según vamos avanzando en la normalización, los requisitos a cumplir serán cada vez más restrictivos, lo que hará que nuestro esquema relacional sea cada vez más robusto.

Como norma general, para garantizar que no existan problemas en la actualización de datos, **es recomendable aplicar el proceso de normalización hasta Tercera Forma Normal o incluso hasta Forma Normal de Boyce-Codd**. En los siguientes epígrafes se describen las características y requisitos de cada una de las formas normales.

11.1.- Tipos de dependencias.

Vamos a desarrollar aquí los conceptos sobre los que se basa el análisis de dependencias entre atributos, que se lleva a cabo en el proceso de normalización antes indicado, son los siguientes:

- ✓ **Dependencia Funcional:** Dados los atributos A y B, se dice que B depende funcionalmente de A, sí, y solo sí, para cada valor de A sólo puede existir un valor de B. La dependencia funcional siempre se establece entre atributos de una misma tabla. El atributo A se denomina determinante, ya que A determina el valor de B. Para representar esta dependencia funcional utilizamos la siguiente notación: **A → B**. Hay que indicar que A y B podrían ser un solo atributo o un conjunto de ellos.
- ✓ **Dependencia Funcional Completa:** Dados los atributos A1, A2, ...Ak y B, se dice que B depende funcionalmente de forma completa de A1, A2, ...Ak, si y solo si B depende funcionalmente del conjunto de atributos A1, A2, ...Ak, pero no de ninguno de sus posibles subconjuntos.
- ✓ **Dependencia Transitiva:** Dados tres atributos A, B y C, se dice que existe una dependencia transitiva entre A y C, si B depende funcionalmente de A y C depende funcionalmente de B. A, B y C podrían ser un solo atributo o un conjunto de ellos.

Para ilustrar los tipos de dependencias descritas, analiza el siguiente ejercicio resuelto.

Ejercicio resuelto

Dadas las siguientes tablas:

```
EMPLEADO( DNI, Nombre, Dirección, Localidad, Cod_Localidad, Nombre_hijo, Edad_hijo)
LIBRO (Título_libro, Num_ejemplar, Autor, Editorial, Precio)
```

Resuelve las siguientes cuestiones:

a. Indica qué atributos presentan una dependencia funcional de la clave primaria de la tabla EMPLEADO.
b. Indica qué atributos presentan una dependencia funcional completa en la tabla LIBRO.
c. Indica qué atributos presentan una dependencia transitiva en la tabla EMPLEADO.

Resultado:

Apartado a)

Los atributos Nombre, y Dirección dependen funcionalmente de DNI, ya que para un DNI específico sólo podrá haber un nombre y una dirección. Pero los atributos Nombre_hijo y Edad_hijo no presentan esa dependencia funcional de DNI, ya que para un DNI específico podríamos tener varios valores diferentes en esos atributos. (Consideraremos para este ejemplo que todos los empleados registrados en esta base de datos tienen nombres distintos). Expresemos estas dependencias funcionales mediante su notación:

```
DNI → Nombre
DNI → Dirección
```

Apartado b)

Los atributos Editorial y Precio dependen funcionalmente del conjunto de atributos que forman la clave primaria de la tabla, pero no dependen de Título_libro o de Num_ejemplar por separado, por lo que presentan una dependencia funcional completa de la clave. El atributo Autor depende funcionalmente sólo y exclusivamente de Titulo_libro, por lo que no presenta una dependencia funcional completa de los atributos que forman la clave.

Apartado c)

Los atributos Cod_Localidad y Localidad dependen funcionalmente de DNI, pero entre Cod_Localidad y Localidad existe otra dependencia funcional. Por tanto, se establece que Localidad depende funcionalmente de Cod_Localidad, y a su vez, Cod_Localidad depende funcionalmente de DNI. Con lo que podemos afirmar que existe una dependencia transitiva entre Localidad y DNI. Si lo representamos con la notación asociada a las dependencias funcionales, quedaría:

```
DNI → Cod_Localidad → Localidad.
```

11.2.- Formas Normales.

Una vez conocidos los conceptos sobre los que se basa el proceso de normalización, se han de llevar a cabo una serie de etapas consecutivas en las que se aplicarán las propiedades de cada una de las formas normales definidas por Codd. A continuación se exponen los requisitos a cumplir por las tablas de nuestra base de datos según la forma normal que apliquemos.

1ª Forma Normal

Una tabla está en Primera Forma Normal (1FN o FN1) sí, y sólo sí, todos los atributos de la misma contienen valores atómicos, es decir, no hay grupos repetitivos. Dicho de otra forma, estará en 1FN si los atributos no clave, dependen funcionalmente de la clave. ¿Cómo se normaliza a Primera Forma Normal?

a. Se crea, a partir de la tabla inicial, una nueva tabla cuyos atributos son los que presentan dependencia funcional de la clave primaria. La clave de esta tabla será la misma clave primaria de la tabla inicial. Esta tabla ya estará en 1FN.
b. Con los atributos restantes se crea otra tabla y se elije entre ellos uno que será la clave primaria de dicha tabla. Comprobaremos si esta segunda tabla está en 1FN. Si es así, la tabla inicial ya estará normalizada a 1FN y el proceso termina. Si no está en 1FN, tomaremos la segunda tabla como tabla inicial y repetiremos el proceso.

2ª Forma Normal

Una tabla está en Segunda Forma Normal (2FN o FN2) sí, y sólo sí, está en 1FN y, además, todos los atributos que no pertenecen a la clave dependen funcionalmente de forma completa de ella. Es obvio que una tabla que esté en 1FN y cuya clave esté compuesta por un único atributo, estará en 2FN. ¿Cómo se normaliza a Segunda Forma Normal?

a. Se crea, a partir de la tabla inicial, una nueva tabla con los atributos que dependen funcionalmente de forma completa de la clave. La clave de esta tabla será la misma clave primaria de la tabla inicial. Esta tabla ya estará en 2FN.
b. Con los atributos restantes, se crea otra tabla que tendrá por clave el subconjunto de atributos de la clave inicial de los que dependen de forma completa. Se comprueba si esta tabla está en 2FN. Si es así, la tabla inicial ya está normalizada y el proceso termina. Si no está en 2FN, tomamos esta segunda tabla como tabla inicial y repetiremos el proceso.

3ª Forma Normal

Una tabla está en Tercera Forma Normal (3FN o FN3) sí, y sólo sí, está en 2FN y, además, cada atributo que no está en la clave primaria no depende transitivamente de la clave primaria. ¿Cómo se normaliza a Tercera Forma Normal?

a. Se crea, a partir de la tabla inicial, una nueva tabla con los atributos que no poseen dependencias transitivas de la clave primaria. Esta tabla ya estará en 3FN.
b. Con los atributos restantes, se crea otra tabla con los dos atributos no clave que intervienen en la dependencia transitiva, y se elije uno de ellos como clave primaria, si cumple los requisitos para ello. Se comprueba si esta tabla está en 3FN. Si es así, la tabla inicial ya está

normalizada y el proceso termina. Si no está en 3FN, tomamos esta segunda tabla como tabla inicial y repetiremos el proceso.

Forma Normal de Boyce Codd

Una tabla está en Forma Normal de Boyce-Codd (FNBC o BCFN) sí, y sólo sí, está en 3FN y todo determinante es una clave candidata. Un determinante será todo atributo simple o compuesto del que depende funcionalmente de forma completa algún otro atributo de la tabla. Aquellas tablas en la que todos sus atributos forman parte de la clave primaria, estarán en FNBC. Por tanto, si encontramos un determinante que no es clave candidata, la tabla no estará en FNBC. Esta redundancia suele ocurrir por una mala elección de la clave. Para normalizar a FNBC tendremos que descomponer la tabla inicial en dos, siendo cuidadosos para evitar la pérdida de información en dicha descomposición.

Otras formas normales

Existen también la Cuarta Forma Normal (4FN o FN4), Quinta Forma Normal (5FN o FN5) y Forma Normal de Dominio-Clave (DKFN), aunque se ha recomendado normalizar hasta 3FN o FNBC. La 4FN se basa en el concepto de Dependencias Multivaluadas, la 5FN en las Dependencias de Join o de reunión y la DKFN en las restricciones impuestas sobre los dominios y las claves.

Si deseas conocer cuáles son las propiedades y requisitos a cumplir establecidos en las formas normales 4ª, 5ª y DKFN, te proponemos los siguientes enlaces:
http://conclase.net/mysql/curso/?cap=004c#NOR_4FN
http://es.wikipedia.org/wiki/5NF
http://es.wikipedia.org/wiki/DKNF

Ejercicio resuelto

Sea la siguiente tabla:

```
COMPRAS (cod_compra, cod_prod, nomb_prod, fecha, cantidad, precio, fecha_rec, cod_prov, nomb_prov, tfno).
```

Se pide normalizarla hasta FNBC.

Resultado:

Comprobamos 1FN:

La tabla COMPRAS está en 1FN ya que todos sus atributos son atómicos y todos los atributos no clave dependen funcionalmente de la clave.

Comprobamos 2FN:

Nos preguntaremos ¿Todo atributo depende de todo el conjunto de atributos que forman la clave primaria, o sólo de parte?. Como vemos, existen atributos que dependen sólo de una parte de la clave, por lo que esta tabla no está en 2FN.

Veamos las dependencias:

```
cod_prod → nomb_prod, y cod_prod es parte de la clave primaria.
```

Al no estar en 2FN, hemos de descomponer la tabla COMPRAS en:

```
COMPRA1 (cod_compra, cod_prod, fecha, cantidad, precio, fecha_rec, cod_prov, nomb_prov, tfno).
```

```
PRODUCTO (cod_prod, nomb_prod).
```

Una vez hecha esta descomposición, ambas tablas están en 2FN. Todos los atributos no clave dependen de toda la clave primaria.

Comprobamos 3FN:
PRODUCTO está en 3FN, ya que por el número de atributos que tiene no puede tener dependencias transitivas.
¿COMPRA1 está en 3FN? Hemos de preguntarnos si existen dependencias transitivas entre atributos no clave.

Veamos las dependencias:

```
cod_prov → nomb_prov
cod_prov → tfno
```

(siendo cod_prov el código del proveedor y nomb_prov el nombre del proveedor)

COMPRA1 no está en 3FN porque existen dependencias transitivas entre atributos no clave, por tanto hemos de descomponer:

```
COMPRA2 (cod_compra, cod_prod, fecha, cantidad, precio, fecha_rec, cod_prov)
PROVEEDOR (cod_prov, nomb_prov, tfno)
```

Comprobamos FNBC:

PRODUCTO está en FNBC, ya que está en 3FN y todo determinante es clave candidata.
COMPRA2 está en FNBC, ya que está en 3FN y todo determinante es clave candidata.
PROVEEDOR está en FNBC, ya que está en 3FN y todo determinante es clave candidata.

La tabla inicial COMPRAS queda normalizada hasta FNBC del siguiente modo:

```
PRODUCTO (cod_prod, nomb_prod)
COMPRA2 (cod_compra, cod_prod, fecha, cantidad, precio, fecha_rec, cod_prov)
PROVEEDOR (cod_prov, nomb_prov, tfno)
```

TEMA 4

INDICE

1.- Introducción. 3
2.- La sentencia SELECT. 5
2.1.- Cláusula SELECT. 5
Ejercicio resuelto 6
2.2.- Cláusula FROM. 9
2.3.- Cláusula WHERE. 9
2.4.- Ordenación de registros. Cláusula ORDER BY. 10
Ejercicio resuelto 11
3.- Operadores. 12
3.1.- Operadores de comparación. 12
Ejercicio resuelto 13
3.2.- Operadores aritméticos y de concatenación. 13
3.3.- Operadores lógicos. 14
Ejercicio resuelto 14
3.4.- Precedencia. 15
4.- Consultas calculadas. 16
5.- Funciones. 17
5.1.- Funciones numéricas. 17
5.2.- Funciones de cadena de caracteres. 18
5.3.- Funciones de manejo de fechas. 19
5.4.- Funciones de conversión. 20
5.5.- Otras funciones: NVL y DECODE. 21
6.- Consultas de resumen. 23
6.1.- Funciones de agregado: SUM y COUNT. 24
Ejercicio resuelto 24
6.2.- Funciones de agregado: MIN y MAX. 24
6.3.- Funciones de agregado: AVG, VAR, STDEV y STDEVP. 25
Ejercicio resuelto 25
7.- Agrupamiento de registros. 26
8.- Consultas multitablas. 28
8.1.- Composiciones internas. 28
Ejercicio resuelto 29
Ejercicio resuelto 30
8.2.- Composiciones externas. 30
Ejercicio resuelto 30
8.3.- Composiciones en la versión SQL99. 31
9.- Otras consultas multitablas: Unión, Intersección y diferencia de consultas. 33
10.- Subconsultas. 35
Varios ejercicios SQL resueltos 37
La tienda de informática 37
Empleados 39
Los Almacenes 42
Películas y Salas 43
Los Directores 44
Piezas y Proveedor 46
Los Científicos 47
Grandes Almacenes 48
Los investigadores 49

Realización de consultas.

Caso práctico

Una de las cosas más importantes que ofrece una base de datos es la opción de poder consultar los datos que guarda, por eso Ana y Juan van a intentar sacar el máximo partido a las tablas que han guardado y sobre ellas van a obtener toda aquella información que su cliente les ha solicitado. Sabemos que dependiendo de quién consulte la base de datos, se debe ofrecer un tipo de información u otra. Es por esto que deben crear distintas consultas y vistas.

Ana sabe que existen muchos tipos de operadores con los que puede "jugar" para crear consultas y también tiene la posibilidad de crear campos nuevos donde podrán hacer cálculos e incluso trabajar con varias tablas relacionadas a la vez.

*Actualmente están con una base de datos en la que se ha almacenado información sobre los **empleados** de la empresa que tiene la página de juegos online, los **departamentos** en los que trabajan y los **estudios** de sus empleados. Se está guardando el **historial laboral y salarial** de todos los empleados. Ya que tienen una base de datos para sus clientes, han visto que también sería conveniente tener registrada esta otra información interna de la empresa.*

De este modo pueden llevar un control más exhaustivo de sus empleados, salario y especialización. Podrán conocer cuánto pagan en sueldos, qué departamento es el que posee mayor número de empleados, el salario medio, etc. Para obtener esta información necesitarán consultar la base utilizando principalmente el comando SELECT.

1.- Introducción.

Caso práctico

Juan quiere comenzar con consultas básicas a los datos, cosas bastante concretas y sencillas de manera que se obtenga información relevante de cada una de las tablas. También quieren realizar algunos cálculos como conocer el salario medio de cada empleado, o el mayor salario de cada departamento, o saber cuánto tiempo lleva cada empleado en la empresa.

En unidades anteriores has aprendido que SQL es un conjunto de sentencias u órdenes que se necesitan para acceder a los datos. Este lenguaje es utilizado por la mayoría de las aplicaciones donde se trabaja con datos para acceder a ellos. Es decir, es la vía de comunicación entre el usuario y la base de datos.

SQL nació a partir de la publicación "A relational model of data for large shared data banks" de Edgar Frank

Codd. IBM aprovechó el modelo que planteaba Codd para desarrollar un lenguaje acorde con el recién nacido modelo relacional, a este primer lenguaje se le llamó SEQUEL (Structured English QUEry Language). Con el tiempo SEQUEL se convirtió en SQL (Structured Query Language). En 1979, la empresa Relational Software sacó al mercado la primera implementación comercial de SQL. Esa empresa es la que hoy conocemos como Oracle.

Actualmente SQL sigue siendo el estándar en lenguajes de acceso a base de datos relacionales.

En 1992, ANSI e ISO completaron la estandarización de SQL y se definieron las sentencias básicas que debía contemplar SQL para que fuera estándar. A este SQL se le denominó ANSI-SQL o SQL92.

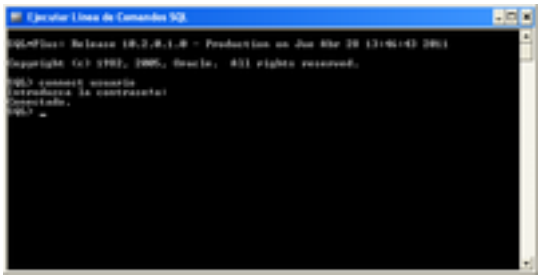
Hoy en día todas las bases de datos comerciales cumplen con este estándar, eso sí, cada fabricante añade sus mejoras al lenguaje SQL.

La primera fase del trabajo con cualquier base de datos comienza con sentencias DDL (en español Lenguaje de Definición de Datos), puesto que antes de poder almacenar y recuperar información debimos definir las estructuras donde agrupar la información: las tablas.

La siguiente fase será manipular los datos, es decir, trabajar con sentencias DML (en español Lenguaje de Manipulación de Datos). Este conjunto de sentencias está orientado a consultas y manejo de datos de los objetos creados. Básicamente consta de cuatro sentencias: SELECT, INSERT, DELETE y UPDATE. En esta unidad nos centraremos en una de ellas, que es la sentencia para consultas: SELECT.

Las sentencias SQL que se verán a continuación pueden ser ejecutadas desde el entorno web **Application Express** de Oracle utilizando el botón **SQL** en la página de inicio, y desplegando su lista desplegable elegir **Comandos SQL > Introducir Comando**.

También se pueden indicar las sentencias SQL desde el entorno de **SQL*Plus** que ofrece Oracle y que puedes encontrar en **Inicio > Todos los programas > Base de Datos Oracle Express Edition > Ejecutar Línea de Comandos SQL.**



Si optas por abrir esa aplicación (**Ejecutar Línea de Comandos SQL**), el primer paso que debe realizarse para manipular los datos de una determinada tabla, es conectarse utilizando un nombre de usuario con los permisos necesarios para hacer ese tipo de operaciones a la tabla deseada. Utiliza para ello la orden CONNECT seguida del nombre de usuario. Posteriormente, solicitará la contraseña correspondiente a dicho usuario.

Para ejecutar cualquiera de las sentencias SQL que aprenderás en los siguientes puntos, simplemente debes escribirla completa y pulsar **Intro** para que se inicie su ejecución.

¿Con qué sentencias se definen las estructuras donde agrupar la información, es decir, las tablas?

- ○ DML
- ◉ **DDL**
- ○ DCL

Así es, dentro del lenguaje de definición de datos está la creación de tablas.

2.- La sentencia SELECT.

Caso práctico

__Ana__ está trabajando con la tabla Partidas, de aquí quiere ver qué información es la más importante, para así crear las consultas más sencillas pero a la vez más frecuentes. Sabe que con SQL y utilizando el comando SELECT puede sacar provecho a los datos contenidos en una tabla.

¿Cómo podemos seleccionar los datos que nos interesen dentro de una base de datos? Para recuperar o seleccionar los datos, de una o varias tablas puedes valerte del lenguaje SQL, para ello utilizarás la sentencia SELECT, que consta de cuatro partes básicas:

- ✓ **Cláusula** SELECT seguida de la descripción de lo que se desea ver, es decir, de los nombres de las columnas que quieres que se muestren separadas por comas simples (" , "). Esta parte es obligatoria.
- ✓ **Cláusula** FROM seguida del nombre de las tablas de las que proceden las columnas de arriba, es decir, de donde vas a extraer los datos. Esta parte también es obligatoria.
- ✓ **Cláusula** WHERE seguida de un criterio de selección o condición. Esta parte es opcional.
- ✓ **Cláusula** ORDER BY seguida por un criterio de ordenación. Esta parte también es opcional.

Por tanto, una primera sintaxis quedaría de la siguiente forma:

```
SELECT [ALL | DISTINCT] columna1, columna2, ... FROM tabla1, tabla2, ... WHERE condición1, condición2, ... ORDER BY ordenación;
```

Las cláusulas ALL y DISTINCT son opcionales.

- ✓ **Si incluyes la cláusula ALL después de SELECT, indicarás que quieres seleccionar todas las filas estén o no repetidas. Es el valor por defecto y no se suele especificar.**
- ✓ **Si incluyes la cláusula DISTINCT después de SELECT, se suprimirán aquellas filas del resultado que tengan igual valor que otras.**

¿Qué se debe indicar a continuación de la cláusula FROM?

- ○ Las columnas que queremos seleccionar.
- ○ Los criterios con los que filtro la selección.
- ◉ **Las tablas de donde se van a extraer los datos.**
- ○ La ordenación ascendente.

Así es, Aparecerán todos los nombres de las tablas cuyas columnas estemos seleccionando, separadas por coma.

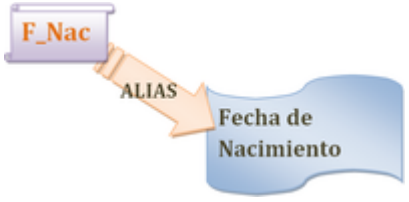
2.1.- Cláusula SELECT.

Ya has visto que a continuación de la sentencia SELECT debemos especificar cada una de las columnas que queremos seleccionar. Además, debemos tener en cuenta lo siguiente:

- ✓ **Se pueden nombrar** a las columnas anteponiendo el nombre de la tabla de la que proceden, pero esto es opcional y quedaría: NombreTabla.NombreColumna
- ✓ Si queremos **incluir todas las columnas** de una tabla podemos utilizar el comodín **asterisco** ("*"). Quedaría así: SELECT * FROM NombreTabla;
- ✓ También podemos **ponerle alias a los nombres** de las columnas. Cuando se consulta una base de datos, los nombres de las columnas se usan como cabeceras de presentación. Si éste resulta largo, corto o poco descriptivo, podemos usar un alias. Para ello a continuación del nombre de la columna ponemos entre comillas dobles el alias que demos a esa columna.

  Veamos un ejemplo:

```
SELECT F_Nacimiento "Fecha de Nacimiento" FROM USUARIOS;
```



- ✓ También podemos **sustituir el nombre** de las columnas por constantes, expresiones o funciones SQL. Un ejemplo:
- ✓ SELECT 4*3/100 "MiExpresión", Password FROM USUARIOS;

Si quieres conocer algo más sobre esta sentencia y ver algunos ejemplos del uso de SELECT aquí tienes el siguiente enlace:

http://www.devjoker.com/contenidos/Tutorial-SQL-/14/Consultar-datos-SELECT.aspx

Ejercicio resuelto

Si quieres practicar algunos ejercicios puedes ayudar a **Ana** con algunas consultas. Para ello te facilitamos las tablas que ha creado recientemente para la base de datos con la que actualmente están trabajando:

```
/* tabla empleados */
CREATE TABLE EMPLEADOS (
DNI NUMBER(8),
NOMBRE VARCHAR2(10) NOT NULL,
APELLIDO1 VARCHAR2(15) NOT NULL,
APELLIDO2 VARCHAR2(15),
SALARIO NUMBER(10,2),  /* podría ganar mucho */
DIRECC1 VARCHAR2(25),
DIRECC2 VARCHAR2(20),
CIUDAD VARCHAR2(20),
MUNICIPIO VARCHAR2(20),
COD_POSTAL VARCHAR2(5),
SEXO CHAR(1),
FECHA_NAC DATE,
CONSTRAINT PK EMPLEADOS PRIMARY KEY (DNI),
CONSTRAINT CK SEXO CHECK (SEXO IN ('H', 'M'))
);

/* tabla departamentos */
CREATE TABLE DEPARTAMENTOS (
DPTO COD NUMBER(5),
NOMBRE DPTO VARCHAR2(30) NOT NULL,
JEFE NUMBER(8),
PRESUPUESTO NUMBER(6) NOT NULL,
PRES_ACTUAL NUMBER(6),
CONSTRAINT PK_DEPARTAMENTOS PRIMARY KEY (DPTO_COD),
CONSTRAINT FK_DEPARTAMENTOS FOREIGN KEY (JEFE) REFERENCES EMPLEADOS (DNI)
);

/* Tabla universidades */
CREATE TABLE UNIVERSIDADES (
UNIV COD NUMBER(5),
NOMBRE UNIV VARCHAR2(25) NOT NULL,
CIUDAD VARCHAR2(20),
MUNICIPIO VARCHAR2(20),
COD_POSTAL VARCHAR2(5),
CONSTRAINT PK UNIVERSIDADES PRIMARY KEY (UNIV COD)
);
/* tabla trabajos */
CREATE TABLE TRABAJOS (
TRABAJO_COD NUMBER(5),
NOMBRE_TRAB VARCHAR2(20) NOT NULL UNIQUE,
SALARIO_MIN NUMBER(5) NOT NULL,
SALARIO MAX NUMBER(5) NOT NULL,
CONSTRAINT PK TRABAJOS PRIMARY KEY (TRABAJO COD)
);

/* tabla estudios */
CREATE TABLE ESTUDIOS (
EMPLEADO DNI NUMBER(8),
UNIVERSIDAD NUMBER(5),
AÑO NUMBER(4),
GRADO VARCHAR2(5),
ESPECIALIDAD VARCHAR2(20),
CONSTRAINT PK_ESTUDIOS PRIMARY KEY (EMPLEADO_DNI, AÑO, GRADO),
CONSTRAINT FK_ESTUDIOS_EMPLEADOS FOREIGN KEY (EMPLEADO_DNI) REFERENCES EMPLEADOS (DNI),
```

```
CONSTRAINT FK_ESTUDIOS_UNIVERSIDADES FOREIGN KEY (UNIVERSIDAD) REFERENCES UNIVERSIDADES
(UNIV_COD)
);

/* tabla historial_laboral */
CREATE TABLE HISTORIAL_LABORAL (
EMPLEADO_DNI NUMBER(8),
TRAB_COD NUMBER(5),
FECHA_INICIO DATE,
FECHA_FIN DATE,
DPTO_COD NUMBER(5),
SUPERVISOR_DNI NUMBER(8),
CONSTRAINT PK HISTORIAL LABORAL PRIMARY KEY (EMPLEADO DNI, FECHA INICIO),
CONSTRAINT FK HLABORAL EMPLEADOS FOREIGN KEY (EMPLEADO DNI) REFERENCES EMPLEADOS (DNI),
CONSTRAINT FK HLABORAL TRABAJOS FOREIGN KEY (TRAB COD) REFERENCES TRABAJOS (TRABAJO COD),
CONSTRAINT FK HLABORAL DEPARTAMENTOS FOREIGN KEY (DPTO COD) REFERENCES DEPARTAMENTOS
(DPTO_COD),
CONSTRAINT FK_HLABORAL_SUPERVISOR FOREIGN KEY (SUPERVISOR_DNI) REFERENCES EMPLEADOS (DNI),
CONSTRAINT CK HLABORAL FECHAS CHECK (FECHA FIN IS NULL OR FECHA INICIO < FECHA FIN)
);
/* tabla historial salarial */
CREATE TABLE HISTORIAL SALARIAL (
EMPLEADO_DNI NUMBER(8),
SALARIO NUMBER(5),
FECHA_COMIENZO DATE,
FECHA_FIN DATE,
CONSTRAINT PK HISTORIAL SALARIAL PRIMARY KEY (EMPLEADO DNI, FECHA COMIENZO),
CONSTRAINT FK_HISTORIAL_SALARIAL FOREIGN KEY (EMPLEADO_DNI) REFERENCES EMPLEADOS (DNI),
CONSTRAINT CK_FECHAS CHECK (FECHA_FIN IS NULL OR FECHA_COMIENZO < FECHA_FIN)
);

ALTER SESSION SET NLS DATE FORMAT='DD/MM/YYYY';
/* EMPLEADOS */
INSERT INTO EMPLEADOS VALUES( '12345','Jose', 'Mercé','López', '1500','C/Sol, 1', 'C/ Otra,
1', 'Cádiz','Cádiz', '11000', 'H', '5/01/74');
INSERT INTO EMPLEADOS VALUES( '22222','María', 'Rosal','Cózar','2000', '', '',
'Ubrique','Cádiz', '11600', 'M', '');
INSERT INTO EMPLEADOS VALUES( '33333','Pilar', 'Pérez','Rollán','1000', '', '', 'Cádiz','',
'11600', 'M', '2/8/73');

/* DEPARTAMENTOS */
INSERT INTO DEPARTAMENTOS VALUES( '001', 'INFORMÁTICA', '33333', 80000, 50000);

/* UNIVERSIDADES */
INSERT INTO UNIVERSIDADES VALUES('0001', 'UNED', 'MADRID', 'M', '41420');
INSERT INTO UNIVERSIDADES VALUES('0002', 'SEVILLA', 'SEVILLA', '', '55555');
INSERT INTO UNIVERSIDADES VALUES('0003', 'CÁDIZ', 'CÁDIZ', '', '11000');

/* TRABAJOS */
INSERT INTO TRABAJOS VALUES ('001', 'ADMINISTRATIVO', 900, 1000);
INSERT INTO TRABAJOS VALUES ('002', 'CONTABLE', 900, 1000);
INSERT INTO TRABAJOS VALUES ('003', 'INGENIERO TÉCNICO', 1000, 1200);
INSERT INTO TRABAJOS VALUES ('004', 'INGENIERO', 1200, 1800);

/* ESTUDIOS */
INSERT INTO ESTUDIOS VALUES( '12345', '0001', '1992', 'MED', 'ADMINISTRATIVO');
INSERT INTO ESTUDIOS VALUES( '22222', '0001', '1998', 'SUP', 'ING INFORMÁTICA');
INSERT INTO ESTUDIOS VALUES( '33333', '0002', '1997', 'SUP', 'LIC INFORMÁTICA');

/* HISTORIAL SALARIAL */
INSERT INTO HISTORIAL SALARIAL VALUES( '12345', 950, '5/01/2003', '');
INSERT INTO HISTORIAL_SALARIAL VALUES( '22222', 1000, '3/11/2004', '3/11/2005');
INSERT INTO HISTORIAL_SALARIAL VALUES( '22222', 1500, '3/11/2005', '');
INSERT INTO HISTORIAL_SALARIAL VALUES( '33333',1600, '15/01/2001', '');

/* HISTORIAL LABORAL */
INSERT INTO HISTORIAL LABORAL VALUES( '12345', '001', '5/01/2003','', '0001', '33333');
INSERT INTO HISTORIAL_LABORAL VALUES( '22222', '003', '3/11/2004', '3/11/2005', '001',
'33333');
INSERT INTO HISTORIAL LABORAL VALUES( '22222', '003', '3/11/2005', '', '001', '33333');
INSERT INTO HISTORIAL_LABORAL VALUES( '33333','004', '15/01/2001', '', '001', '33333');
```

También tienes algunos datos incluidos para probar las distintas consultas que crees. A partir de ahora nos referiremos a estos datos como tablas de la empresa JuegosCA.

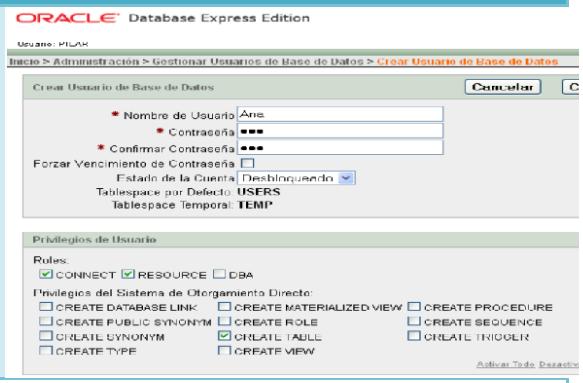
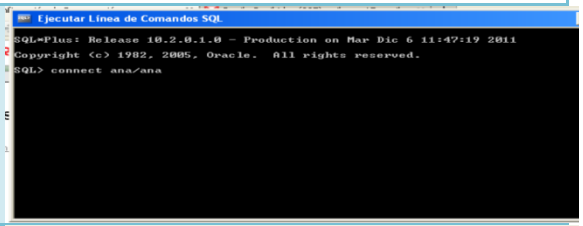
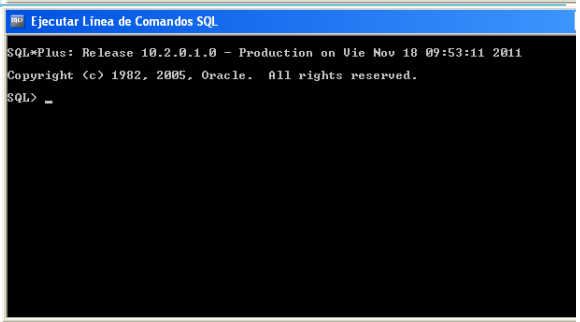
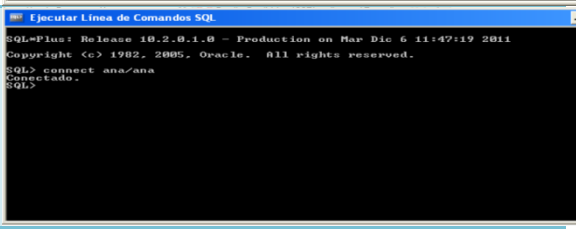
Por tanto lo primero que tienes que hacer es abrir el editor de SQL, para ello debes ir a Base de Datos de Oracle 10g Express y a continuación pulsar en Ejecutar Línea de Comandos SQL. Aparecerá una pantalla donde tienes que realizar los siguientes pasos:

1. Conectarte a través de un usuario.
2. Ejecutar el archivo que has bajado, para ello debes escribir :

```
@Ruta_donde_se_encuentra_el_archivo/BD04_CONT_R07_02.sql
```

En este ejercicio te pedimos que ejecutes el archivo y crees las tablas necesarias para poder realizar ejercicios posteriores.

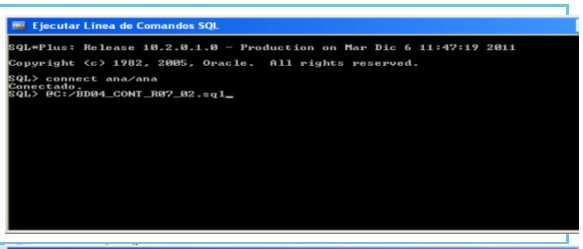
Resultado:
El resultado puedes verlo en la siguiente tabla.

Creación de tablas y preparación de Oracle.	
Para comenzar crearemos a partir de un usuario con privilegios el usuario ANA y pondremos una contraseña. Además, le daremos posibilidad de crear tablas.	
Desconectamos a este usuario, ya que queremos crear las tablas e insertar los datos en el nuevo usuario creado. Vamos a utilizar la línea de comandos de SQL para ejecutar el archivo descargado, para ello seguiremos los pasos que aparecen a continuación.	
Vamos a Base de Datos de Oracle 10g Express.	
Pulsamos en Ejecutar Línea de Comandos SQL. Aparecerá la siguiente pantalla	
Ejecutamos la instrucción: connect ana/ana. Cuando ejecutamos el comando debe decirnos que ya está conectado	

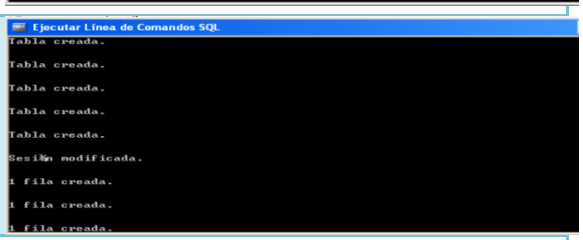
Ahora ya podemos ejecutar el archivo del siguiente modo:

```
@Ruta_donde_se_encuentra_el_archivo/BD04_CONT_R07_02.sql
```

En nuestro caso, el archivo está guardado directamente en la unidad C para que nos resulte más fácil localizarlo



Si todo es correcto, deberían crearse las tablas e insertarse los datos que contiene el archivo



A partir de aquí ya tienes un usuario con tablas y datos inlcuidos para poder practicar a la vez que Ana.

Puedes hacerlo a través de línea de comandos o entrando a entorno web **Application Express** de Oracle utilizando el botón **SQL** en la página de inicio, y desplegando su lista desplegable elegir **Comandos SQL** > **Introducir Comando**.

2.2.- Cláusula FROM.

Al realizar la consulta o selección has visto que puedes elegir las columnas que necesites, pero ¿de dónde extraigo la información?

En la sentencia `SELECT` debemos establecer de dónde se obtienen las columnas que vamos a seleccionar, para ello disponemos en la sintaxis de la cláusula `FROM`.

Por tanto, en la cláusula `FROM` **se definen los nombres de las tablas** de las que proceden las columnas. Si se utiliza más de una, éstas deben aparecer separadas por comas. A este tipo de consulta se denomina **consulta combinada** o `join`. Más adelante verás que para que la consulta combinada pueda realizarse, necesitaremos aplicar una condición de combinación a través de una cláusula `WHERE`.

También puedes añadir el nombre del usuario que es **propietario** de esas tablas, indicándolo de la siguiente manera:

```
USUARIO.TABLA
```

de este modo podemos distinguir entre las tablas de un usuario y otro (ya que esas tablas pueden tener el mismo nombre).

También puedes asociar un alias a las tablas para abreviar, en este caso **no es necesario que lo encierres entre comillas**.

Pongamos un ejemplo:

```
SELECT * FROM USUARIOS U;
```

2.3.- Cláusula WHERE.

¿Podríamos desear seleccionar los datos de una tabla que cumplan una determinada condición? Hasta ahora hemos podido ver la sentencia `SELECT` para obtener todas o un subconjunto de columnas de una o varias tablas. Pero esta selección afectaba a todas las filas (registros) de la tabla. Si queremos restringir esta selección a un subconjunto de filas debemos especificar una condición que

deben cumplir aquellos registros que queremos seleccionar. Para poder hacer esto vamos a utilizar la cláusula `WHERE`.

A continuación de la palabra `WHERE` será donde pongamos la condición que han de cumplir las filas para salir como resultado de dicha consulta.

El criterio de búsqueda o condición puede ser más o menos sencillo y para crearlo se pueden conjugar operadores de diversos tipos, funciones o expresiones más o menos complejas.
Si en nuestra tabla USUARIOS, necesitáramos un listado de los usuarios que son mujeres, bastaría con crear la siguiente consulta:

```
SELECT nombre, apellidos
FROM USUARIOS
WHERE sexo = 'M';
```

Más adelante te mostraremos los operadores con los que podrás crear condiciones de diverso tipo.

Aquí te adelantamos los operadores para que vayas conociéndolos. Con ellos trabajarás cuando hayas adquirido algunos conocimientos más:
http://www.desarrolloweb.com/articulos/1870.php

2.4.- Ordenación de registros. Cláusula ORDER BY.

En la consulta del ejemplo anterior hemos obtenido una lista de nombres y apellidos de las usuarias de nuestro juego. Sería conveniente que aparecieran ordenadas por apellidos, ya que siempre quedará más profesional además de más práctico. De este modo, si necesitáramos localizar un registro concreto la búsqueda sería más rápida. ¿Cómo lo haremos? Para ello usaremos la cláusula `ORDER BY`.

`ORDER BY` se utiliza para **especificar el criterio de ordenación** de la respuesta a nuestra consulta. Tendríamos:

```
SELECT [ALL | DISTINCT] columna1, columna2, ...
FROM tabla1, tabla2, ...
WHERE condición1, condición2, ...
ORDER BY columna1 [ASC | DESC], columna2 [ASC | DESC], …, columnaN [ASC | DESC];
```

Después de cada columna de ordenación se puede incluir el tipo de ordenación (ascendente o descendente) utilizando las palabras reservadas `ASC` o `DESC`. Por defecto, y si no se pone nada, la ordenación es ascendente.

Debes saber que es **posible ordenar por más de una columna**. Es más, puedes ordenar no solo por columnas sino a través de una expresión creada con columnas, una constante (aunque no tendría mucho sentido) o funciones SQL.

En el siguiente ejemplo, ordenamos por apellidos y en caso de empate por nombre:

```
SELECT nombre, apellidos
FROM USUARIOS
ORDER BY apellidos, nombre;
```

Puedes colocar el número de orden del campo por el que quieres que se ordene en lugar de su nombre, es decir, referenciar a los campos por su posición en la lista de selección. Por ejemplo, si queremos el resultado del ejemplo anterior ordenado por localidad:

```
SELECT nombre, apellidos, localidad
FROM usuarios
ORDER BY 3;
```

Si colocamos un número mayor a la cantidad de campos de la lista de selección, aparece un mensaje de error y la sentencia no se ejecuta.

¿Se puede utilizar cualquier tipo de datos para ordenar? No todos los tipos de campos te servirán para ordenar, únicamente aquellos de tipo **carácter, número o fecha**.

Relaciona cada cláusula de la sentencia SELECT con la información que debe seguirle:

Ejercicio de relacionar

Cláusula	Relación	Información que le sigue.
WHERE	**4**	1. Ordenación.
ORDER BY	**1**	2. Columnas.
FROM	**3**	3. Tablas.
SELECT	**2**	4. Condiciones.

Ejercicio resuelto

Utilizando las tablas y datos de la empresa **JuegosCA** descargados anteriormente, vamos a realizar una consulta donde obtengamos de la tabla ESTUDIOS, DNI de los empleados ordenados por Universidad descendente y año de manera ascendente.

Respuesta:

```
SELECT EMPLEADO_DNI
FROM ESTUDIOS
ORDER BY UNIVERSIDAD DESC, AÑO;
```

3.- Operadores.

Caso práctico

*En el proyecto en el que actualmente trabajan **Ana** y **Juan**, tendrán que realizar consultas que cumplan unos criterios concretos, por ejemplo, obtener el número de jugadores que tienen cierto número de créditos o aquellos que son mujeres e incluso conocer el número de usuarios que son de una provincia y además sean hombres.*

*Para poder realizar este tipo de consultas necesitaremos utilizar operadores que sirvan para crear las expresiones necesarias. **Ana** y **Juan** conocen los 4 tipos de operadores con los que se puede trabajar: relacionales, aritméticos, de concatenación y lógicos.*

Veíamos que en la cláusula `WHERE` podíamos incluir expresiones para filtrar el conjunto de datos que queríamos obtener. Para crear esas expresiones necesitas utilizar distintos operadores de modo que puedas comparar, utilizar la lógica o elegir en función de una suma, resta, etc.

Los operadores son símbolos que permiten realizar operaciones matemáticas, concatenar cadenas o hacer comparaciones.

Oracle reconoce 4 tipos de operadores:

1. Relacionales o de comparación.
2. Aritméticos.
3. De concatenación.
4. Lógicos.

¿Cómo se utilizan y para qué sirven? En los siguientes apartados responderemos a estas cuestiones.

Si quieres conocer un poco más sobre los operadores visita este enlace:
http://deletesql.com/viewtopic.php?f=5&t=10

3.1.- Operadores de comparación.

Los puedes conocer con otros nombres como **relacionales**, nos **permitirán comparar expresiones**, que pueden ser valores concretos de campos, variables, etc.

Los operadores de comparación son símbolos que se usan como su nombre indica para comparar dos valores. Si el resultado de la comparación es correcto la expresión considerada es verdadera, en caso contrario es falsa.

Tenemos los siguientes operadores y su operación:

Operadores y su significado.	
OPERADOR	**SIGNIFICADO**
=	Igualdad.
!=, < >, ^=	Desigualdad.
<	<
>	Mayor que.
<=	Menor o igual que.
>=	Mayor o igual que.
IN	Igual que cualquiera de los miembros entre paréntesis.
NOT IN	Distinto que cualquiera de los miembros entre paréntesis.
BETWEEN	Entre. Contenido dentro del rango.
NOT BETWEEN	Fuera del rango.
LIKE '_abc%'	Se utiliza sobre todo con textos y permite obtener columnas cuyo valor en un

	campo cumpla una condición textual. Utiliza una cadena que puede contener los símbolos "**%**" que sustituye a un conjunto de caracteres o "**_**" que sustituye a un carácter.
IS NULL	Devuelve verdadero si el valor del campo de la fila que examina es nulo.

El valor `NULL` significaba valor inexistente o desconocido y por tanto es tratado de forma distinta a otros valores. Si queremos verificar que un valor es `NULL` no serán validos los operadores que acabamos de ver. Debemos utilizar los valores `IS NULL` como se indica en la tabla o `IS NOT NULL` que devolverá verdadero si el valor del campo de la fila no es nulo.

Además, cuando se utiliza un `ORDER BY`, los valores `NULL` se presentarán en primer lugar si se emplea el modo ascendente y al final si se usa el descendente.

Si queremos obtener aquellos empleados cuyo salario es superior a 1000€ podemos crear la siguiente consulta:

```
SELECT nombre FROM EMPLEADOS WHERE SALARIO > 1000;
```

Ahora queremos aquellos empleados cuyo apellido comienza por R:

```
SELECT nombre FROM EMPLEADOS WHERE APELLIDO1 LIKE 'R%';
```

Ejercicio resuelto

Utilizando las tablas y datos de la empresa **JuegosCA** descargados anteriormente, vamos a realizar una consulta donde obtengamos las universidades de Sevilla o Cádiz.

Resultado:

```
SELECT UNIV_COD, NOMBRE_UNIV FROM UNIVERSIDADES
WHERE CIUDAD IN ('SEVILLA', 'CÁDIZ');
```

Fíjate que buscará aquellas ciudades que coincidan textualmente con las que ponemos entre comillas.

Los operadores que se utilizan en MySQL puedes verlos en el siguiente enlace:
http://dev.mysql.com/doc/refman/5.0/es/comparison-operators.html

3.2.- Operadores aritméticos y de concatenación.

Aprendimos que los operadores son símbolos que permiten realizar distintos tipos de operaciones. Los operadores aritméticos permiten realizar cálculos con valores numéricos. Son los siguientes:

Operadores aritméticos y su significado.	
OPERADOR	**SIGNIFICADO**
+	Suma
-	Resta
*****	Multiplicación
/	División

Utilizando expresiones con operadores **es posible obtener** salidas en las cuales **una columna sea el resultado de un cálculo** y no un campo de una tabla.

Mira este ejemplo en el que obtenemos el salario aumentado en un 5% de aquellos empleados que cobran menos de 1000€:

```
SELECT SALARIO*1,05
FROM EMPLEADOS
WHERE SALARIO<=1000;
```

Cuando una expresión aritmética se calcula sobre valores `NULL`, el resultado es el propio valor `NULL`.
Para concatenar cadenas de caracteres existe el operador de concatenación (" **||** "). Oracle puede convertir automáticamente valores numéricos a cadenas para una concatenación.

En la tabla EMPLEADOS tenemos separados en dos campos el primer y segundo apellido de los empleados, si necesitáramos mostrarlos juntos podríamos crear la siguiente consulta:

```
SELECT Nombre, Apellido1 || Apellido2
FROM EMPLEADOS;
```

Si queremos dejar un espacio entre un apellido y otro, debemos concatenar también el espacio en blanco de la siguiente manera:

```
SELECT Nombre, Apellido1 || ' ' ||Apellido2
FROM EMPLEADOS;
```

Los operadores que se utilizan en MySQL puedes verlos en el siguiente enlace:
http://dev.mysql.com/doc/refman/5.0/es/arithmetic-functions.html

3.3.- Operadores lógicos.

Habrá ocasiones en las que tengas que evaluar más de una expresión y necesites verificar que se cumple una única condición, otras veces comprobar si se cumple una u otra o ninguna de ellas. Para poder hacer esto utilizaremos los operadores lógicos.

Tenemos los siguientes:

Operadores lógicos y su significado.	
OPERADOR	**SIGNIFICADO**
AND	Devuelve verdadero si sus expresiones a derecha e izquierda son ambas verdaderas.
OR	Devuelve verdadero si alguna de sus expresiones a derecha o izquierda son verdaderas.
NOT	Invierte la lógica de la expresión que le precede, si la expresión es verdadera devuelve falsa y si es falsa devuelve verdadera.

Fíjate en los siguientes ejemplos:
Si queremos obtener aquellos empleados en cuyo historial salarial tengan sueldo menor o igual a 800€ o superior a 2000€:

```
SELECT empleado_dni
FROM HISTORIAL_SALARIAL
WHERE salario<=800 OR salario>2000;
```

Ejercicio resuelto

Utilizando las tablas y datos de la empresa JuegosCA descargados anteriormente, vamos a realizar una consulta donde obtengamos todos nombres de trabajos menos el de contable.

Respuesta:

```
SELECT NOMBRE_TRAB
FROM TRABAJOS
WHERE NOMBRE_TRAB NOT IN ('CONTABLE');
```

3.4.- Precedencia.

Con frecuencia utilizaremos la sentencia SELECT acompañada de expresiones muy extensas y resultará difícil saber que parte de dicha expresión se evaluará primero, por ello es conveniente conocer el orden de precedencia que tiene Oracle:

1. Se evalúa la multiplicación (*****) y la división (**/**) al mismo nivel.
2. A continuación sumas (**+**) y restas (**-**).
3. Concatenación (**| |**).
4. Todas las comparaciones (**<**, **>**, …).
5. Después evaluaremos los operadores **IS NULL**, **IN NOT NULL**, **LIKE**, **BETWEEN**.
6. **NOT**.
7. **AND**.
8. **OR**.

Si quisiéramos variar este orden necesitaríamos utilizar paréntesis.

En la siguiente consulta:

```
SELECT APELLIDOS
FROM JUGADORES
WHERE APELLIDOS LIKE 'A%S%';
```

¿Qué estaríamos seleccionando?

- ○ Aquellos jugadores cuyos apellidos contienen la letra A y la S
- ◉ **Aquellos jugadores cuyos apellidos comienzan por la letra A y contienen la letra S**
- ○ Aquellos jugadores cuyos apellidos no contienen ni la letra A ni la S
- ○ Todos los apellidos de todos los jugadores menos los que su apellido comienza por S

También tenemos orden de precedencia en MySQL:
http://dev.mysql.com/doc/refman/5.0/es/operator-precedence.html

4.- Consultas calculadas.

Caso práctico

*A la empresa ha llegado **Carlos** que está en fase de prácticas y como anda un poco desubicado ha comenzado su trabajo revisando la teoría y práctica que han dado en clase. No recuerda bien como se creaban campos nuevos a partir de otros ya existentes en la base de datos. Sabe que es algo sencillo pero no quiere meter la pata ya que está ayudando a **Juan** en un proyecto que acaba de entrar.*

Lo que hará será practicar a partir de una tabla que tenga bastantes campos numéricos de manera que pueda manipular la información sin modificar nada.

En clase trabajaban con la tabla ARTICULOS que tenía, entre otros, los campos Precio y Cantidad. A partir de ellos podría realizar consultas calculadas para obtener el precio con IVA incluido, un descuento sobre el precio e incluso aumentar ese precio en un porcentaje concreto. Seguro que se pone al día rápidamente.

En algunas ocasiones es interesante realizar operaciones con algunos campos para obtener información derivada de éstos. Si tuviéramos un campo Precio, podría interesarnos calcular el precio incluyendo el IVA o si tuviéramos los campos Sueldo y Paga Extra, podríamos necesitar obtener la suma de los dos campos. Estos son dos ejemplos simples pero podemos construir expresiones mucho más complejas. Para ello haremos uso de la creación de campos calculados.

Los operadores aritméticos se pueden utilizar para hacer cálculos en las consultas.

Estos **campos calculados** se obtienen a través de la sentencia `SELECT` poniendo a continuación la expresión que queramos. Esta consulta **no modificará los valores originales** de las columnas ni de la tabla de la que se está obteniendo dicha consulta, únicamente mostrará una columna nueva con los valores calculados. Por ejemplo:

```
SELECT Nombre, Credito, Credito + 25
FROM USUARIOS;
```

Con esta consulta hemos creado un campo que tendrá como nombre la expresión utilizada. Podemos ponerle un alias a la columna creada añadiéndolo detrás de la expresión junto con la palabra AS. En nuestro ejemplo quedaría de la siguiente forma:

```
SELECT Nombre, Credito, Credito + 25 AS CreditoNuevo
FROM USUARIOS;
```

Los campos calculados pueden ir en:

- [x] La cláusula SELECT
- [x] La cláusula WHERE
- [ ] La cláusula FROM

Así es, y también podemos añadir un alias a ese campo poniéndolo a continuación.

5.- Funciones.

Caso práctico

__Juan__ le ha pedido a __Ana__ que calcule la edad actual de los usuarios que tienen registrados en la base de datos pues sería interesante realizar estadísticas mensuales sobre los grupos de edad que acceden al sistema y en función de ello obtener algunos resultados interesantes para la empresa. Para realizar el cálculo de la edad tendríamos que echar mano a funciones que nos ayuden con los cálculos. Existen funciones que nos facilitarán la tarea y nos ayudarán a obtener información que de otro modo resultaría complicado.

¿Has pensado en todas las operaciones que puedes realizar con los datos que guardas en una base de datos? Seguro que son muchísimas. Pues bien, en casi todos los Sistemas Gestores de Base de Datos existen funciones ya creadas que facilitan la creación de consultas más complejas. Dichas funciones varían según el SGBD, veremos aquí las que utiliza Oracle.

Las funciones son realmente operaciones que se realizan sobre los datos y que realizan un determinado cálculo. Para ello necesitan unos datos de entrada llamados parámetros o argumentos y en función de éstos, se realizará el cálculo de la función que se esté utilizando. Normalmente los parámetros se especifican entre paréntesis.

Las funciones se pueden incluir en las cláusulas `SELECT`, `WHERE` y `ORDER BY`.

Las funciones se especifican de la siguiente manera:

```
NombreFunción [(parámetro1, [parámetro2, …)]
```

Puedes anidar funciones dentro de funciones.

Existe una gran variedad para cada tipo de datos:
- ✓ numéricas,
- ✓ de cadena de caracteres,
- ✓ de manejo de fechas,
- ✓ de conversión,
- ✓ otras

Oracle proporciona una tabla con la que podemos hacer pruebas, esta tabla se llama Dual y contiene un único campo llamado DUMMY y una sola fila.

Podremos utilizar la tabla Dual en algunos de los ejemplos que vamos a ver en los siguientes apartados.

5.1.- Funciones numéricas.

¿Cómo obtenemos el cuadrado de un número o su valor absoluto? Nos referimos a valores numéricos y por tanto necesitaremos utilizar funciones numéricas.

Para trabajar con campos de tipo número tenemos las siguientes funciones:

Funciones Numéricas	
ABS(n)	Calcula el valor ***absoluto*** de un número **n** `SELECT ABS(-17) FROM DUAL; -- Resultado: 17`
EXP(n)	Calcula $\mathbf{e}^{\mathbf{n}}$, es decir, el ***exponente*** en base **e** del número **n** `SELECT EXP(2) FROM DUAL; -- Resultado: 7,38`
CEIL(n)	Calcula el valor ***entero*** inmediatamente ***superior*** o igual al

	argumento **n** `SELECT CEIL(17.4) FROM DUAL; -- Resultado: 18`
FLOOR(n)	Calcula el valor ***entero*** inmediatamente ***inferior*** o igual al parámetro **n** `SELECT FLOOR(17.4) FROM DUAL; --Resultado: 17`
MOD(m,n)	Calcula el ***resto*** resultante de dividir **m** entre **n** `SELECT MOD(15, 2) FROM DUAL; --Resultado: 1`
POWER(valor, exponente)	***Eleva*** el valor ***al exponente*** indicado `SELECT POWER(4, 5) FROM DUAL; -- Resultado: 1024`
ROUND(n, decimales)	**Redondea** el número **n** al siguiente número con el número de decimales que se indican. `SELECT ROUND(12.5874, 2) FROM DUAL; -- Resultado: 12.59`
SQRT(n)	Calcula la ***raíz cuadrada*** de **n**. `SELECT SQRT(25) FROM DUAL; --Resultado: 5`

TRUNC(m,n)	***Trunca*** un número a la cantidad de decimales especificada por el segundo argumento. Si se omite el segundo argumento, se truncan todos los decimales. Si "**n**" es negativo, el número es truncado desde la parte entera.
	`SELECT TRUNC(127.4567, 2) FROM DUAL; -- Resultado: 127.45`
	`SELECT TRUNC(4572.5678, -2) FROM DUAL; -- Resultado: 4500`
	`SELECT TRUNC(4572.5678, -1) FROM DUAL; -- Resultado: 4570`
	`SELECT TRUNC(4572.5678) FROM DUAL; -- Resultado: 4572`

SIGN(n)	Si el argumento "**n**"es un valor ***positivo***, retorna ***1***, si es ***negativo***, devuelve ***-1*** y 0 si es 0. `SELECT SIGN(-23) FROM DUAL; - Resultado: -1`

Aquí encontrarás las funciones que has visto y algunas más.
http://sites.google.com/site/josepando/home/funciones-sql/funciones-que-devuelven-una-valor-nico-para-cada-fila-de-una-consulta-o-vista/funciones-numricas

5.2.- Funciones de cadena de caracteres.

Ya verás como es muy común manipular campos de tipo carácter o cadena de caracteres. Como resultado podremos obtener caracteres o números. Estas son las funciones más habituales:

Funciones de cadena de caracteres	
CHR(n)	Devuelve el carácter cuyo valor codificado es **n**. `SELECT CHR(81) FROM DUAL; --Resultado: Q`
ASCII(n)	Devuelve el ***valor ASCII*** de **n** `SELECT ASCII('O') FROM DUAL; --Resultado: 79`
CONCAT(cad1, cad2)	Devuelve ***las dos cadenas unidas***. Es equivalente al operador **||** `SELECT CONCAT('Hola', 'Mundo') FROM DUAL; --Resultado: HolaMundo`

LOWER(cad)	Devuelve la cadena **cad** con ***todos*** sus ***caracteres*** en ***minúsculas***.
	`SELECT LOWER('En MInúsculAS') FROM DUAL;`
	`--Resultado: en minúsculas`

UPPER(cad)	Devuelve la cadena **cad** con ***todos*** sus ***caracteres*** en ***mayúsculas***.
	`SELECT UPPER('En MAyúsculAS') FROM DUAL;`
	`--Resultado: EN MAYÚSCULAS`

INITCAP(cad)	Devuelve la cadena **cad** con su ***primer carácter*** en ***mayúscula***. `SELECT INITCAP('hola') FROM DUAL; --Resultado: Hola`
LPAD(cad1, n, cad2)	Devuelve **cad1** con longitud **n**, ajustada a la derecha, ***rellenando por la izquierda*** con **cad2**. `SELECT LPAD('M', 5, '*') FROM DUAL; --Resultado: ****M`

RPAD(cad1, n, cad2)	Devuelve **cad1** con longitud **n**, ajustada a la izquierda, ***rellenando por la derecha*** con **cad2**. `SELECT RPAD('M', 5, '*') FROM DUAL; --Resultado: M****`

REPLACE(cad, ant, nue)	Devuelve ***cad*** en la que cada ocurrencia de la cadena ***ant*** ha sido ***sustituida por*** la cadena ***nue***.
	`SELECT REPLACE('correo@gmail.es', 'es', 'com') FROM DUAL;`
	`--Resultado: correo@gmail.com`

SUBSTR(cad, m, n)	Devuelve la cadena ***cad*** compuesta por ***n caracteres a partir*** de la posición ***m***. `SELECT SUBSTR('1234567', 3, 2) FROM DUAL; --Resultado: 34`
LENGTH(cad)	Devuelve la ***longitud de cad*** `SELECT LENGTH('hola') FROM DUAL; --Resultado: 4`

TRIM(cad)	***Elimina*** los ***espacios*** en blanco a la ***izquierda y*** la ***derecha*** de ***cad*** y los espacios dobles del interior.
	`SELECT TRIM(' Hola de nuevo ') FROM DUAL;`
	`--Resultado: Hola de nuevo`

LTRIM(cad)	***Elimina*** los ***espacios*** a la ***izquierda*** que posea ***cad***. `SELECT LTRIM(' Hola') FROM DUAL; --Resultado: Hola`
RTRIM(cad)	***Elimina*** los ***espacios*** a la ***derecha*** que posea cad. `SELECT RTRIM('Hola ') FROM DUAL; --Resultado: Hola`

INSTR(cad, cadBuscada [, posInicial [, nAparición]])	Obtiene la ***posición*** en la que se encuentra la ***cadena buscada en*** la ***cadena inicial cad***. Se puede comenzar a buscar desde una posición inicial concreta e incluso indicar el número de aparición de la cadena buscada. Si no encuentra nada devuelve cero.
	`SELECT INSTR('usuarios', 'u') FROM DUAL; --Resultado: 1`
	`SELECT INSTR('usuarios', 'u', 2) FROM DUAL; --Resultado: 3`
	`SELECT INSTR('usuarios', 'u', 2, 2) FROM DUAL; --Resultado: 0`

En la siguiente consulta: SELECT LENGTH("Adiós") FROM DUAL; ¿qué obtendríamos?

- ○ 5
- ○ 4
- ○ 6
- ◉ **Nos devolvería un error.**

Cierto, las cadenas van con comillas simples.

5.3.- Funciones de manejo de fechas.

La fecha de emisión de una factura, de llegada de un avión, de ingreso en una web, podríamos seguir poniendo infinidad de ejemplos, lo que significa que es una información que se requiere en muchas situaciones y es importante guardar.

En los SGBD se utilizan mucho las fechas. Oracle tiene dos tipos de datos para manejar fechas, son `DATE` y `TIMESTAMP`.

- ✓ `DATE` almacena **fechas concretas incluyendo a veces la hora.**
- ✓ `TIMESTAMP` almacena **un instante de tiempo más concreto** que puede incluir hasta fracciones de segundo.

Podemos realizar operaciones numéricas con las fechas:

- ✓ Le podemos **sumar números** y esto se entiende como sumarles días, si ese número tiene decimales se suman días, horas, minutos y segundos.
- ✓ La **diferencia** entre dos fechas también nos dará un número de días.

En Oracle tenemos las siguientes funciones más comunes:

Funciones de fecha	
SYSDATE	Devuelve la ***fecha y hora actuales*** `SELECT SYSDATE FROM DUAL; --Resultado: 26/07/11`

SYSTIMESTAMP	Devuelve la ***fecha y hora actuales*** en formato `TIMESTAMP`
	`SELECT SYSTIMESTAMP FROM DUAL;`
	`--Resultado: 26-JUL-11 08.32.59,609000 PM +02:00`

ADD_MONTHS(fecha, n)	***Añade a la fecha*** el número de ***meses*** indicado con ***n***
	`SELECT ADD_MONTHS('27/07/11', 5) FROM DUAL;`
	`--Resultado: 27/12/11`

MONTHS_BETWEEN(fecha1, fecha2)	Devuelve el ***número de meses*** que hay ***entre fecha1*** y ***fecha2***
	`SELECT MONTHS_BETWEEN('12/07/11','12/03/11') FROM DUAL;`
	`--Resultado: 4`

LAST_DAY(fecha)	Devuelve el ***último día del mes*** al que pertenece la fecha. El valor devuelto es tipo `DATE` `SELECT LAST_DAY('27/07/11') FROM DUAL; --Resultado: 31/07/11`

NEXT_DAY(fecha, d)	Indica el ***día*** que corresponde ***si añadimos*** a la fecha el día **d**. El día devuelto puede ser texto ('Lunes', Martes', ..) o el número del día de la semana (1=lunes, 2=martes, ..) dependiendo de la configuración.
	`SELECT NEXT_DAY('31/12/11','LUNES') FROM DUAL;`
	`--Resultado: 02/01/12`

EXTRACT(valor FROM fecha)	Extrae un valor de una fecha concreta. El valor puede ser `day`, `month`, `year`, `hours`, etc. `SELECT EXTRACT(MONTH FROM SYSDATE) FROM DUAL; --Resultado: 7`

En Oracle: Los operadores aritméticos "**+**" (más) y "**-**" (menos) pueden emplearse para las fechas. Por ejemplo:

```
SELECT SYSDATE - 5;
```

Devolvería la fecha correspondiente a 5 días antes de la fecha actual.

Se pueden emplear estas funciones enviando como argumento el nombre de un campo de tipo fecha.

¿Cuáles de estas afirmaciones sobre funciones de manejo de fechas son ciertas?

- ☑ **Existen dos tipos de fechas de datos con las que podemos trabajar, DATE y TIMESTAMP.**
- ☐ Se puede poner como argumento el nombre de un campo de cualquier tipo.
- ☑ **Le podemos sumar o restar números, lo cual se entiende como sumarle o restarle días.**
- ☑ **La diferencia entre dos fechas nos dará un número de días.**

5.4.- Funciones de conversión.

Los SGBD tienen funciones que pueden pasar de un tipo de dato a otro. Oracle convierte automáticamente datos de manera que el resultado de una expresión tenga sentido. Por tanto, de manera automática se pasa de texto a número y al revés. Ocurre lo mismo para pasar de tipo texto a

fecha y viceversa. Pero existen ocasiones en que queramos realizar esas conversiones de modo explícito, para ello contamos con funciones de conversión.

- ✓ TO_NUMBER(cad, formato) Convierte textos en números. Se suele utilizar para dar un formato concreto a los números. Los formatos que podemos utilizar son los siguientes:

Formatos para números y su significado.	
Símbolo	**Significado**
9	**Posiciones numéricas**. Si el número que se quiere visualizar contiene menos dígitos de los que se especifican en el formato, se rellena con blancos.
0	Visualiza **ceros por la izquierda** hasta completar la longitud del formato especificado.
$	**Antepone** el signo de **dólar** al número.
L	Coloca en la posición donde se incluya, el **símbolo de la moneda local** (se puede configurar en la base de datos mediante el parámetro `NSL CURRENCY`)
S	Aparecerá el **símbolo** del signo.
D	**Posición** del símbolo **decimal**, que en español es la coma.
G	**Posición** del **separador de grupo**, que en español es el punto.

- ✓ `TO CHAR(d, formato)` Convierte un número o fecha d a cadena de caracteres, se utiliza normalmente para fechas ya que de número a texto se hace de forma implícita como hemos visto antes.
- ✓ `TO DATE( cad, formato)` Convierte textos a fechas. Podemos indicar el formato con el que queremos que aparezca.

Para las funciones `TO CHAR` y `TO DATE`, en el caso de fechas, indicamos el formato incluyendo los siguientes símbolos:

Formatos para fechas y su significado.	
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
DD	Día en formato de dos cifras
D	Día de la semana del 1 al 7
Q	Semestre
WW	Semana del año
AM **PM**	Indicador a.m. Indicador p.m.
HH12 **HH24**	Hora de 1 a 12 Hora de 0 a 23
MI	Minutos de 0 a 59
SS **SSSS**	Segundos dentro del minuto Segundos dentro desde las 0 horas

5.5.- Otras funciones: NVL y DECODE.

¿Recuerdas que era el valor `NULL`? Cualquier columna de una tabla podía contener un valor nulo independientemente al tipo de datos que tuviera definido. Eso sí, esto no era así en los casos en que definíamos esa columna como no nula (`NOT NULL`), o que fuera clave primaria (`PRIMARY KEY`).

Cualquier operación que se haga con un valor `NULL` devuelve un `NULL`. Por ejemplo, si se intenta dividir por `NULL`, no nos aparecerá ningún error sino que como resultado obtendremos un `NULL` (no se producirá ningún error tal y como puede suceder si intentáramos dividir por cero).

También es posible que el resultado de una función nos de un valor nulo.

Por tanto, es habitual encontrarnos con estos valores y es entonces cuando aparece la necesidad de poder hacer algo con ellos. Las **funciones con nulos** nos permitirán hacer algo en caso de que aparezca un valor nulo.

- ✓ `NVL(valor, expr1)`

  Si valor es **NULL**, entonces devuelve **expr1**. Ten en cuenta que **expr1** debe ser del mismo tipo que **valor**.

  ¿Y no habrá alguna función que nos permita evaluar expresiones? La respuesta es afirmativa y esa función se llama `DECODE`.

- ✓ `DECODE(expr1, cond1, valor1 [, cond2, valor2, ...], default )`

  Esta función evalúa una expresión **expr1**, si se cumple la primera condición (**cond1**) devuelve el **valor1**, en caso contrario evalúa la siguiente condición y así hasta que una de las condiciones se cumpla. Si no se cumple ninguna condición se devuelve el valor por defecto que hemos llamado **default**.

Si en la tabla EMPLEADOS queremos un listado de sus direcciones, podemos pedir que cuando una dirección no exista, aparezca el texto **No tiene dirección**, para ello podemos utilizar la siguiente consulta:

```
SELECT NVL(DIRECC1, 'No tiene dirección conocida') FROM EMPLEADOS;
```

Obtendremos:

Resultado de la sentencia SELECT

DIRECCIONES

No tiene dirección conocida

C/Sol, 1

¿Qué función convierte un número o fecha a cadena de caracteres?

- ○ TO_DATE.
- ◉ **TO_CHAR.**
- ○ DECODE.
- ○ TO_NUMBER.

Así es, se utiliza normalmente para fechas ya que de número a texto se hace de forma implícita

6.- Consultas de resumen.

Caso práctico

__Ada__ le ha pedido a __Juan__ que le eche una mano en otro de los proyectos en los que está inmersa la empresa. Necesita que cree varias consultas de resumen sobre unas tablas de empleados de banca. Está interesada en obtener el salario medio de los empleados clasificado por tipo de empleado, y quiere también que obtenga el total de empleados por sucursal.

Realmente no es un trabajo difícil ya que las consultas de resumen son muy fáciles de crear, pero __Ada__ está tan ocupada que no tiene tiempo para esos detalles.

Seguro que alguna vez has necesitado realizar cálculos sobre un campo para obtener algún resultado global, por ejemplo, si tenemos una columna donde estamos guardando las notas que obtienen unos alumnos o alumnas en Matemáticas, podríamos estar interesados en saber cual es la nota máxima que han obtenido o la nota media.

La sentencia `SELECT` nos va a permitir **obtener resúmenes de los datos de modo vertical**. Para ello consta de una serie de cláusulas específicas (`GROUP BY`, `HAVING`) y tenemos también unas **funciones** llamadas **de agrupamiento o de agregado** que son las que nos dirán qué cálculos queremos realizar sobre los datos (sobre la columna).

Hasta ahora las consultas que hemos visto daban como resultado un subconjunto de filas de la tabla de la que extraíamos la información. Sin embargo, este tipo de consultas que vamos a ver no corresponde con ningún valor de la tabla sino un **total calculado** sobre los datos de la tabla. Esto hará que las consultas de resumen tengan limitaciones que iremos viendo.

Las funciones que podemos utilizar se llaman de agrupamiento (de agregado). Éstas **toman un grupo de datos** (una columna) y **producen un único dato que resume el grupo**. Por ejemplo, la función `SUM()` acepta una columna de datos numéricos y devuelve la suma de estos.

El simple hecho de utilizar una función de agregado en una consulta la convierte en consulta de resumen.

Todas las funciones de agregado tienen una estructura muy parecida: `FUNCIÓN ([ALL| DISTINCT] Expresión)` y debemos tener en cuenta que:

- ✓ La palabra `ALL` indica que se tienen que tomar todos los valores de la columna. Es el valor por defecto.
- ✓ La palabra `DISTINCT` indica que se considerarán todas las repeticiones del mismo valor como uno solo (considera valores distintos).
- ✓ El grupo de valores sobre el que actúa la función lo determina el resultado de la expresión que será el nombre de una columna o una expresión basada en una o varias columnas. Por tanto, en la expresión nunca puede aparecer ni una función de agregado ni una subconsulta.
- ✓ Todas las funciones se aplican a las filas del origen de datos una vez ejecutada la cláusula `WHERE` (si la tuviéramos).
- ✓ Todas las funciones (excepto `COUNT`) ignoran los valores `NULL`.
- ✓ Podemos encontrar una función de agrupamiento dentro de una lista de selección en cualquier sitio donde pudiera aparecer el nombre de una columna. Es por eso que puede formar parte de una expresión pero no se pueden anidar funciones de este tipo.
- ✓ No se pueden mezclar funciones de columna con nombres de columna ordinarios, aunque hay excepciones que veremos más adelante.

Ya estamos preparados para conocer cuáles son estas funciones de agregado (o agrupamiento). Las veremos a continuación.

Puedes acceder a este enlace si quieres conocer más sobre este tipo de consultas.
http://www.aulaclic.es/sql/t_4_1.htm

6.1.- Funciones de agregado: SUM y COUNT.

Sumar y contar filas o datos contenidos en los campos es algo bastante común. Imagina que para nuestra tabla Usuarios necesitamos sumar el número de créditos total que tienen nuestros jugadores. Con una función que sumara los valores de la columna crédito sería suficiente, siempre y cuando lo agrupáramos por cliente, ya que de lo contrario lo que obtendríamos sería el total de todos los clientes jugadores.

- ✓ La función `SUM`:
  - `SUM([ALL|DISTINCT] expresión)`
  - Devuelve la suma de los valores de la expresión.
  - Sólo puede utilizarse con columnas cuyo tipo de dato sea número. El resultado será del mismo tipo aunque puede tener una precisión mayor.
  - Por ejemplo,
    ```
    SELECT SUM( credito) FROM Usuarios;
    ```
- ✓ La función `COUNT`:
  - `COUNT([ALL|DISTINCT] expresión)`
  - Cuenta los elementos de un campo. Expresión contiene el nombre del campo que deseamos contar. Los operandos de expresión pueden incluir el nombre del campo, una constante o una función.
  - Puede contar cualquier tipo de datos incluido texto.
  - `COUNT` simplemente cuenta el número de registros sin tener en cuenta qué valores se almacenan.
  - La función `COUNT` no cuenta los registros que tienen campos `NULL` a menos que **expresión** sea el carácter comodín **asterisco** (*).
  - Si utilizamos `COUNT(*)`, calcularemos el total de filas, incluyendo aquellas que contienen valores `NULL`.
  - Por ejemplo,
    ```
    SELECT COUNT(nombre) FROM Usuarios;
    SELECT COUNT(*) FROM Usuarios;
    ```

Ejercicio resuelto

Utilizando las tablas y datos de la empresa **JuegosCA** descargados anteriormente, vamos a realizar una consulta donde contemos el número de empleados que son mujeres.

Resultado:

```
SELECT COUNT(Nombre) FROM EMPLEADOS WHERE SEXO='M';
```

6.2.- Funciones de agregado: MIN y MAX.

¿Y si pudiéramos encontrar el valor máximo y mínimo de una lista enormemente grande? Esto es lo que nos permiten hacer las siguientes funciones.

- ✓ **Función** `MIN`:
  - `MIN ([ALL| DISTINCT] expresión)`
  - Devuelve el valor mínimo de la expresión sin considerar los nulos (`NULL`).
  - En expresión podemos incluir el nombre de un campo de una tabla, una constante o una función (pero no otras funciones agregadas de SQL).

- Un ejemplo sería:

```
SELECT MIN(credito) FROM Usuarios;
```

- ✓ **Función `MAX`:**
  - `MAX ([ALL| DISTINCT] expresión)`
  - Devuelve el valor máximo de la expresión sin considerar los nulos (`NULL`).
  - En expresión podemos incluir el nombre de un campo de una tabla, una constante o una función (pero no otras funciones agregadas de SQL).
  - Un ejemplo,

```
SELECT MAX (credito) FROM Usuarios;
```

6.3.- Funciones de agregado: AVG, VAR, STDEV y STDEVP.

Quizás queramos obtener datos estadísticos de los datos guardados en nuestra base de datos. Para ello podemos hacer uso de las funciones que calculan el promedio, la varianza y la desviación típica.

- ✓ **Función `AVG`**
  - `AVG ([ALL| DISTINCT] expresión)`
  - Devuelve el promedio de los valores de un grupo, para ello se omiten los valores nulos (`NULL`).
  - El grupo de valores será el que se obtenga como resultado de la expresión y ésta puede ser un nombre de columna o una expresión basada en una columna o varias de la tabla.
  - Se aplica a campos tipo número y el tipo de dato del resultado puede variar según las necesidades del sistema para representar el valor.
- ✓ **Función `VAR`**
  - `VAR ([ALL| DISTINCT] expresión)`
  - Devuelve la varianza estadística de todos los valores de la expresión.
  - Como tipo de dato admite únicamente columnas numéricas. Los valores nulos (`NULL`) se omiten.
- ✓ **Función `STDEV`**
  - `STDEV ([ALL| DISTINCT] expresión)`
  - Devuelve la desviación típica estadística de todos los valores de la expresión.
  - Como tipo de dato admite únicamente columnas numéricas. Los valores nulos (`NULL`) se omiten.

Ejercicio resuelto

Utilizando las tablas y datos de la empresa **JuegosCA** descargados anteriormente, vamos a realizar una consulta donde obtengamos la media del salario mínimo y máximo de la tabla TRABAJOS.

Resultado:

```
SELECT AVG(SALARIO_MIN), AVG(SALARIO_MAX) FROM TRABAJOS;
```

¿Cuáles de las siguientes afirmaciones sobre las consultas de resumen son ciertas?

- [x] Toman un grupo de datos de una columna.
- [x] Producen un único dato que resume el grupo.
- [x] Utilizar una función de agregado en una consulta la convierte en consulta de resumen.
- [ ] Dan como resultado un subconjunto de filas de la tabla.

7.- Agrupamiento de registros.

Caso práctico

__Juan__ ha estado realizando algunas consultas de resumen y ahora quiere continuar sacando todo el jugo posible a las tablas realizando operaciones como las anteriores pero agrupándolas por algún campo. Hay veces que se obtiene mucha información si estudiamos los datos por grupos, como puede ser el número de jugadores por provincia, o el saldo medio según el sexo del jugador para así poder obtener conclusiones sobre la información guardada.

Hasta aquí las consultas de resumen que hemos visto obtienen totales de todas las filas de un campo o una expresión calculada sobre uno o varios campos. Lo que hemos obtenido ha sido una única fila con un único dato.

Ya verás como en muchas ocasiones en las que utilizamos consultas de resumen nos va a interesar calcular **totales parciales**, es decir, **agrupados según un determinado campo**.

De este modo podríamos obtener de una tabla EMPLEADOS, en la que se guarda su sueldo y su actividad dentro de la empresa, el valor medio del sueldo en función de la actividad realizada en la empresa. También podríamos tener una tabla clientes y obtener el número de veces que ha realizado un pedido, etc.

En todos estos casos en lugar de una única fila de resultados necesitaremos una fila por cada actividad, cada cliente, etc.

Podemos obtener estos **subtotales** utilizando la cláusula `GROUP BY`.

La sintaxis es la siguiente:

```
SELECT columna1, columna2, ...
FROM tabla1, tabla2, ...
WHERE condición1, condición2, …
GROUP BY columna1, columna2, …
HAVING condición
ORDER BY ordenación;
```

En la cláusula `GROUP BY` se colocan las columnas por las que vamos a agrupar. En la cláusula `HAVING` se especifica la condición que han de cumplir los grupos para que se realice la consulta.

Es muy importante que te fijes bien en el orden en el que se ejecutan las cláusulas:

1. `WHERE` que filtra las filas según las condiciones que pongamos.
2. `GROUP BY` que crea una tabla de grupos nueva.
3. `HAVING` filtra los grupos.
4. `ORDER BY` que ordena o clasifica la salida.

Las columnas que aparecen en el `SELECT` y que no aparezcan en la cláusula `GROUP BY` deben tener una función de agrupamiento. Si esto no se hace así producirá un error. Otra opción es poner en la claúsula `GROUP BY` las mismas columnas que aparecen en `SELECT`.

Veamos un par de ejemplos:

```
SELECT provincia, SUM(credito) FROM Usuarios GROUP BY provincia;
```

Obtenemos la suma de créditos de nuestros usuarios agrupados por provincia. Si estuviéramos interesados en la suma de créditos agrupados por provincia pero únicamente de las provincias de Cádiz y Badajóz nos quedaría:

PROVINCIA	COUNT(CREDITO)
BADAJOZ	3
CÁDIZ	4

```
SELECT provincia, SUM(credito) FROM Usuarios GROUP BY provincia HAVING provincia = 'CÁDIZ' OR
provincia= 'BADAJOZ';
```

Relaciona cada cláusula con su orden de ejecución:

Cláusula.	Relación.	Función.
WHERE	**1**	1. PRIMERO
ORDER BY	**4**	2. SEGUNDO
HAVING	**3**	3. TERCERO
GROUP BY	**2**	4. CUARTO

8.- Consultas multitablas.

Caso práctico

*Hasta ahora **Juan** ha estado haciendo uso de consultas a una única tabla, pero eso limita la obtención de resultados. Si esas tablas están relacionadas **Juan** podrá coger información de cada una de ellas según lo que le interese. En las tablas de la empresa **JuegosCA**, tiene por un lado la tabla que recoge los datos del empleado y por otra su historial laboral. En esta última tabla, lo único que recogemos de los empleados es su código. Como a **Juan** le interesa obtener el historial laboral incluyendo nombres y apellidos de sus empleados, debe utilizar la información que viene en ambas tablas.*

Recuerda que una de las propiedades de las bases de datos relacionales era que distribuíamos la información en varias tablas que a su vez estaban relacionadas por algún campo común. Así evitábamos repetir datos. Por tanto, también será frecuente que tengamos que consultar datos que se encuentren distribuidos por distintas tablas.

Si disponemos de una tabla USUARIOS cuya clave principal es Login y esta tabla a su vez está relacionada con la tabla PARTIDAS a través del campo **Cod_Creación**. Si quisiéramos obtener el nombre de los usuarios y las horas de las partidas de cada jugador necesitaríamos coger datos de ambas tablas pues las horas se guardan en la tabla PARTIDAS. Esto significa que cogeremos filas de una y de otra.

Imagina también que en lugar de tener una tabla USUARIOS, dispusiéramos de dos por tenerlas en servidores distintos. Lo lógico es que en algún momento tendríamos que unirlas.

Hasta ahora las consultas que hemos usado se referían a una sola tabla, pero también es posible hacer consultas usando varias tablas en la misma sentencia `SELECT`. Esto permitirá realizar distintas operaciones como son:

- ✓ La composición interna.
- ✓ La composición externa.

En la versión SQL de 1999 se especifica una nueva sintaxis para consultar varias tablas que Oracle incorpora, así que también la veremos. La razón de esta nueva sintaxis era separar las condiciones de asociación respecto a las condiciones de selección de registros.

La sintaxis es la siguiente:

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

8.1.- Composiciones internas.

¿Qué ocurre si combinamos dos o más tablas sin ninguna restricción? El resultado será un producto cartesiano.

El producto cartesiano entre dos tablas da como resultado todas las combinaciones de todas las filas de esas dos tablas.

Se indica poniendo en la cláusula FROM las tablas que queremos componer separadas por comas. Y puedes obtener el producto cartesiano de las tablas que quieras.

Como lo que se obtiene son todas las posibles combinaciones de filas, debes tener especial cuidado con las tablas que combinas. Si tienes dos tablas de 10 filas cada una, el resultado tendrá 10x10 filas,

a medida que aumentemos el número de filas que contienen las tablas, mayor será el resultado final, con lo cual se puede considerar que nos encontraremos con una operación costosa.

Esta operación no es de las más utilizadas ya que coge una fila de una tabla y la asocia con todos y cada uno de las filas de la otra tabla, independientemente de que tengan relación o no. Lo más normal es que queramos seleccionar los registros según algún criterio.

Necesitaremos **discriminar** de alguna forma para que únicamente aparezcan filas de una tabla que estén relacionadas con la otra tabla. A esto se le llama **asociar tablas** (`JOIN`).

Para hacer una composición interna se parte de un producto cartesiano y se eliminan aquellas filas que no cumplen la condición de composición.

Lo importante en las composiciones internas es emparejar los campos que han de tener valores iguales.

Las reglas para las composiciones son:
- ✓ Pueden combinarse tantas tablas como se desee.
- ✓ El criterio de combinación puede estar formado por más de una pareja de columnas.
- ✓ En la cláusula `SELECT` pueden citarse columnas de ambas tablas, condicionen o no, la combinación.
- ✓ Si hay columnas con el mismo nombre en las distintas tablas, deben identificarse especificando la tabla de procedencia o utilizando un alias de tabla.

Las columnas que aparecen en la cláusula `WHERE` se denominan **columnas de emparejamiento** ya que son las que permiten emparejar las filas de las dos tablas. Éstas no tienen por qué estar incluidas en la lista de selección. Emparejaremos tablas que estén relacionadas entres sí y además, una de las columnas de emparejamiento será clave principal en su tabla. Cuando emparejamos campos debemos especificar de la siguiente forma:

```
NombreTabla1. Camporelacionado1 = NombreTabla2.Camporelacionado2.
```

Puedes combinar una tabla consigo misma pero debes poner de manera obligatoria un alias a uno de los nombres de la tabla que vas a repetir.

Veamos un ejemplo, si queremos obtener el historial laboral de los empleados incluyendo nombres y apellidos de los empleados, la fecha en que entraron a trabajar y la fecha de fin de trabajo si ya no continúan en la empresa, tendremos:

```
SELECT Nombre, Apellido1, Apellido2, Fecha_inicio, Fecha_fin
FROM EMPLEADOS, HISOTRIAL LABORAL
WHERE HISTORIAL_LABORAL.Empleado_DNI= EMPLEADOS.DNI;
```

Vamos a obtener el historial con los nombres de departamento, nombre y apellidos del empleado de todos los departamentos:

```
SELECT Nombre_Dpto, Nombre, Apellido1, Apellido2
FROM DEPARTAMENTOS, EMPLEADOS, HISTORIAL LABORAL
WHERE EMPLEADOS.DNI= HISTORIAL_LABORAL. EMPLEADO_DNI
AND HISTORIAL_LABORAL.DPTO_COD = DEPARTAMENTOS. DPTO_COD;
```

Ejercicio resuelto

Utilizando las tablas y datos de la empresa JuegosCA descargados anteriormente, vamos a realizar una consulta donde obtengamos el nombre de los empleados junto a su salario.

Respuesta:

```
SELECT Nombre, Apellido1, Apellido2, Salario
FROM EMPLEADOS, HISTORIAL_SALARIAL
WHERE HISTORIAL_SALARIAL.Empleado_DNI.= EMPLEADOS.DNI;
```

Ejercicio resuelto

Obtener un listado con el histórico laboral de un empleador cuyo DNI sea '12345'. En dicho listado interesa conocer el nombre del puesto, así como el rango salarial.

Respuesta:

```
SELECT    T.NOMBRE_TRAB,    T.SALARIO_MIN,    T.SALARIO_MAX,    HL.EMPLEADO_DNI,    HL.TRAB_COD,
HL.FECHA INICIO, HL.FECHA FIN, HL.DPTO COD, HL.SUPERVISOR DNI
FROM TRABAJOS T, HISTORIAL LABORAL HL
WHERE T.TRABAJO_COD=HL.TRAB_COD AND
EMPLEADO_DNI='12345';
```

8.2.- Composiciones externas.

¿Has pensado que puede que te interese seleccionar algunas filas de una tabla aunque éstas no tengan correspondencia con las filas de la otra tabla? Esto puede ser necesario.

Imagina que tenemos en una base de datos guardadas en dos tablas la información de los empleados de la empresa (**Cod_empleado, Nombre, Apellidos, salario y Cod_dpto**) por otro lado los departamentos (**Codigo_dep, Nombre**) de esa empresa. Recientemente se ha remodelado la empresa y se han creado un par de departamentos más pero no se les ha asignado los empleados. Si tuviéramos que obtener un informe con los datos de los empleados por departamento, seguro que deben aparecer esos departamentos aunque no tengan empleados. Para poder hacer esta combinación usaremos las composiciones externas.

¿Cómo es el formato? Muy sencillo, añadiremos un signo más entre paréntesis (+) en la igualdad entre campos que ponemos en la cláusula WHERE. El carácter (+) irá detrás del nombre de la tabla en la que deseamos aceptar valores nulos.

En nuestro ejemplo, la igualdad que tenemos en la cláusula WHERE es Cod_dpto (+)= Codigo_dep ya que es en la tabla empleados donde aparecerán valores nulos.

Ejercicio resuelto

Obtener un listado con los nombres de los distintos departamentos y sus jefes con sus datos personales. Ten en cuenta que deben aparecer todos los departamentos aunque no tengan asignado ningún jefe.

Resultado:

```
SELECT  D.NOMBRE_DPTO, D.JEFE, E.NOMBRE, E.APELLIDO1, E.APELLIDO2
FROM DEPARTAMENTOS D, EMPLEADOS E
WHERE D.JEFE(+) = E.DNI;
```

Si queremos incluir aquellas filas que no tienen aún correspondencia con la tabla relacionada, tendremos que poner un signo más entre paréntesis:

- Delante del nombre de la tabla en la cláusula FROM.
- Delante del nombre del campo que relaciona donde sabemos que hay valores nulos.

- (●) **Detrás del nombre del campo que relaciona donde sabemos que hay valores nulos.**
- ( ) Delante del nombre del campo que relaciona donde sabemos que no hay valores nulos.

8.3.- Composiciones en la versión SQL99.

Como has visto, SQL incluye en esta versión mejoras de la sintaxis a la hora de crear composiciones en consultas. Recuerda que la sintaxis es la siguiente:

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

`CROSS JOIN`: creará un producto cartesiano de las filas de ambas tablas por lo que podemos olvidarnos de la cláusula `WHERE`.

`NATURAL JOIN`: detecta automáticamente las claves de unión, basándose en el nombre de la columna que coincide en ambas tablas. Por supuesto, se requerirá que las columnas de unión tengan el mismo nombre en cada tabla. Además, esta característica funcionará incluso si no están definidas las claves primarias o ajenas.

`JOIN USING`: las tablas pueden tener más de un campo para relacionar y no siempre queremos que se relacionen por todos los campos. Esta cláusula permite establecer relaciones indicando qué campo o campos comunes se quieren utilizar para ello.

`JOIN ON`: se utiliza para unir tablas en la que los nombres de columna no coinciden en ambas tablas o se necesita establecer asociaciones más complicadas.

`OUTER JOIN`: se puede eliminar el uso del signo (+) para composiciones externas utilizando un `OUTER JOIN`, de este modo resultará más fácil de entender.

`LEFT OUTER JOIN`: es una composición externa izquierda, todas las filas de la tabla de la izquierda se devuelven, aunque no haya ninguna columna correspondiente en las tablas combinadas.

`RIGTH OUTER JOIN`: es una composición externa derecha, todas las filas de la tabla de la derecha se devuelven, aunque no haya ninguna columna correspondiente en las tablas combinadas.

`FULL OUTER JOIN`: es una composición externa en la que se devolverán todas las filas de los campos no relacionados de ambas tablas.

Podríamos transformar algunas de las consultas con las que hemos estado trabajando:

Queríamos obtener el historial laboral de los empleados incluyendo nombres y apellidos de los empleados, la fecha en que entraron a trabajar y la fecha de fin de trabajo si ya no continúa en la empresa. Es una consulta de composición interna, luego utilizaremos `JOIN ON`:

```
SELECT E.Nombre, E.Apellido1, E.Apellido2, H.Fecha_inicio, H.Fecha_fin
FROM EMPLEADOS E JOIN HISTORIAL_LABORAL H ON (H.Empleado_DNI= E.DNI);
```

Queríamos también, obtener un listado con los nombres de los distintos departamentos y sus jefes con sus datos personales. Ten en cuenta que deben aparecer todos los departamentos aunque no

tengan asignado ningún jefe. Aquí estamos ante una composición externa, luego podemos utilizar `OUTER JOIN`:

```
SELECT  D.NOMBRE_DPTO, D.JEFE, E.NOMBRE, E.APELLIDO1, E.APELLIDO2
FROM DEPARTAMENTOS D LEFT OUTER JOIN EMPLEADOS E ON ( D.JEFE = E.DNI);
```

En MySQL también se utilizan las composiciones, aquí puedes verlo:
http://mysql.conclase.net/curso/?cap=012a#MUL_JOIN

9.- Otras consultas multitablas: Unión, Intersección y diferencia de consultas.

Caso práctico

Ana le cuenta a Carlos que ya tienen terminado casi todo el trabajo, pero que no le importa enseñarle otros tipos de consultas que no han necesitado utilizar en esta ocasión pero que es conveniente conocer, se refiere al uso de uniones, intersecciones y diferencia de consultas. Le explicará que es muy parecido a la teoría de conjuntos que recordará de haber terminado hace poco sus estudios.

Seguro que cuando empieces a trabajar con bases de datos llegará un momento en que dispongas de varias tablas con los mismos datos guardados para distintos registros y quieras unirla en una única tabla. ¿Esto se puede hacer? Es una operación muy común junto a otras. Al fin y al cabo, una consulta da como resultado un conjunto de filas y con conjuntos podemos hacer entre otras, tres tipos de operaciones comunes como son: unión, intersección y diferencia.

`UNION`: combina las filas de un primer `SELECT` con las filas de otro `SELECT`, desapareciendo las filas duplicadas.

`INTERSECT`: examina las filas de dos `SELECT` y devolverá aquellas que aparezcan en ambos conjuntos. Las filas duplicadas se eliminarán.

`MINUS`: devuelve aquellas filas que están en el primer `SELECT` pero no en el segundo. Las filas duplicadas del primer `SELECT` se reducirán a una antes de comenzar la comparación.

Para estas tres operaciones es muy **importante** que utilices en los dos `SELECT` el **mismo número** y tipo de **columnas** y en el **mismo orden**.

Estas operaciones se pueden combinar anidadas, pero es conveniente utilizar paréntesis para indicar que operación quieres que se haga primero.

Veamos un ejemplo de cada una de ellas.

`UNIÓN`**: Obtener los nombres y ciudades de todos los proveedores y clientes de Alemania.**

```
SELECT NombreCia, Ciudad FROM  PROVEEDORES WHERE Pais = 'Alemania'
UNION
SELECT NombreCia, Ciudad  FROM CLIENTES WHERE Pais = 'Alemania';
```

I

`NTERSECCIÓN`**: Una academia de idiomas da clases de inglés, frances y portugues; almacena los datos de los alumnos en tres tablas distintas una llamada "ingles", en una tabla denominada "frances" y los que aprenden portugues en la tabla "portugues". La academia necesita el nombre y domicilio de todos los alumnos que cursan los tres idiomas para enviarles iformación sobre los exámenes.**

```
SELECT nombre, domicilio FROM ingles INTERSECT
SELECT nombre, domicilio FROM frances INTERSECT
SELECT nombre, domicilio FROM portugues;
```

`DIFERENCIA`**: Ahora la academia necesita el nombre y domicilio solo de todos los alumnos que cursan inglés (no quiere a los que ya cursan portugués pues va a enviar publicidad referente al curso de portugués).**

```
SELECT nombre, domicilio FROM INGLES
MINUS
SELECT nombre,domicilio FROM PORTUGUES;
```

¿Cuáles de las siguientes afirmaciones son correctas?

- [x] La unión combina las filas de un primer SELECT con las filas de otro SELECT, desapareciendo las filas duplicadas.

- [x] **La diferencia devuelve aquellas filas que están en el primer SELECT pero no en el segundo. Correcta.**
- [x] **La intersección examina las filas de un SELECT y de otro y devolverá aquellas que aparezcan en ambos conjuntos.**
- [x] **En uniones, intersecciones y diferencias, los dos SELECT deben tener el mismo número pero no tienen por qué tener el mismo tipo de columnas.**

10.- Subconsultas.

Caso práctico

*— ¿Es posible consultar dentro de otra consulta? — pregunta **Carlos**.*

*Ha estado pensando que a veces va a necesitar filtrar los datos en función de un resultado que a priori desconoce. **Ana** se pone manos a la obra porque ve que ha llegado el momento de explicarle a **Carlos** las subconsultas.*

A veces tendrás que utilizar en una consulta los resultados de otra que llamaremos **subconsulta**. La sintaxis es:

```
SELECT listaExpr
FROM tabla
WHERE expresión OPERADOR
     ( SELECT listaExpr
     FROM tabla);
```

La subconsulta puede ir dentro de las cláusulas `WHERE`, `HAVING` o `FROM`.

El **OPERADOR** puede ser **>**, **<**, **>=**, **<=**, **!=**, **=** o **IN**. Las subconsultas que se utilizan con estos operadores devuelven un único valor, si la subconsulta devolviera más de un valor devolvería un error.

Como puedes ver en la sintaxis, las subconsultas deben ir entre paréntesis y a la derecha del operador.

Pongamos un ejemplo:

```
SELECT Nombre_empleado, sueldo
FROM EMPLEADOS
WHERE SUELDO <
     (SELECT SUELDO FROM EMPLEADOS
     WHERE Nombre_emple = 'Ana');
```

Obtendríamos el nombre de los empleados y el sueldo de aquellos que cobran menos que Ana.

Los tipos de datos que devuelven la subconsulta y la columna con la que se compara ha de ser el mismo.

¿Qué hacemos si queremos comparar un valor con varios, es decir, si queremos que la subconsulta devuelva más de un valor y comparar el campo que tenemos con dichos valores? Imagina que queremos ver si el sueldo de un empleado que es administrativo es mayor o igual que el sueldo medio de otros puestos en la empresa. Para saberlo deberíamos calcular el sueldo medio de las demás ocupaciones que tiene la empresa y éstos compararlos con la de nuestro empleado. Como ves, el resultado de la subconsulta es más de una fila. ¿Qué hacemos?

Cuando el resultado de la subconsulta es más de una fila, SQL utiliza **instrucciones especiales** entre el operador y la consulta. Estas instrucciones son:

- ✓ `ANY`. Compara con cualquier fila de la consulta. La instrucción es válida si hay un registro en la subconsulta que permite que la comparación sea cierta.
- ✓ `ALL`. Compara con todas las filas de la consulta. La instrucción resultará cierta si es cierta toda la comparación con los registros de la subconsulta.
- ✓ `IN`. No utiliza comparador, lo que hace es comprobar si el valor se encuentra en el resultado de la subconsulta.
- ✓ `NOT IN`. Comprueba si un valor no se encuentra en una subconsulta.

En la siguiente consulta obtenemos el empleado que menos cobra:

```
SELECT nombre, sueldo
FROM EMPLEADOS
WHERE sueldo <= ALL (SELECT sueldo FROM EMPLEADOS);
```

Relaciona cada instrucción con su función:

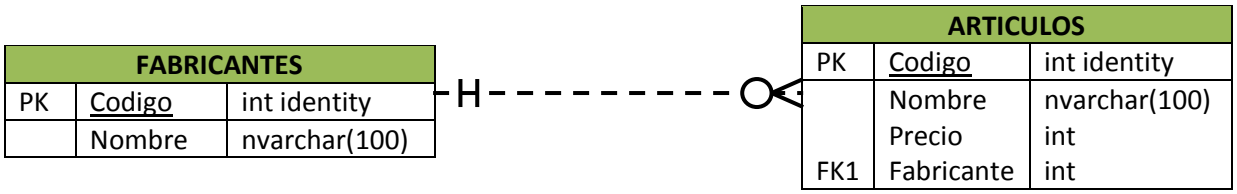
Instrucción.	Relación.	Función.
ANY	**1**	1. Compara con cualquier fila de la consulta.
ALL	**3**	2. Comprueba si el valor se encuentra en el resultado de la subconsulta.
IN	**2**	3. Compara con todas las filas de la consulta.
OT IN	**4**	4. Comprueba si un valor no se encuentra en una subconsulta.

¿Quieres más ejemplos con los que practicar?

http://superalumnos.net/ejercicios_resueltos_de_sql

Varios ejercicios SQL resueltos

La tienda de informática



1. Obtener los nombres de los productos de la tienda

```
SELECT Nombre FROM ARTICULOS
```

2. Obtener los nombres y los precios de los productos de la tienda

```
SELECT Nombre, Precio FROM ARTICULOS
```

3. Obtener el nombre de los productos cuyo precio sea menor o igual a 200€

```
SELECT Nombre FROM ARTICULOS WHERE Precio > 200
```

4. Obtener todos los datos de los artículos cuyo precio esté entre los 60€ y los 120€ (ambas cantidades incluidas)

```
/* Con AND */
SELECT * FROM ARTICULOS
   WHERE Precio >= 60 AND Precio <= 120

/* Con BETWEEN */
SELECT * FROM ARTICULOS
   WHERE Precio BETWEEN 60 AND 120
```

5. Obtener el nombre y el precio en pesetas (es decir, el precio en euros multiplicado por 166,386)

```
/* Sin AS */
SELECT Nombre, Precio * 166.386 FRO ARTICULOS

/* Con AS */
SELECT Nombre, Precio * 166.386 AS PrecioPtas FROM ARTICULOS
```

6. Seleccionar el precio medio de todos los productos

```
SELECT AVG(Precio) FROM ARTICULOS
```

7. Obtener el precio medio de los artículos cuyo código de fabricante sea 2

```
SELECT AVG(Precio) FROM ARTICULOS WHERE Fabricante=2
```

8. Obtener el número de artículos cuyo precio sea mayor o igual a 180€

```
SELECT COUNT(*) FROM ARTICULOS WHERE Precio >= 180
```

9. Obtener el nombre y precio de los artículos cuyo precio sea mayor o igual a 180€ y ordenarlos descendentemente por precio, y luego ascendentemente por nombre

```
SELECT Nombre, Precio FROM ARTICULOS
   WHERE Precio >= 180
   ORDER BY Precio DESC, Nombre
```

10. Obtener un listado completo de artículos, incluyendo por cada artículo los datos del artículo y de su fabricante

```
/* Sin INNER JOIN */
SELECT * FROM ARTICULOS, FABRICANTES
   WHERE ARTICULOS.Fabricante = FABRICANTES.Codigo

/* Con INNER JOIN */
```

```
SELECT *
   FROM ARTICULOS INNER JOIN FABRICANTES
   ON ARTICULOS.Fabricante = FABRICANTES.Codigo
```

11. Obtener un listado de artículos, incluyendo el nombre del artículo, su precio, y el nombre de su fabricante

```
/* Sin INNER JOIN */
SELECT ARTICULOS.Nombre, Precio, FABRICANTES.Nombre
   FROM ARTICULOS, FABRICANTES
   WHERE ARTICULOS.Fabricante = FABRICANTES.Codigo

/* Con INNER JOIN */
SELECT ARTICULOS.Nombre, Precio, FABRICANTES.Nombre
   FROM ARTICULOS INNER JOIN FABRICANTES
   ON ARTICULOS.Fabricante = FABRICANTES.Codigo
```

12. Obtener el precio medio de los productos de cada fabricante, mostrando solo los códigos de fabricante

```
SELECT AVG(Precio), Fabricante FROM ARTICULOS
   GROUP BY Fabricante
```

13. Obtener el precio medio de los productos de cada fabricante, mostrando el nombre del fabricante

```
/* Sin INNER JOIN */
SELECT AVG(Precio), FABRICANTES.Nombre
   FROM ARTICULOS, FABRICANTE
   WHERE ARTICULOS.Fabricante = FABRICANTES.Codigo
   GROUP BY FABRICANTES.Nombre

/* Con INNER JOIN */
SELECT AVG(Precio), FABRICANTES.Nombre
   FROM ARTICULOS INNER JOIN FABRICANTES
   ON ARTICULOS.Fabricante = FABRICANTES.Codigo
   GROUP BY FABRICANTES.Nombre
```

14. Obtener los nombres de los fabricantes que ofrezcan productos cuyo precio medio sea mayor o igual a 150€

```
/* Sin INNER JOIN */
SELECT AVG(Precio), FABRICANTES.Nombre
   FROM ARTICULOS, FABRICANTES
   WHERE ARTICULOS.Fabricante = FABRICANTES.Codigo
   GROUP BY FABRICANTES.Nombre
   HAVING AVG(Precio) >= 150

/* Con INNER JOIN */
SELECT AVG(Precio), FABRICANTES.Nombre
   FROM ARTICULOS INNER JOIN FABRICANTES
   ON ARTICULOS.Fabricante = FABRICANTES.Codigo
   GROUP BY FABRICANTES.Nombre
   HAVING AVG(Precio) >= 150
```

15. Obtener el nombre y precio del artículo más barato

```
SELECT Nombre, Precio
   FROM ARTICULOS
   WHERE Precio = (SELECT MIN(Precio) FROM ARTICULOS)
```

16. Obtener una lista con el nombre y precio de los artículos más caros de cada proveedor (incluyendo el nombre del proveedor)

```
/* Sin INNER JOIN */
SELECT A.Nombre, A.Precio, F.Nombre
   FROM ARTICULOS A, FABRICANTES F
   WHERE A.Fabricante = F.Codigo
      AND A.Precio =
      (
         SELECT MAX(A.Precio)
            FROM ARTICULOS A
```

```
            WHERE A.Fabricante = F.Codigo
      )

/* Con INNER JOIN */
SELECT A.Nombre, A.Precio, F.Nombre
   FROM ARTICULOS A INNER JOIN FABRICANTES F
   ON A.Fabricante = F.Codigo
      AND A.Precio =
      (
         SELECT MAX(A.Precio)
            FROM ARTICULOS A
            WHERE A.Fabricante = F.Codigo
      )
```

17. Añadir un nuevo producto: Altavoces de 70€ (del fabricante 2)

```
INSERT INTO ARTICULOS(Nombre, Precio, Fabricante)
   VALUES ( 'Altavoces', 70, 2 )
```

18. Cambiar el nombre del producto 8 a 'Impresora Laser'

```
UPDATE ARTICULOS
   SET Nombre = 'Impresora Laser'
   WHERE Codigo = 8
```

19. Aplicar un descuento del 10% (multiplicar el precio por 0,9) a todos los productos

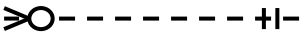
```
UPDATE ARTICULOS
   SET Precio = Precio * 0.9
```

20. Aplicar un descuento de 10€ a todos los productos cuyo precio sea mayor o igual a 120€

```
UPDATE ARTICULOS
   SET Precio = Precio - 10
   WHERE Precio >= 120
```

Empleados

EMPLEADOS		
PK	DNI	varchar(8)
	Nombre	nvarchar(100)
	Apellidos	nvarchar(255)
FK1	Departamento	int



DEPARTAMENTOS		
PK	Codigo	int
	Nombre	nvarchar(100)
	Presupuesto	int

1. Obtener los apellidos de los empleados

```
SELECT Apellidos FROM EMPLEADOS
```

2. Obtener los apellidos de los empleados sin repeticiones

```
SELECT DISTINCT Apellidos FROM EMPLEADOS
```

3. Obtener todos los datos de los empleados que se apellidan 'López'

```
SELECT * FROM EMPLEADOS WHERE Apellidos = 'López'
```

4. Obtener todos los datos de los empleados que se apellidan 'López' y los que se apellidan 'Pérez'

```
/* Con OR */
SELECT * FROM EMPLEADOS
   WHERE Apellidos = 'López' OR Apellidos = 'Pérez'

/* Con IN */
SELECT * FROM EMPLEADOS
   WHERE Apellidos IN ('López' , 'Pérez')
```

5. Obtener todos los datos de los empleados que trabajan para el departamento 14

```
SELECT * FROM EMPLEADOS WHERE Departamento = 14
```

6. Obtener todos los datos de los empleados que trabajan para el departamento 37 y para el departamento 77

```
/* Con OR */
SELECT * FROM EMPLEADOS
   WHERE Departamento = 37 OR Departamento = 77

/* Con IN */
SELECT * FROM EMPLEADOS
   WHERE Departamento IN (37,77)
```

7. Obtener todos los datos de los empleados cuyo apellido comience por 'P'

```
SELECT * FROM EMPLEADOS
   WHERE Apellidos LIKE 'P%'
```

8. Obtener el presupuesto total de todos los departamentos

```
SELECT SUM(Presupuesto) FROM DEPARTAMENTOS
```

9. Obtener el número de empleados en cada departamento

```
SELECT Departamento, COUNT(*)
   FROM EMPLEADOS
   GROUP BY Departamento
```

10. Obtener un listado completo de empleados, incluyendo por cada empleado los datos del empleado y de su departamento

```
SELECT *
   FROM EMPLEADOS INNER JOIN DEPARTAMENTOS
   ON EMPLEADOS.Departamento = DEPARTAMENTOS.Codigo
```

11. Obtener un listado completo de empleados, incluyendo el nombre y apellidos del empleado junto al nombre y presupuesto de su departamento.

```
/* Sin etiquetas */
SELECT EMPLEADOS.Nombre, Apellidos, DEPARTAMENTOS.Nombre, Prespuesto
   FROM EMPLEADOS INNER JOIN DEPARTAMENTOS
   ON EMPLEADOS.Departamento = DEPARTAMENTOS.Codigo

/* Con etiquetas */
SELECT E.Nombre, Apellidos, D.Nombre, Presupuesto
   FROM EMPLEADOS E INNER JOIN DEPARTAMENTOS D
   ON E.Departamento = D.Codigo
```

12. Obtener los nombres y apellidos de los empleados que trabajan en departamentos cuyo presupuesto sea mayor de 60.000€

```
/* Sin subconsulta */
SELECT EMPLEADOS.Nombre, Apellidos
   FROM EMPLEADOS INNER JOIN DEPARTAMENTOS
   ON EMPLEADOS.Departamento = DEPARTAMENTOS.Codigo
      AND DEPARTAMENTOS.Presupuesto > 60000

/* Con subconsulta */
SELECT Nombre, Apellidos FROM EMPLEADOS
   WHERE Departamento IN
   (SELECT Codigo FROM DEPARTAMENTOS WHERE Presupuesto > 60000)
```

13. Obtener los datos de los departamentos cuyo presupuesto es superior al presupuesto medio de todos los departamentos

```
SELECT * FROM DEPARTAMENTOS
   WHERE Presupuesto >
   (
      SELECT AVG(Presupuesto)
      FROM DEPARTAMENTOS
```

```
   )
```

14. Obtener los nombres (únicamente los nombres) de los departamentos que tienen más de dos empleados

```
/* Con subconsulta */
SELECT Nombre FROM DEPARTAMENTOS
   WHERE Codigo IN
   (
      SELECT Departamento
         FROM EMPLEADOS
         GROUP BY Departamento
         HAVING COUNT(*) > 2
   )

/* Con UNION. No funciona si dos departamentos tienen el mismo nombre */
SELECT DEPARTAMENTOS.Nombre
   FROM EMPLEADOS INNER JOIN DEPARTAMENTOS
   ON Departamento = Codigo
   GROUP BY DEPARTAMENTOS.Nombre
   HAVING COUNT(*) > 2
```

15. Añadir un nuevo departamento: 'Calidad', con presupuesto de 40.000€ y código 11. Añadir un empleado vinculado al departamento recién creado: Esther Vázquez, DNI: 89267109

```
INSERT INTO DEPARTAMENTOS
   VALUES(11,'Calidad',40000)

INSERT INTO EMPLEADOS
   VALUES('89267109','Esther','Vázquez',11)
```

16. Aplicar un recorte presupuestario del 10% a todos los departamentos

```
UPDATE DEPARTAMENTOS SET Presupuesto = Presupuesto * 0.9
```

17. Reasignar a los empleados del departamento de investigación (código 77) al departamento de informática (código 14)

```
UPDATE EMPLEADOS SET Departamento = 14 WHERE Departamento = 77
```

18. Despedir a todos los empleados que trabajan para el departamento de informática (código 14)

```
DELETE FROM EMPLEADOS
   WHERE Departamento = 14
```

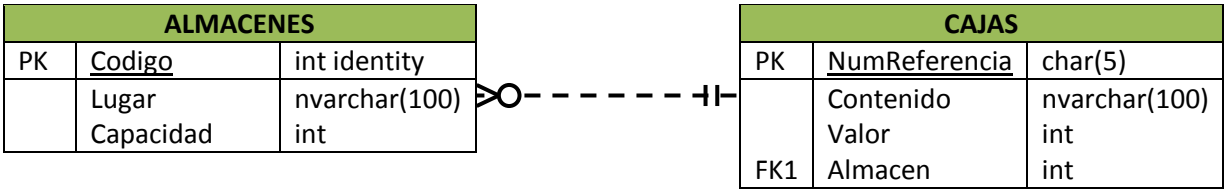
19. Despedir a todos los empleados que trabajen para departamentos cuyo presupuesto sea superior a los 60.000€

```
DELETE FROM EMPLEADOS
   WHERE Departamento IN
   (
      SELECT Codigo FROM DEPARTAMENTO
         WHERE Presupuesto >= 60000
   )
```

20. Despedir a todos los empleados

```
DELETE FROM EMPLEADOS
```

Los Almacenes



1. Obtener todos los almacenes

```
SELECT * FROM ALMACENES
```

2. Obtener todas las cajas cuyo contenido tenga un valor superior a 150€

```
SELECT * FROM CAJAS WHERE Valor > 150
```

3. Obtener los tipos de contenidos de las cajas

```
SELECT DISTINCT Contenido FROM CAJAS
```

4. Obtener el valor medio de todas las cajas

```
SELECT AVG(Valor) FROM CAJAS
```

5. Obtener el valor medio de las cajas de cada almacén

```
SELECT Almacen, AVG(Valor)
   FROM CAJAS
   GROUP BY Almacen
```

6. Obtener los códigos de los almacenes en los cuales el valor medio de las cajas sea superior a 150€

```
SELECT Almacen, AVG(Valor)
   FROM CAJAS
   GROUP BY Almacen
   HAVING AVG(Valor > 150
```

7. Obtener el número de referencia de cada caja junto con el nombre de la ciudad en la que se encuentra.

```
SELECT NumReferencia, Lugar
   FROM ALMACENES INNER JOIN CAJAS
   ON ALMACENES.Codigo = CAJAS.Almacen
```

8. Obtener el número de cajas que hay en cada almacén

```
/* Esta consulta no tiene en cuenta los almacenes vacíos */
SELECT Almacen, COUNT(*)
   FROM CAJAS
   GROUP BY Almacen

/* Esta consulta tiene en cuenta los almacenes vacíos */
SELECT Codigo, COUNT(NumReferencia)
   FROM ALMACENES LEFT JOIN CAJAS
   ON ALMACENES.Codigo = CAJAS.Almacen
   GROUP BY Codigo
```

9. Obtener los códigos de los almacenes que están saturados (los almacenes donde el número de cajas es superior a la capacidad)

```
SELECT Codigo
   FROM ALMACENES
   WHERE Capacidad <
   (
      SELECT COUNT(*)
         FROM CAJAS
         WHERE Almacen = Codigo
   )
```

10. Obtener los números de referencia de las cajas que están en Bilbao

```
/* Sin subconsultas */
SELECT NumReferencia
   FROM ALMACENES LEFT JOIN CAJAS
   ON ALMACENES.Codigo = CAJAS.Almacen
   WHERE Lugar = 'Bilbao'

/* Con subconsultas */
SELECT NumReferencia
   FROM CAJAS
   WHERE Almacen IN
   (
      SELECT Codigo
         FROM ALMACENES
         WHERE Lugar = 'Bilbao'
   )
```

11. Insertar un nuevo almacén en Barcelona con capacidad para 3 cajas

```
INSERT INTO ALMACENES(Lugar,Capacidad) VALUES('Barcelona',3)
```

12. Insertar una nueva caja, con número de referencia 'H5RT', con contenido 'Papel, valor 200, y situada en el almacén 2

```
INSERT INTO CAJAS
   VALUES('H5RT','Papel',200,2)
```

13. Rebajar el valor de todas las cajas un 15%

```
UPDATE CAJAS SET Valor = Valor * 0.85
```

14. Rebajar un 20% el valor de todas las cajas cuyo valor sea superior al valor medio de todas las cajas

```
UPDATE CAJAS SET Valor = Valor * 0.80
   WHERE Valor > (SELECT AVG(Valor) FROM CAJAS)
```

15. Eliminar todas las cajas cuyo valor sea inferior a 100€

```
DELETE FROM CAJAS WHERE Valor < 100
```

16. Vaciar el contenido de los almacenes que están saturados

```
DELETE FROM CAJAS WHERE Almacen IN
   (
      SELECT Codigo
         FROM ALMACENES
         WHERE Capacidad <
         (
            SELECT COUNT(*)
               FROM CAJAS
                  WHERE Almacen = Codigo
         )
   )
```

Películas y Salas

SALAS		
PK	Codigo	int identity
	Nombre	nvarchar(100)
FK1	Película	int

PELICULAS		
PK	Codigo	int identity
	Nombre	nvarchar(100)
	CalificacionEdad	int

1. Mostrar el nombre de todas las películas

```
SELECT Nombre FROM PELICULAS
```

2. Mostrar las distintas calificaciones de edad que existen

```
SELECT DISTINCT CalificacionEdad FROM PELICULAS
```

3. Mostrar todas las películas que no han sido calificadas

```
SELECT * FROM PELICULAS WHERE CalificacionEdad IS NULL
```

4. Mostrar todas las salas que no proyectan ninguna película

```
SELECT * FROM SALAS WHERE Pelicula IS NULL
```

5. Mostrar la información de todas las salas y, si se proyecta alguna película en la sala, mostrar también la información de la película

```
SELECT *
   FROM SALAS LEFT JOIN PELICULAS
   ON SALAS.Pelicula = PELICULAS,Codigo
```

6. Mostrar la información de todas las películas y, si se proyecta en alguna sala, mostrar también la información de la sala

```
SELECT *
   FROM SALAS RIGHT JOIN PELICULAS
   ON SALAS.Pelicula = PELICULAS.Codigo
```

7. Mostrar los nombres de las películas que no se proyectan en ninguna sala

```
/* Con JOIN */
SELECT PELICULAS.Nombre
   FROM SALAS RIGHT JOIN PELICULAS
   ON SALAS.Pelicula = PELICULAS.Codigo
   WHERE SALAS.Pelicula IS NULL

/* Con Subconsulta */
SELECT Nombre FROM PELICULAS
   WHERE Codigo NOT IN
   (
      SELECT Pelicula FROM SALAS
      WHERE Pelicula IS NOT NULL
   )
```

8. Añadir una nueva película 'Uno, Dos, Tres', para mayores de 7 años

```
INSERT INTO PELICULAS(Nombre,CalificacionEdad) VALUES('Uno, Dos, Tres', 7)
```

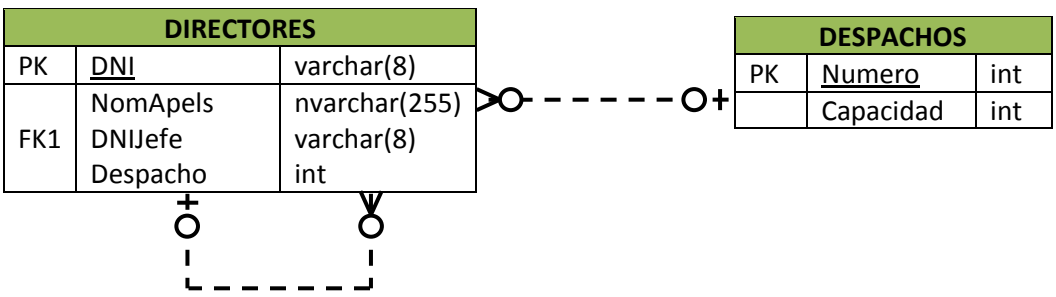
9. Hacer constar que todas las películas no calificadas han sido calificadas 'no recomendables para menores de 13 años'

```
UPDATE PELICULAS SET CalificacionEdad=13 WHERE CalificacionEdad IS NULL
```

10. Eliminar todas las salas que proyectan películas recomendadas para todos los públicos

```
DELETE FROM SALAS WHERE Pelicula IN
   (SELECT Codigo FROM PELICULAS WHERE CalificacionEdad = 0)
```

Los Directores



1. Mostrar el DNI, nombre y apellidos de todos los directores

```
SELECT DNI, NomApels FROM DIRECTORES
```

2. Mostrar los datos de los directores que no tienen jefes

```
SELECT * FROM DIRECTORES WHERE DNIJefe IS NULL
```

3. Mostrar el nombre y apellidos de cada director, junto con la capacidad del despacho en el que se encuentra

```
SELECT NomApels, Despacho, Capacidad
   FROM DIRECTORES INNER JOIN DESPACHOS
   ON DIRECTORES.Despacho = DESPACHOS.Numero
```

4. Mostrar el número de directores que hay en cada despacho

```
/* Sin tener en cuenta despachos vacíos */
SELECT Despacho, COUNT(*)
   FROM DIRECTORES
   GROUP BY Despacho

/* Teniendo en cuenta despachos vacíos */
SELECT Numero, COUNT(DNI)
   FROM DESPACHOS LEFT JOIN DIRECTORES
   ON DESPACHOS.Numero = DIRECTORES.Despacho
   GROUP BY Numero
```

5. Mostrar los datos de los directores cuyos jefes no tienen jefes

```
SELECT * FROM DIRECTORES
   WHERE DNIJefe IN
   (SELECT DNI FROM DIRECTORES WHERE DNIJefe IS NULL)
```

6. Mostrar los nombres y apellidos de los directores junto con los de su jefe

```
/* Con INNER JOIN. No muestra directores que no tienen jefes */
SELECT d1.NomApels, d2.NomApels
   FROM DIRECTORES d1 INNER JOIN DIRECTORES d2
   ON d1.DNIJefe = d2.DNI

/* Con LEFT JOIN. Si muestra directores sin jefe */
SELECT d1.NomApels, d2.NomApels
   FROM DIRECTORES d1 LEFT JOIN DIRECTORES d2
   ON d1.DNIJefe = d2.DNI
```

7. Mostrar el número de despachos que están sobreutilizados

```
SELECT Numero FROM DESPACHOS
   WHERE Capacidad <
   (
      SELECT COUNT(*)
      FROM DIRECTORES
      WHERE Despacho = Numero
   )
```

8. Añadir un nuevo director llamado Paco Pérez, DNI 28301700, sin jefe, y situado en el despacho 124

```
INSERT INTO DIRECTORES VALUES('28301700', 'Paco Pérez', NULL, 124)
```

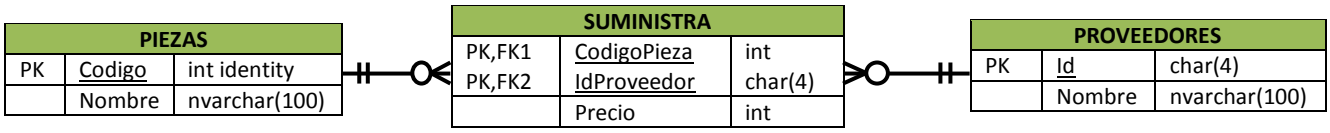
9. Asignar a todos los empleados apellidados Pérez un nuevo jefe con DNI 74568521

```
UPDATE DIRECTORES SET DNIJefe = '74568521' WHERE NomApels LIKE '%Pérez%'
```

10. Despedir a todos los directores, excepto a los que no tienen jefe

```
DELETE FROM DIRECTORES WHERE DNIJefe IS NOT NULL
```

Piezas y Proveedor



1. Obtener los nombres de todas las piezas

```
SELECT Nombre FROM PIEZAS
```

2. Obtener todos los datos de todos los proveedores

```
SELECT * FROM PROVEEDORES
```

3. Obtener el precio medio al que se nos suministran las piezas

```
   SELECT CodigoPieza, AVG(Precio)
      FROM SUMINISTRA
      GROUP BY CodigoPieza
```

4. Obtener los nombres de los proveedores que suministran la pieza 1

```
/* Sin subconsulta */
SELECT PROVEEDORES.Nombre
   FROM PROVEEDORES INNER JOIN SUMINISTRA
   ON PROVEEDORES.Id = SUMINISTRA.IdProveedor
      AND SUMINISTRA.CodigoPieza = 1

/* Con subconsulta */
SELECT Nombre
   FROM PROVEEDORES
   WHERE Id IN
   (SELECT IdProveedor FROM SUMINISTRA WHERE CodigoPieza = 1)
```

5. Obtener los nombres de las piezas suministradas por el proveedor coyo código es HAL

```
/* Sin subconsulta */
SELECT PIEZAS.Nombre
   FROM PIEZAS INNER JOIN SUMINISTRA
   ON PIEZAS.Codigo = SUMINISTRA.CodigoPieza
      AND SUMINISTRA.IdProveedor = 'HAL'

/* Con subconsultas IN */
SELECT Nombre
   FROM PIEZAS
   WHERE Codigo IN
      (SELECT CodigoPieza FROM SUMINISTRA WHERE IdProveedor = 'HAL')

/* Con subconsulta EXISTS */
SELECT Nombre
   FROM PIEZAS
   WHERE EXISTS
   (
      SELECT * FROM SUMINISTRA
         WHERE IdProveedor = 'HAL'
            AND CodigoPieza = PIEZAS.Codigo
   )
```

6. Obtener los nombres de los proveedores que suministran las piezas más caras indicando el nombre de la pieza y el precio al que la suministran

```
SELECT p1.Nombre, pr1.Nombre, Precio
   FROM PIEZAS p1 INNER JOIN
      (SUMINISTRA s1 INNER JOIN PROVEEDORES pr1
      ON s1.IdProveedor = pr1.Id)
   ON p1.Codigo = s1.CodigoPieza
   WHERE Precio IN
   (
      SELECT MAX(Precio) FROM SUMINISTRA s2
      GROUP BY s2.CodigoPieza
      HAVING s2.CodigoPieza = p1.Codigo
```

```
   )
```

7. Hacer constar en la base de datos que la empresa “Skellington Supplies” (código TNBC) va a empezar a suministrarnos tuercas (códio 1) a 7 pesetas cada tuerca.

```
INSERT INTO SUMINISTRA VALUES('TNBC',1,7)
```

8. Aumentar los precios en una unidad

```
UPDATE SUMINISTRA SET Precio = Precio + 1
```

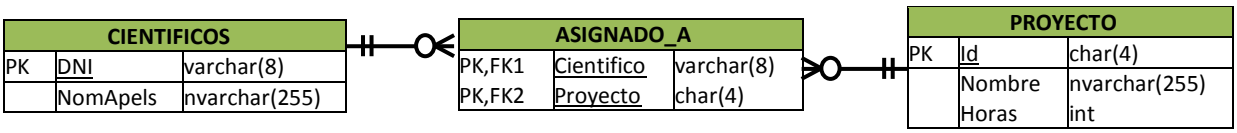
9. Hacer constar en la base de datos que la empresa “Susan Calvin Corp” (RBT) no va a suministrarnos ninguna pieza (aunque la empresa en sí va a seguir constando en nuestra base de datos)

```
DELETE FROM SUMINISTRA WHERE IdProveedor = 'RBT'
```

10. Hacer constar en la base de datos que la empresa “Susan Calvin Corp.” (RBT) ya no va a suministrarnos clavos (código 4)

```
DELETE FROM SUMINISTRA
   WHERE IdProveedor = 'RBT'
      AND CodigoPieza = 4
```

Los Científicos



1. Sacar una relación completa de los científicos asignados a cada proyecto. Mostrar DNI, Nombre del científico, Identificador del proyecto y nombre del proyecto

```
/* Sin JOIN */
SELECT DNI, NomApels, Id, Nombre
   FROM CIENTIFICOS C, ASIGNADO_A A, PROYECTO P
   WHERE C.DNI = A.Cientifico
   AND A.Proyecto = P.Id

/* Con JOIN */
SELECT DNI, NomApels, Id, Nombre
   FROM CIENTIFICOS C INNER JOIN
   (ASIGNADO_A A INNER JOIN PROYECTO P
   ON A.Proyecto = P.Id)
   ON C.DNI = A.Cientifico
```

2. Obtener el número de proyectos al que está asignado cada científico (mostrar el DNI y el nombre)

```
SELECT DNI,NomApels, COUNT(Proyecto)
   FROM CIENTIFICOS LEFT JOIN ASIGNADO_A
   ON CIENTIFICOS.DNI = ASIGNADO_A.Cientifico
   GROUP BY DNI, NomApels
```

3. Obtener el número de científicos asignados a cada proyecto (mostrar el identificador de proyecto y el nombre del proyecto)

```
SELECT Id, Nombre, COUNT(Proyecto)
   FROM PROYECTO LEFT JOIN ASIGNADO A
   ON PROYECTO.Id = ASIGNADO A.Proyecto
   GROUP BY Id, Nombre
```

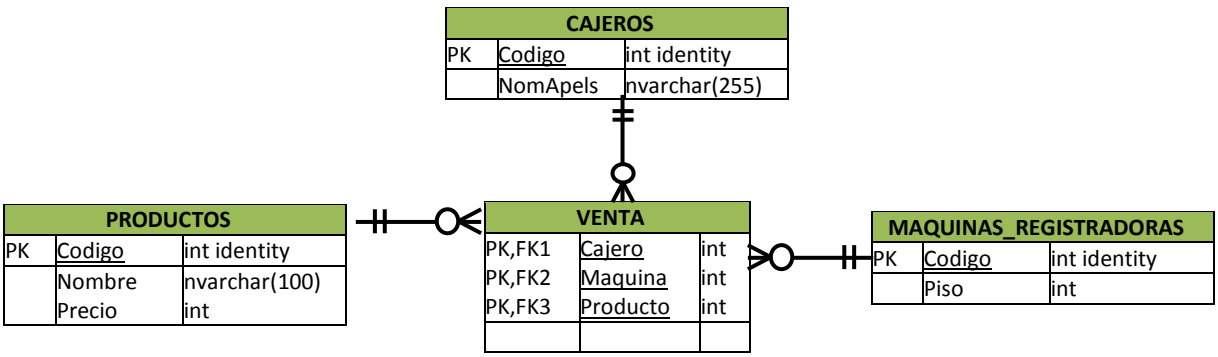
4. Obtener el número de horas de dedicación de cada científico

```
SELECT DNI,NomApels, SUM(Horas)
   FROM CIENTIFICOS C LEFT JOIN
   (ASIGNADO A A INNER JOIN PROYECTO P
   ON A.Proyecto = P.Id)
   ON C.DNI = A.Cientifico
   GROUP BY DNI,NomApels
```

5. Obtener el DNI y nombre de los científicos que se dedican a más de un proyecto y cuya dedicación media a cada proyecto sea superior a las 80 horas

```
/* Con dos subconsultas */
SELECT DNI, NomApels
   FROM CIENTIFICOS C
   WHERE 1 <
   (
      SELECT COUNT(*) FROM ASIGNADO_A
      WHERE Cientifico = C.DNI
   )
   AND 80 <
   (
      SELECT AVG(Horas)
      FROM PROYECTO INNER JOIN ASIGNADO A
      ON PROYECTO.Id = ASIGNADO A.Proyecto
      WHERE Cientifico = C.DNI
   )

/* Juntando tablas y con HAVING */
SELECT DNI, NomApels
   FROM CIENTIFICOS C, ASIGNADO A A, PROYECTO P
   WHERE C.DNI = A.Cientifico
   AND P.Id = A.Proyecto
   GROUP BY DNI, NomApels
   HAVING COUNT(Proyecto) > 1 and AVG(Horas) > 80
```

Grandes Almacenes



1. Mostrar el número de ventas de cada producto, ordenado de más a menos ventas

```
SELECT Codigo, Nombre, COUNT(VENTA.Producto)
   FROM PRODUCTOS LEFT JOIN VENTA
   ON PRODUCTOS.Codigo = VENTA.Producto
   GROUP BY Codigo, Nombre
   ORDER BY COUNT(VENTA.Producto) DESC
```

2. Obtener un informe completo de ventas, indicando el nombre del cajero que realizó la venta, nombre y precios de los productos vendidos, y piso en el que se encuentra la máquina registradora donde se realizó la venta

```
/* Sin JOIN */
SELECT NomApels, Nombre, Precio, Piso
   FROM VENTA V, CAJEROS C, PRODUCTOS P, MAQUINAS REGISTRADORAS M
   WHERE V.Cajero = C.Codigo
   AND V.Producto = P.Codigo
```

```
   AND V.Maquina = M.Codigo

/* Con JOIN */
SELECT NomApels, Nombre, Precio, Piso
   FROM CAJEROS C INNER JOIN
      (PRODUCTOS P INNER JOIN
         (MAQUINAS_REGISTRADORAS M INNER JOIN VENTA V
         ON V.Maquina = M.Codigo)
      ON V.Producto = P.Codigo)
   ON V.Cajero = C.Codigo

Obtener las ventas totales realizadas en cada piso
SELECT Piso, SUM(Precio)
   FROM VENTA V, PRODUCTOS P, MAQUINAS REGISTRADORAS M
   WHERE V.Producto = P.Codigo
   AND V.Maquina = M.Codigo
   GROUP BY Piso
```

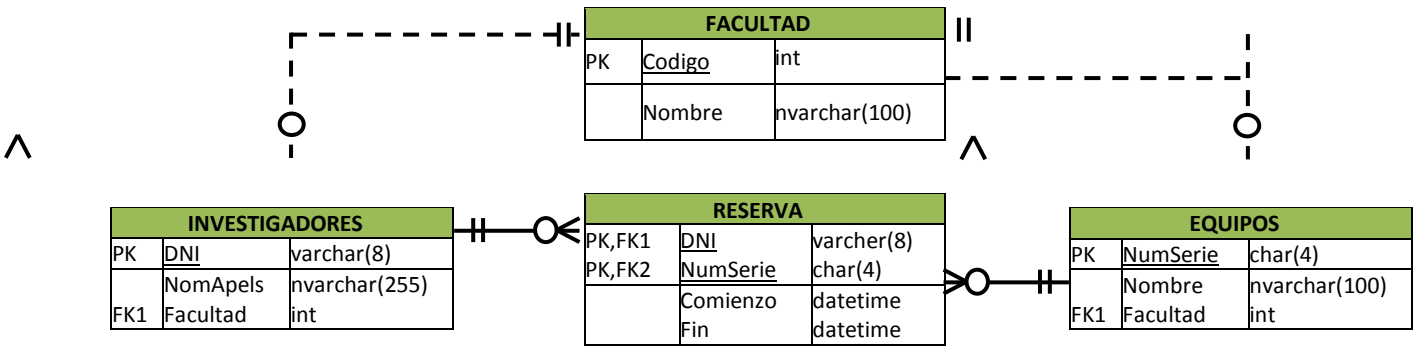
3. Obtener el código y nombre de cada empleado junto con el importe total de sus ventas

```
SELECT C.Codigo, C.NomApels, SUM(Precio)
   FROM PRODUCTOS P INNER JOIN
      (CAJEROS C LEFT JOIN VENTA V
         ON V.Cajero = C.Codigo)
   ON V.Producto = P.Codigo
   GROUP BY C.Codigo, NomApels
```

4. Obtener el código y nombre de aquellos cajeros que hayan realizado ventas en pisos cuyas ventas totales sean inferiores a los 500€

```
SELECT Codigo, NomApels FROM CAJEROS
   WHERE Codigo IN
   (
      SELECT Cajero FROM VENTA
         WHERE Maquina IN
         (
            SELECT Codigo FROM MAQUINAS REGISTRADORAS
               WHERE Piso IN
               (
                  SELECT Piso
                  FROM VENTA V, PRODUCTOS P, MAQUINAS_REGISTRADORAS M
                  WHERE V.Producto = P.Codigo
                  AND V.Maquina = M.Codigo
                  GROUP BY Piso
                  HAVING SUM(Precio) < 500
               )
         )
   )
```

Los investigadores



1. Obtener el DNI y nombre de aquellos investigadores que han realizado más de una reserva

```
/* Juntando tablas */
```

```
SELECT I.DNI, NomApels
   FROM INVESTIGADORES I LEFT JOIN RESERVA R
   ON R.DNI = I.DNI
   GROUP BY I.DNI, NomApels
   HAVING COUNT(R.DNI) > 1

/* Con subconsulta */
SELECT DNI, NomApels
   FROM INVESTIGADORES
   WHERE DNI IN
   (
      SELECT DNI FROM RESERVA
      GROUP BY DNI
      HAVING COUNT(*) > 1
   )
```

2. Obtener un listado completo de reservas, incluyendo los siguientes datos:
   - ✓ DNI y nombre del investigador, junto con el nombre de su facultad
   - ✓ Numero de serie y nombre del equipo reservado, junto con el nombre de la facultad a la que pertenece
   - ✓ Fecha de comienzo y fin de la reserva

```
SELECT I.DNI, NomApels, F INV.Nombre, E.NumSerie, E.Nombre, F EQUIP.Nombre, Comienzo, Fin
   FROM RESERVA R, INVESTIGADORES I, EQUIPOS E, FACULTAD F INV, FACULTAD F EQUIP
   WHERE R.DNI = I.DNI
   AND R.NumSerie = E.NumSerie
   AND I.Facultad = F INV.Codigo
   AND E.Facultad = F_EQUIP.Codigo
```

3. Obtener el DNI y el nombre de los investigadores que han reservado equipos que no son de su facultad

```
/* Juntando tablas */
SELECT DISTINCT I.DNI,NomApels
   FROM RESERVA R, INVESTIGADORES I, EQUIPOS E
   WHERE R.DNI = I.DNI
   AND R.NumSerie = E.NumSerie
   AND I.Facultad <> E.Facultad

/* Con EXISTS */
SELECT DNI, NomApels
   FROM INVESTIGADORES I
   WHERE EXISTS
   (
      SELECT * FROM RESERVA R INNER JOIN EQUIPOS E
         ON R.NumSerie = E.NumSerie
         WHERE R.DNI = I.DNI
         AND I.Facultad <> E.Facultad
   )
```

4. Obtener los nombres de las facultades en las que ningún investigador ha realizado una reserva

```
SELECT Nombre FROM FACULTAD
   WHERE Codigo IN
   (
      SELECT Facultad FROM INVESTIGADORES I LEFT JOIN RESERVA R
         ON I.DNI = R.DNI
         GROUP BY Facultad
         HAVING COUNT(R.DNI) = 0
   )
```

5. Obtener los nombres de las facultades con investigadores ‘ociosos’ (investigadores que no han realizado ninguna reserva)

```
SELECT Nombre FROM FACULTAD
   WHERE Codigo IN
   (
      SELECT Facultad FROM INVESTIGADORES
      WHERE DNI NOT IN
      (
         SELECT DNI FROM RESERVA
      )
```

```
   )
```

6. Obtener el número de serie y nombre de los equipos que nunca han sido reservados

```
/* Juntando tablas */
SELECT E.NumSerie, Nombre
   FROM EQUIPOS E LEFT JOIN RESERVA R
   ON R.NumSerie = E.NumSerie
   GROUP BY E.NumSerie, Nombre
   HAVING COUNT(R.NumSerie) = 0

/* Con subconsult IN */
SELECT NumSerie, Nombre FROM EQUIPOS
   WHERE NumSerie NOT IN
   (
      SELECT NumSerie FROM RESERVA
   )

/* Con EXISTS */
SELECT NumSerie, Nombre
   FROM EQUIPOS E
   WHERE NOT EXISTS
   (
      SELECT * FROM RESERVA R
      WHERE R.NumSerie = E.NumSerie
   )
```

TEMA 5

INDICE

1.- Introducción. 3
2.- Edición de la información mediante herramientas gráficas. 4
2.1.- Inserción de registros. 4
2.2.- Modificación de registros. 5
2.3.- Borrado de registros. 5
3.- Edición de la información mediante sentencias SQL. 7
3.1.- Inserción de registros. 7
3.2.- Modificación de registros. 8
3.3.- Borrado de registros. 9
4.- Integridad referencial. 10
4.1.- Integridad en actualización y supresión de registros. 10
4.2.- Supresión en cascada. 11
5.- Subconsultas y composiciones en órdenes de edición. 13
5.1.- Inserción de registros a partir de una consulta. 13
5.2.- Modificación de registros a partir de una consulta. 14
5.3.- Supresión de registros a partir de una consulta. 14
6.- Transacciones. 16
6.1.- Hacer cambios permanentes. 16
6.2.- Deshacer cambios. 17
6.3.- Deshacer cambios parcialmente. 18
7.- Problemas asociados al acceso simultáneo a los datos. 19
Integridad: Control de concurrencia. 19
Introducción 19
Problemas clásicos de concurrencia: 20
Técnicas de Bloqueo. 20
Bloqueo. Variable cerrojo 20
Tipos: 21
Asegura la seriabilidad 21
Prevención el interbloqueo: 21
Marcas de Tiempo 22
Marcas de tiempo multiversión 22
Modelo Multiversión de ORACLE: 23
7.1.- Políticas de bloqueo. 23
7.2.- Bloqueos compartidos y exclusivos. 24
7.3.- Bloqueos automáticos. 24
7.4.- Bloqueos manuales. 25

Tratamiento de datos.

Caso práctico
Ada le ha preguntado a Juan sobre el estado actual del proyecto y él le comenta que está empezando el desarrollo de la aplicación y va a empezar a desarrollar una serie de procesos en los que se deberá almacenar la información que debe manejar la aplicación, así como modificarla o eliminar los datos que así lo requieran.

Estas acciones de tratamiento de la información deberán asegurar que no se obtengas resultados incorrectos, por errores en la ejecución de la aplicación o por las acciones de los usuarios, y además debe asegurar que los datos puedan ser accesibles por varios usuarios simultáneamente.

La aplicación requiere que se puedan dar de alta nuevos usuarios en la base de datos, así como juegos y partidas. Además se podrá modificar en un determinado momento la información personal de los usuarios, de los juegos, o añadir nuevos usuarios a las partidas. También asegurará la posibilidad de suprimir cualquiera de esos datos.

Se debe asegurar que, por ejemplo, una partida no haga referencia a usuario que han sido eliminado, o a juegos que no existen. Un usuario podrá ver reducido su crédito en un determinado momento, y la nueva información de su crédito sólo deberá ser accesible cuando haya finalizado el proceso de reducción del crédito, y no mientras se realiza esa actualización, ya que el crédito disponible no estará actualizado.

Por supuesto, al ser una aplicación online, distintos usuarios podrán realizar operaciones simultáneamente, como crear partidas al mismo tiempo.

1.- Introducción.

Caso práctico

Juan le pregunta a Ana, la alumna que se encuentra en prácticas, qué mecanismos conoce para poder manipular los datos que deben encontrarse en una base de datos, de manera que se puedan añadir nuevos datos, modificarlos o eliminarlos. Ella recuerda que estudió una serie de sentencias o comandos del lenguaje SQL que permiten realizar todas esas operaciones, y que además, desde el entorno visual de la base de datos Oracle también se pueden realizar todas esas acciones de manera más cómoda para el usuario, pero menos flexible.

Las bases de datos no tienen razón de ser sin la posibilidad de hacer operaciones para el tratamiento de la información almacenada en ellas. Por operaciones de tratamiento de datos se deben entender las acciones que permiten añadir información en ellas, modificarla o bien suprimirla.

En esta unidad podrás conocer que existen distintos medios para realizar el tratamiento de los datos. Desde la utilización de herramientas gráficas hasta el uso de instrucciones o sentencias del lenguaje SQL que permiten realizar ese tipo de operaciones de una forma menos visual pero con más detalle, flexibilidad y rapidez. El uso de unos mecanismos u otros dependerá de los medios disponibles y de nuestras necesidades como usuarios de la base de datos.

Pero la información no se puede almacenar en la base de datos sin tener en cuenta que debe seguir una serie de requisitos en las relaciones existentes entre las tablas que la componen. Todas las operaciones que se realicen respecto al tratamiento de los datos deben asegurar que las relaciones existentes entre ellos se cumplan correctamente en todo momento.

Por otro lado, la ejecución de las aplicaciones puede fallar en un momento dado y eso no debe impedir que la información almacenada sea incorrecta. O incluso el mismo usuario de las aplicaciones debe tener la posibilidad de cancelar una determinada operación y dicha cancelación no debe suponer un problema para que los datos almacenados se encuentren en un estado fiable.

Todo esto requiere disponer de una serie de herramientas que aseguren esa fiabilidad de la información, y que además puede ser consultada y manipulada en sistemas multiusuario sin que las acciones realizadas por un determinado usuario afecte negativamente a las operaciones de los demás usuarios.

2.- Edición de la información mediante herramientas gráficas.

Caso práctico

Ana ha recibido el encargo de que introduzca una serie de datos en las tablas de la base de datos para poder realizar varias pruebas de su funcionamiento. No son muchos registros los que tiene que introducir, así que va a utilizar una herramienta gráfica que le ofrece el sistema gestor de base de datos que van a utilizar. Ella podría hacerlo escribiendo una serie de instrucciones que ha aprendido durante sus estudios, pero como no son muchos los registros que debe introducir, ha optado por utilizar la herramienta gráfica, ya que facilita el tratamiento de los datos para casos sencillos.

Los sistemas gestores de bases de datos como el de Oracle, pueden ofrecer mecanismos para la manipulación de la información contenida en las bases de datos. Principalmente se dividen en herramientas gráficas y herramientas en modo texto (también reciben el nombre de terminal, consola o línea de comandos).

Para realizar el tratamiento de los datos por línea de comandos se requiere la utilización de un lenguaje de base de datos como SQL, lo cual implica el conocimiento de dicho lenguaje.

En cambio, si se dispone de herramientas gráficas para la manipulación de los datos, no es imprescindible conocer las sentencias de un lenguaje de ese tipo, y permite la introducción, edición y borrado de datos desde un entorno gráfico con la posibilidad de uso de ratón y una ventana que facilita esas operaciones con un uso similar a las aplicaciones informáticas a las que estamos acostumbrados como usuarios.

La base de datos Oracle ofrece en su distribución Oracle Database Express la herramienta Application Express a la que puedes acceder en Windows desde `Inicio` > `Todos los programas` > `Base de Datos Oracle Express Edition` > `Ir a Página Inicial de Base de Datos`.

La única manera de realizar el tratamiento de datos en una base de datos es a través de una herramienta gráfica.

(•) Falso ( ) Verdadero

Se pueden encontrar otras herramientas en modo texto en las que la manipulación de los datos se hace a través de una serie de comandos.

2.1.- Inserción de registros.

La inserción de registros permite introducir nuevos datos en las tablas que componen la base de datos. Para insertar registros en tablas, utilizando las herramientas gráficas que ofrece Oracle Database Express, se deben seguir los siguientes **pasos**:

1. Ir a la **página inicial** de bases de datos de Oracle Database Express, si no te encuentras en ella.
2. Hacer clic en el botón ***Explorador de objetos***.

3. **Seleccionar una tabla** en la lista izquierda.
4. Seleccionar la pestaña ***Datos*** y hacer clic en el botón ***Insertar Fila***.

5. **Escribir los datos** correspondientes para cada campo del nuevo registro.
6. Hacer clic en el botón ***Crear*** para guardar los datos introducidos, o ***Crear*** y ***Crear Otro*** si se desea seguir añadiendo otro registro nuevo. Se utilizará el botón *Cancelar* si no se desea guardar los datos.

7. Si la fila se ha **añadido correctamente** se mostrará el mensaje correspondiente.

Permaneciendo en la vista de la pestaña *Datos* se podrá ver, en la parte central, la **lista de los datos** contenidos en los registros que se han ido insertando.

En caso de que se haya producido un **error** al intentar insertar los datos, habrá que comprobar el mensaje que se muestra, e intentar solucionar el problema. Por ejemplo, si se intenta introducir un texto en un campo de tipo numérico se obtendrá un error como el siguiente: "*error ORA-00984: columna no permitida aquí*", y no se habrá realizado ninguna operación de la inserción del nuevo registro.

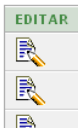
En el enlace *Show Me and Try it* puedes abrir una animación en la que se demuestra el proceso de inserción de registros en una base de datos de Oracle Express utilizando su herramienta gráfica.
http://st-curriculum.oracle.com/tutorial/DBXETutorial/html/module5/les05_ins_tab10_show_me.htm

2.2.- Modificación de registros.

Para insertar registros en tablas, utilizando las herramientas gráficas que ofrece Oracle *Database Express*, se deben seguir los siguientes **pasos**:

1. Ir a la **página inicial** de bases de datos de Oracle Database Express, si no te encuentras en ella.
2. Hacer clic en el botón ***Explorador de objetos***.
3. **Seleccionar una tabla** en la lista izquierda.
4. Seleccionar la pestaña ***Datos***.

5. Debe aparecer, bajo los botones anteriores, una **lista** con los registros que previamente se hayan insertado en la tabla. En el lado izquierdo de cada registro aparece el **icono** que permite la modificación de los datos del registro que se encuentra en la misma fila.



6. Tras hacer clic en el icono, se muestran los campos que forman el registro con los datos que contiene actualmente. Para **modificar cualquier dato** simplemente debemos escribirlo en el campo correspondiente. Así habrá que modificar todos los datos necesarios.
7. Para aceptar los cambios realizados, se debe hacer clic en el botón ***Aplicar Cambios***, pero si se desea volver al estado anterior, simplemente hay que utilizar el botón *Cancelar*.
8. Si todo ha ido bien aparecerá un mensaje informando que los cambios se han aplicado.

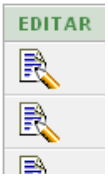
Los cambios efectuados se pueden **comprobar** en la lista de registros mostrada en la parte inferior de la ventana.

Al igual que se comentó en la inserción de registros, al aceptar los cambios realizados en los datos, éstos se comprobarán automáticamente para ver si cumplen con los requisitos establecidos en la tabla. En caso de que no se cumplan, aparecerá un mensaje informando del error que se ha producido. Una vez solucionado el problema se podrá volver a intentar aplicar los cambios efectuados.

En el enlace *Show Me and Try it* puedes abrir una animación en la que se demuestra el proceso de modificación de registros en una base de datos de Oracle Express utilizando su herramienta gráfica.
http://st-curriculum.oracle.com/tutorial/DBXETutorial/html/module5/les05_upd_rows10_show_me.htm

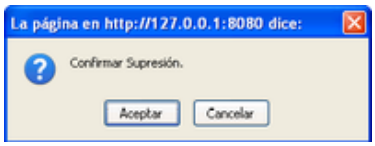
2.3.- Borrado de registros.

En el caso de que quieras eliminar un registro de una determinada tabla hay que seguir, en principio, los **mismos pasos** que se han comentado anteriormente para **editar un registro**. Es decir, una vez seleccionada la tabla, elegir la pestaña Datos y hacer clic en el botón *Editar* junto al registro que quieres suprimir.



Cuando se muestra la información del contenido del registro, puedes observar en la parte superior derecha que dispones de un botón ***Suprimir***, junto al que has podido utilizar en el apartado anterior para *Aplicar Cambios*.

Al hacer clic en ese botón verás una ventana de diálogo donde solicita que confirmes si deseas borrar el registro.



Si todo ha ido bien se mostrará un mensaje informando de ello:

La eliminación de un registro no podrá realizarse si un registro de otra tabla hace referencia a él. En ese caso, se mostrará el mensaje correspondiente al intentar eliminarlo (similar a: "`error ORA-02292: restricción de integridad violada - registro secundario encontrado`"). Si ocurriera esto, para eliminar el registro se debe eliminar el registro que hace referencia a él, o bien modificarlo para que haga referencia a otro registro.

Para eliminar un registro se debe utilizar:

- ○ Botón Suprimir que se encuentra en la lista de datos junto a cada registro.
- ◉ **Botón Editar junto al registro, y luego el botón Suprimir.**
- ○ Botón derecho del ratón y seleccionar la opción Suprimir del menú desplegado.
- ○ Seleccionar el registro y pulsar la tecla Suprimir.

En el enlace *Show Me and Try it* puedes abrir una animación en la que se demuestra el proceso de eliminación de registros en una base de datos de Oracle Express utilizando su herramienta gráfica.

http://st-curriculum.oracle.com/tutorial/DBXETutorial/html/module5/les05_del_rows10_show_me.htm

3.- Edición de la información mediante sentencias SQL.

Caso práctico

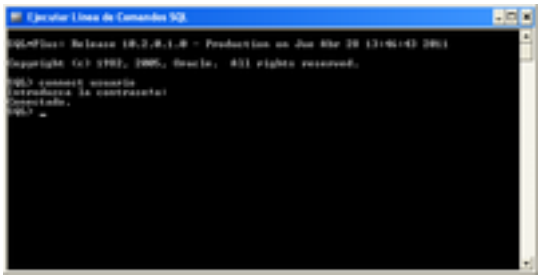
La aplicación web de juegos online que está desarrollando Juan, debe acceder a la base de datos desde su código fuente para recoger datos almacenados en ella. La herramienta gráfica que ha estado utilizando Ana para introducir datos no puede usarse para que la aplicación que está desarrollando realice esa misma operación. Tendrá que utilizar, desde el código fuente de la aplicación, sentencias del lenguaje SQL que permiten la manipulación de los datos. Por ello, va a practicar con Ana el uso de esas sentencias SQL desde las aplicaciones de Oracle Express, para más adelante implementarlas en el código fuente de la aplicación.

El lenguaje SQL dispone de una serie de sentencias para la edición (inserción, actualización y borrado) de los datos almacenados en una base de datos. Ese conjunto de sentencias recibe el nombre de *Data Manipulation Language* (DML).

Las sentencias SQL que se verán a continuación pueden ser ejecutadas desde el entorno web *Application Express* de Oracle utilizando el botón SQL en la página de inicio, y desplegando su lista desplegable elegir `Comandos SQL` > `Introducir Comando`.

También se pueden indicar las sentencias SQL desde el entorno de *SQL*Plus* que ofrece Oracle y que puedes encontrar en `Inicio` > `Todos los programas` > `Base de Datos Oracle Express Edition` > `Ejecutar Línea de Comandos SQL.`

Si optas por abrir esa aplicación (*Ejecutar Línea de Comandos SQL*), el primer paso que debe realizarse para manipular los datos de una determinada tabla, es conectarse utilizando un nombre de usuario con los permisos necesarios para hacer ese tipo de operaciones a la tabla deseada. Utiliza para ello la orden `CONNECT` seguida del nombre de usuario. Seguidamente, solicitará la contraseña correspondiente a dicho usuario.



Para ejecutar cualquiera de las sentencias SQL que aprenderás en los siguientes puntos, simplemente debes escribirla completa y pulsar *`Intro`* para que se inicie su ejecución.

3.1.- Inserción de registros.

La sentencia *`INSERT`* permite la inserción de nuevas filas o registros en una tabla existente.

El formato más sencillo de utilización de la sentencia `INSERT` tiene la siguiente sintaxis:

```
INSERT INTO nombre_tabla (lista_campos) VALUES (lista_valores);
```

Donde *nombre_tabla* será el nombre de la tabla en la que quieras añadir nuevos registros. En lista_campos se indicarán los campos de dicha tabla en los que se desea escribir los nuevos valores indicados en *lista_valores*. Es posible omitir la lista de campos (*lista_campos*), si se indican todos los valores de cada campo y en el orden en el que se encuentran en la tabla.

Tanto la lista de campos *lista_campos* como la de valores *lista_valores*, tendrán separados por comas cada uno de sus elementos. Hay que tener en cuenta también que cada campo de *lista_campos* debe tener un valor válido en la posición correspondiente de la *lista_valores* (Si no recuerdas los valores

válidos para cada campo puedes utilizar la sentencia `DESCRIBE` seguida del nombre de la tabla que deseas consultar).

En el siguiente ejemplo se inserta un nuevo registro en la tabla *USUARIOS* en el que se tienen todos los datos disponibles:

```
INSERT INTO USUARIOS (Login, Password, Nombre, Apellidos, Direccion, CP,
Localidad, Provincia, Pais, F_Nacimiento,
F_Ingreo, Correo, Credito, Sexo) VALUES ('migrod86', '6PX5=V', 'MIGUEL
ANGEL', 'RODRIGUEZ RODRIGUEZ', 'ARCO DEL LADRILLO,PASEO',
'47001', 'VALLADOLID', 'VALLADOLID', 'ESPAÑA', '27/04/1977', '10/01/2008',
'migrod86@gmail.com', 200, 'H');
```

En este otro ejemplo, se inserta un registro de igual manera, pero sin disponer de todos los datos:

```
INSERT INTO USUARIOS (Login, Password, Nombre, Apellidos, Correo) VALUES
('natsan63',
'VBROMI', 'NATALIA', 'SANCHEZ GARCIA', 'natsan63@hotmail.com');
```

Al hacer un `INSERT` en el que no se especifiquen los valores de todos los campos, se obtendrá el valor `NULL` en aquellos campos que no se han indicado.

Si la lista de campos indicados no se corresponde con la lista de valores, se obtendrá un error en la ejecución. Por ejemplo, si no se indica el campo Apellidos pero sí se especifica un valor para dicho campo:

```
INSERT INTO USUARIOS (Login, Password, Nombre, Correo) VALUES ('caysan56',
'W4IN5U', 'CAYETANO', 'SANCHEZ CIRIZA', 'caysan56@gmail.com');
```

Se obtiene el siguiente error:

```
ORA-00913: demasiados valores
```

¿Cuál de las siguientes sentencias INSERT es correcta?

- ○ `INSERT INTO PRODUCTOS (Codigo, Nombre, Existencias) VALUES (3, Leche, 100);`
- ○ `INSERT INTO PRODUCTOS (3, 'Leche', 100);`
- ◉ **`INSERT INTO PRODUCTOS (Codigo, Nombre, Existencias) VALUES (3, 'Leche', 100);`**
- ○ `INSERT INTO PRODUCTOS (Codigo, Nombre, Existencias) VALUES ('Leche', 3, 100);`

3.2.- Modificación de registros.

La sentencia *`UPDATE`* permite modificar una serie de valores de determinados registros de las tablas de la base de datos.

La manera más sencilla de utilizar la sentencia `UPDATE` tiene la siguiente sintaxis:

```
UPDATE nombre_tabla SET nombre_campo = valor [, nombre_ campo = valor]...
[ WHERE condición ];
```

Donde *nombre_tabla* será el nombre de la tabla en la que quieras modificar datos. Se pueden especificar los nombres de campos que se deseen de la tabla indicada. A cada campo especificado se le debe asociar el nuevo valor utilizando el signo **=**. Cada emparejamiento `campo=valor` debe separarse del siguiente utilizando comas (**,**).

La cláusula `WHERE` seguida de la condición es opcional (como pretenden indicar los corchetes). Si se indica, la actualización de los datos sólo afectará a los registros que cumplen la condición. Por tanto, ten en cuenta que si no indicas la cláusula `WHERE`, los cambios afectarán a todos los registros.

Por ejemplo, si se desea poner a 200 el crédito de todos los usuarios:

```
UPDATE USUARIOS SET Credito = 200;
```

En este otro ejemplo puedes ver la actualización de dos campos, poniendo a 0 el crédito y borrando la información del campo *Pais* de todos los usuarios:

```
UPDATE USUARIOS SET Credito = 0, Pais = NULL;
```

Para que los cambios afecten a determinados registros hay que especificar una condición. Por ejemplo, si se quiere cambiar el crédito de todas las mujeres, estableciendo el valor 300:

```
UPDATE USUARIOS SET Credito = 300 WHERE Sexo = 'M';
```

Cuando termina la ejecución de una sentencia `UPDATE`, se muestra la cantidad de registros (filas) que han sido actualizadas, o el error correspondiente si se ha producido algún problema. Por ejemplo podríamos encontrarnos con un mensaje similar al siguiente:

```
45 fila(s) actualizada(s).
```

En el enlace *Show Me and Try it* puedes abrir una animación en la que se demuestra el proceso de modificación de registros en una base de datos de Oracle Express utilizando el lenguaje SQL.
http://st-curriculum.oracle.com/tutorial/DBXETutorial/html/module5/les05_upd_rows20_show_me.htm

3.3.- Borrado de registros.

La sentencia `DELETE` es la que permite eliminar o borrar registros de una tabla.

Esta es la sintaxis que debes tener en cuenta para utilizarla:

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

Al igual que hemos visto en las sentencias anteriores, nombre_tabla hace referencia a la tabla sobre la que se hará la operación, en este caso de borrado. Se puede observar que la cláusula `WHERE` es opcional. Si no se indica, debes tener muy claro que se borrará todo el contenido de la tabla, aunque la tabla seguirá existiendo con la estructura que tenía hasta el momento. Por ejemplo, si usas la siguiente sentencia, borrarás todos los registros de la tabla *USUARIOS*:

```
DELETE FROM USUARIOS;
```

Para ver un ejemplo de uso de la sentencia `DELETE` en la que se indique una condición, supongamos que queremos eliminar todos los usuarios cuyo crédito es cero:

```
DELETE FROM USUARIOS WHERE Credito = 0;
```

Como resultado de la ejecución de este tipo de sentencia, se obtendrá un mensaje de error si se ha producido algún problema, o bien, el número de filas que se han eliminado.

```
45 fila(s) suprimida(s).
```

¿Si no se especifica una condición en la sentencia DELETE se borra todo el contenido de la tabla especificada?

(•) **Verdadero** ( ) Falso

Debes tener cuidado al usar la sentencia DELETE, porque si no se especifica qué datos se desea eliminar de la tabla, se eliminará todo su contenido

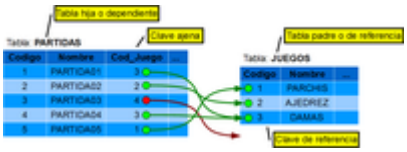
4.- Integridad referencial.

Caso práctico

Ana tiene una duda en el planteamiento de la base de datos del proyecto de la plataforma de juegos online y se la plantea a Juan: ¿Qué ocurre si el registro correspondiente a los datos de un determinado juego es eliminado y existen partidas creadas de dicho juego? Juan le responde que para eso existe la integridad referencial, que además asegurará otras cosas como por ejemplo que no puedan existir partidas que reflejen que su creador es un usuario que no existe en la base de datos.

Dos tablas pueden ser relacionadas entre ellas si tienen en común uno o más campos, que reciben el nombre de clave ajena. La restricción de integridad referencial requiere que haya coincidencia en todos los valores que deben tener en común ambas tablas. Cada valor del campo que forma parte de la integridad referencial definida, debe corresponderse, en la otra tabla, con otro registro que contenga el mismo valor en el campo referenciado.

Siguiendo con el ejemplo de juegos online, supongamos que en una determinada partida de un juego, se han unido una serie de usuarios. En la tabla de *PARTIDAS* existe un campo de referencia al tipo de juego al que corresponde, mediante su código de juego. Por tanto, no puede existir ninguna partida cuyo código de juego no se corresponda con ninguno de los juegos de la tabla *JUEGOS*.



En este ejemplo, no se cumple la integridad referencial, porque la partida "*PARTIDA03*" corresponde al juego cuyo código es 4, y en la tabla *JUEGOS* no existe ningún registro con ese código.

Para que se cumpla la integridad referencial, todos los valores del campo *Cod_Juego* deben corresponderse con valores existentes en el campo *Codigo* de la tabla *JUEGOS*.

Cuando se habla de integridad referencial se utilizan los siguientes términos:

- ✓ **Clave ajena**: Es el campo o conjunto de campos incluidos en la definición de la restricción que deben hacer referencia a una clave de referencia. En el ejemplo anterior, la clave ajena sería el campo *Cod_Juego* de la tabla *PARTIDAS*.
- ✓ **Clave de referencia**: Clave única o primaria de la tabla a la que se hace referencia desde una clave ajena. En el ejemplo, la clave de referencia es el campo *Codigo* de la tabla *JUEGOS*.
- ✓ **Tabla hija o dependiente**: Tabla que incluye la clave ajena, y que, por tanto, depende de los valores existentes en la clave de referencia. Correspondería a la tabla *PARTIDAS* del ejemplo, que sería la tabla hija de la tabla *JUEGOS*.
- ✓ **Tabla padre o de referencia**: Corresponde a la tabla que es referenciada por la clave ajena en la tabla hija. Esta tabla determina las inserciones o actualizaciones que son permitidas en la tabla hija, en función de dicha clave. En el ejemplo, la tabla *JUEGOS* es padre de la tabla *PARTIDAS*.

Descripción del concepto de integridad referencial con ejemplo.
http://es.wikipedia.org/wiki/Integridad_referencial

4.1.- Integridad en actualización y supresión de registros.

La relación existente entre la clave ajena y la clave padre tiene implicaciones en el borrado y modificación de sus valores.

Si se modifica el valor de la clave ajena en la tabla hija, debe establecerse un nuevo valor que haga referencia a la clave principal de uno de los registros de la tabla padre. De la misma manera, no se puede modificar el valor de la clave principal en un registro de la tabla padre, y una clave ajena hace referencia a dicho registro.

Los borrados de registros en la tabla de referencia también pueden suponer un problema, ya que no pueden suprimirse registros que son referenciados con una clave ajena desde otra tabla.

Suponiendo el siguiente ejemplo:

En el registro de la partida con nombre "*PARTIDA01*" no puede ser modificado el campo *Cod_Juego* al valor 4, porque no es una clave ajena válida, puesto que no existe un registro en la tabla *JUEGOS* con esa clave primaria.

El código del juego "*DAMAS*" no puede ser cambiado, ya que hay registros en la tabla *PARTIDAS* que hacen referencia a dicho juego a través del campo *Cod_Juego*.

Si se eliminara en la tabla *JUEGOS* el registro que contiene el juego "*PARCHIS*", la partida "*PARTIDA05*" quedaría con un valor inválido en el campo *Cod_Juego*.

Cuando se hace el borrado de registros en una tabla de referencia, se puede configurar la clave ajena de diversas maneras para que se conserve la integridad referencial:

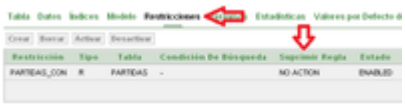
- ✓ **No Permitir Supresión**: Es la opción por defecto. En caso de que se intente borrar en la tabla de referencia un registro que está siendo referenciado desde otra tabla, se produce un error en la operación de borrado impidiendo dicha acción.
- ✓ **Supresión en Cascada**: Al suprimir registros de la tabla de referencia, los registros de la tabla hija que hacían referencia a dichos registros, también son borrados.
- ✓ **Definir Nulo en Suprimir**: Los valores de la clave ajena que hacían referencia a los registros que hayan sido borrados de la tabla de referencia, son cambiados al valor `NULL`.

Apuntes sobre integridad referencial en Oracle.
http://www.dsic.upv.es/asignaturas/facultad/lsi/trabajos/172000.doc

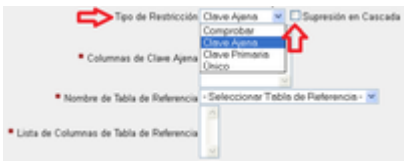
4.2.- Supresión en cascada.

Las opciones de ***Supresión en Cascada*** o ***Definir Nulo en Suprimir*** pueden establecerse desde el momento de creación de las tablas, en el tercer paso del asistente que ofrece ***Application Express de Oracle***, al establecer las claves ajenas.

Si la tabla ya estaba creada, y posteriormente se desea establecer una restricción de clave ajena con opción de ***Supresión en Cascada***, se puede establecer desde el ***Explorador de objetos de Application Express***, seleccionando la tabla que contiene el campo con la clave ajena. La pestaña ***Restricciones*** ofrece la posibilidad de crear, borrar, activar y desactivar restricciones de este tipo.



Si se está creando una restricción de clave ajena con supresión en cascada, tras usar el botón ***Crear*** anterior, hay que seleccionar la opción ***clave ajena*** en la lista desplegable, y marcar la opción **Supresión en Cascada**.



Si estas operaciones se quieren realizar con código SQL, se dispone de las siguientes opciones durante la declaración de la clave ajena de la tabla: utilizar la opción `ON DELETE CASCADE` para hacer la supresión en cascada, o bien `ON DELETE SET NULL` si se prefiere definir nulo en suprimir. Por ejemplo:

```
CONSTRAINT JUEGOS_CON FOREIGN KEY (Cod_Juego) REFERENCES JUEGO (Codigo) ON
DELETE CASCADE
```

Hay que recordar que una declaración de este tipo debe hacerse en el momento de crear la tabla (`CREATE TABLE`) o modificar su estructura (`ALTER TABLE`).

5.- Subconsultas y composiciones en órdenes de edición.

Caso práctico

Juan necesita un mecanismo que permita actualizaciones en la base de datos de forma masiva con una serie de condiciones que afectan no sólo a la tabla sobre la que debe hacer los cambios en los datos. Por ejemplo, va a implementar, en la aplicación que está desarrollando, una opción para que los usuarios que han creado partidas obtengan algunos beneficios en los créditos que disponen.

Para conseguir esto no le sirven las sentencias SQL simples que has podido ver en apartados anteriores. Deberá utilizarlas en unión con consultas que determinen los registros que han de ser modificados.

Anteriormente has podido conocer una serie de instrucciones del lenguaje SQL que han servido para realizar operaciones de inserción, modificación y eliminación de registros. Tal como las hemos analizado, esas operaciones se realizan sobre registros de una misma tabla, pero vamos a ver que esas mismas sentencias pueden utilizarse de una forma más avanzada insertando consultas dentro de esas mismas operaciones de tratamiento de datos.

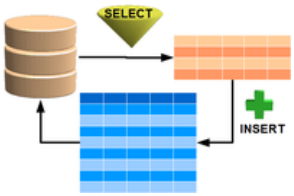
Por tanto, veremos que los resultados de las operaciones pueden afectar a más de una tabla, es decir, que con una misma instrucción se pueden añadir registros a más de una tabla, o bien actualizar o eliminar registros de varias tablas simultáneamente.

Los valores que se añadan o se modifiquen podrán ser obtenidos también como resultado de una consulta.

Además, las condiciones que hemos podido añadir hasta ahora a las sentencias, pueden ser también consultas, por lo que pueden establecerse condiciones bastante más complejas.

En este manual pueden encontrar una sección sobre las funciones agregadas y subconsultas (módulo 3). También puedes ver ejemplos en la parte final.
http://es.scribd.com/doc/56646934/SQL-BASICO-Material-de-Apoyo

5.1.- Inserción de registros a partir de una consulta.



Anteriormente hemos visto la posibilidad de insertar registros en una tabla a través de la sentencia `INSERT`, por ejemplo:

```
INSERT INTO USUARIOS (LOGIN, PASSWORD, NOMBRE, APELLIDOS, CORREO) VALUES
('natsan63'
, 'VBROMI', 'NATALIA', 'SANCHEZ GARCIA', 'natsan63@hotmail.com');
```

Esta misma acción se puede realizar usando una consulta `SELECT` dentro de la sentencia `INSERT`, así por ejemplo, la equivalente a la anterior sería:

```
INSERT INTO (SELECT LOGIN, PASSWORD, NOMBRE, APELLIDOS, CORREO FROM
USUARIOS) VALUES ('natsan63',
'VBROMI', 'NATALIA', 'SANCHEZ GARCIA', 'natsan63@hotmail.com');
```

Puedes observar que simplemente se ha sustituido el nombre de la tabla, junto con sus campos, por una consulta equivalente.

Y no sólo eso, sino que es posible insertar en una tabla valores que se obtienen directamente del resultado de una consulta. Supongamos por ejemplo, que disponemos de una tabla

USUARIOS_SIN_CREDITO con la misma estructura que la tabla ***USUARIOS***. Si queremos insertar en esa tabla todos los usuarios que tienen el crédito a cero:

```
INSERT INTO USUARIOS_SIN_CREDITO SELECT * FROM USUARIOS WHERE Credito = 0;
```

Observa que en ese caso no se debe especificar la palabra `VALUES`, ya que no se está especificando una lista de valores.

¿Cuál de las siguientes sentencias INSERT es la correcta para insertar en la tabla CLIENTES los nombres de los registros de la tabla NUEVOS_CLIENTES, suponiendo que los campos que contienen los nombres se llaman Nombre_CLI en la tabla CLIENTES y Nombre_NCLI en la tabla NUEVOS_CLIENTES?

- `INSERT INTO CLIENTES (Nombre CLI) VALUES Nombre NCLI FROM NUEVOS CLIENTES;`
- **`INSERT INTO CLIENTES (Nombre_CLI) VALUES (SELECT Nombre_NCLI FROM NUEVOS_CLIENTES);`**
- `INSERT INTO CLIENTES VALUES (SELECT Nombre CLI, Nombre NCLI FROM NUEVOS CLIENTES);`
- `INSERT INTO NUEVOS CLIENTES (Nombre CLI) VALUES (SELECT Nombre NCLI FROM CLIENTES);`

5.2.- Modificación de registros a partir de una consulta.

La acción de actualizar registros mediante la sentencia `UPDATE` también puede ser utilizada con consultas para realizar modificaciones más complejas de los datos. Las consultas pueden formar parte de cualquiera de los elementos de la sentencia `UPDATE`.

Por ejemplo, la siguiente sentencia modifica el crédito de aquellos usuarios que tienen una partida creada y cuyo estado es 1 (activada). El valor del crédito que se les asigna es el valor más alto de los créditos de todos los usuarios.

```
UPDATE USUARIOS SET Credito = (SELECT MAX(Credito) FROM
USUARIOS) WHERE Login IN (SELECT Cod_Crea FROM PARTIDAS WHERE Estado=1);
```

¿Cuál de las siguientes sentencias UPDATE es la correcta para actualizar en la tabla USUARIOS el crédito del usuario con código 3 para asignarle el mismo crédito que el del usuario con código 5?

- `UPDATE USUARIOS SET Credito = Credito WHERE Codigo = 3 AND WHERE Codigo = 5;`
- `UPDATE USUARIOS SET Credito = (SELECT Credito FROM USUARIOS WHERE Codigo = 3 AND WHERE Codigo = 5);`
- `UPDATE USUARIOS SET Codigo = 5 WHERE (SELECT Credito FROM USUARIOS WHERE Codigo = 3);`
- **`UPDATE USUARIOS SET Credito = (SELECT Credito FROM USUARIOS WHERE Codigo = 3) WHERE Codigo = 5;`**

5.3.- Supresión de registros a partir de una consulta.

Al igual que las sentencias `INSERT` y `UPDATE` vistas anteriormente, también se pueden hacer borrados de registros utilizando consultas como parte de las tablas donde se hará la eliminación o como parte de la condición que delimita la operación.

Por ejemplo, si se ejecuta la siguiente sentencia:

```
DELETE FROM (SELECT * FROM USUARIOS, PARTIDAS WHERE Login=Cod_Crea AND
Estado=0);
```

El resultado es que se eliminan determinados registros de las tablas ***USUARIOS*** y ***PARTIDAS***, en concreto, aquellos usuarios que han creado alguna partida cuyo estado es 0 (desactivada).
Puedes observar que no se ha establecido ninguna condición `WHERE` en la sentencia, ya que se ha incluido dentro de la consulta. Otra manera de realizar la misma acción, pero utilizando la cláusula `WHERE` es la siguiente:

```
DELETE FROM (SELECT * FROM USUARIOS, PARTIDAS WHERE Login=Cod_Crea) WHERE Estado=0;
```

¿Cuál de las siguientes sentencias **DELETE** es la correcta para eliminar de la tabla USUARIOS todos aquellos cuyo código se encuentra en una tabla llamada ANTIGUOS?

- (•) `DELETE FROM USUARIOS WHERE Codigo IN (SELECT Codigo FROM ANTIGUOS);`
- ( ) `DELETE FROM USUARIOS WHERE Codigo IN ANTIGUOS;`
- ( ) `DELETE FROM (SELECT Codigo FROM ANTIGUOS) WHERE Codigo IN (SELECT Codigo FROM USUARIOS);`
- ( ) `DELETE FROM Codigo WHERE USUARIOS IN (SELECT Codigo FROM ANTIGUOS);`

6.- Transacciones.

Caso práctico

Ana ha estado haciendo algunas pruebas del funcionamiento de la aplicación y ha observado un error: Con los créditos que dispone un determinado usuario ha empezado la creación de una nueva partida, pero antes de finalizar el proceso de creación de la partida ha utilizado un botón "Cancelar" para simular que el usuario ha optado por dar marcha atrás en la creación de la partida. En ese caso, el crédito del usuario debería permanecer inalterado, ya que no ha finalizado el proceso de creación de la partida, pero ha observado los datos que hay en la base de datos y se encuentra con que el crédito del usuario se ha decrementado.

Al comentarle el problema a Juan, éste le comenta que debe gestionar ese proceso utilizando transacciones.

Una transacción es una unidad atómica (no se puede dividir) de trabajo que contiene una o más sentencias SQL. Las transacciones agrupan sentencias SQL de tal manera que a todas ellas se le aplica una operación `COMMIT`, que podríamos traducir como aplicadas o guardadas en la base de datos, o bien a todas ellas se les aplica la acción `ROLLBACK`, que interpretamos como deshacer las operaciones que deberían hacer sobre la base de datos.

Mientras que sobre una transacción no se haga `COMMIT`, los resultados de ésta pueden deshacerse. El efecto de una sentencia del lenguaje de manipulación de datos (DML) no es permanente hasta que se hace la operación `COMMIT` sobre la transacción en la que esté incluida.

Las transacciones de Oracle cumplen con las propiedades básicas de las transacciones en base de datos:

- ✓ **Atomicidad**: Todas las tareas de una transacción son realizadas correctamente, o si no, no se realiza ninguna de ellas. No hay transacciones parciales. Por ejemplo, si una transacción actualiza 100 registros, pero el sistema falla tras realizar 20, entonces la base de datos deshace los cambios realizados a esos 20 registros.
- ✓ **Consistencia**: La transacción se inicia partiendo de un estado consistente de los datos y finaliza dejándola también con los datos consistentes.
- ✓ **Aislamiento**: El efecto de una transacción no es visible por otras transacciones hasta que finaliza.
- ✓ **Durabilidad**: Los cambios efectuados por las transacciones que han volcado sus modificaciones, se hacen permanentes.

Las sentencias de control de transacciones gestionan los cambios que realizan las sentencias DML y las agrupa en transacciones. Estas sentencias te permiten realizar las siguientes acciones:

- ✓ Hacer permanentes los cambios producidos por una transacción (`COMMIT`).
- ✓ Deshacer los cambios de una transacción (`ROLLBACK`) desde que fue iniciada o desde un punto de restauración (`ROLLBACK TO SAVEPOINT`). Un punto de restauración es un marcador que puedes establecer dentro del contexto de la transacción. Debes tener en cuenta que la sentencia *ROLLBACK* finaliza la transacción, pero `ROLLBACK TO SAVEPOINT` no la finaliza.
- ✓ Establecer un punto intermedio (`SAVEPOINT`) a partir del cual se podrá deshacer la transacción.
- ✓ Indicar propiedades para una transacción (`SET TRANSACTION`).
- ✓ Especificar si una restricción de integridad aplazable se comprueba después de cada sentencia DML o cuando se ha realizado el `COMMIT` de la transacción (`SET CONSTRAINT`).

6.1.- Hacer cambios permanentes.

Una transacción comienza cuando se encuentra la primera sentencia SQL ejecutable. Para que los cambios producidos durante la transacción se hagan permanentes (no puedan deshacerse), se dispone de las siguientes opciones:

- ✓ Utilizar la sentencia `COMMIT`, la cual ordena a la base de datos que haga permanentes las acciones incluidas en la transacción.

- ✓ Ejecutar una sentencia DDL (como `CREATE`, `DROP`, `RENAME`, o `ALTER`). La base de datos ejecuta implícitamente una orden `COMMIT` antes y después de cada sentencia DDL.
- ✓ Si el usuario cierra adecuadamente las aplicaciones de gestión de las bases de datos Oracle, se produce un volcado permanente de las cambios efectuados por la transacción.

Desde la aplicación gráfica ***Application Express***, la ejecución de sentencias SQL desde ***Inicio*** > ***SQL*** > ***Comandos SQL*** permite que se hagan los cambios permanentes tras su ejecución, marcando la opción ***Confirmación Automática***.

¿Si se cierra correctamente la aplicación gráfica después de haber realizado una operación de modificación de datos, y no se ha indicado la opción de Confirmación automática, ni se ha ejecutado la sentencia COMMIT, se quedan guardados los cambios efectuados por la transacción?

(•) **Verdadero** ( ) Falso

6.2.- Deshacer cambios.

La sentencia `ROLLBACK` permite deshacer los cambios efectuados por la transacción actual, dándola además por finalizada.

Siempre se recomienda que explícitamente finalices las transacciones en las aplicaciones usando las sentencias `COMMIT` o `ROLLBACK`.

Recuerda que si no se han hecho permanentes los cambios de una transacción, y la aplicación termina incorrectamente, la base de datos de Oracle retorna al estado de la última transacción volcada, deshaciendo los cambios de forma implícita.

Para deshacer los cambios de la transacción simplemente debes indicar:

```
ROLLBACK;
```

Hay que tener en cuenta que si una transacción termina de forma anormal, por ejemplo, por un fallo de ejecución, los cambios que hasta el momento hubiera realizado la transacción son deshechos de forma automática.

¿Se pueden deshacer los cambios con la sentencia ROLLBACK después de que se haya ejecutado COMMIT?

( ) Verdadero (•) **Falso**

Ejemplo sobre el uso de `Rollback`.

```
CREATE TABLE emp_name AS SELECT employee_id, last_name FROM employees;
CREATE UNIQUE INDEX empname_ix ON emp_name (employee_id);
CREATE TABLE emp_sal AS SELECT employee_id, salary FROM employees;
CREATE UNIQUE INDEX empsal_ix ON emp_sal (employee_id);
CREATE TABLE emp_job AS SELECT employee_id, job_id FROM employees;
CREATE UNIQUE INDEX empjobid_ix ON emp_job (employee_id);

DECLARE
   emp_id       NUMBER(6);
   emp_lastname VARCHAR2(25);
   emp_salary   NUMBER(8,2);
   emp_jobid    VARCHAR2(10);
BEGIN
   SELECT employee_id, last_name, salary, job_id INTO emp_id, emp_lastname,
     emp_salary, emp_jobid FROM employees WHERE employee_id = 120;
   INSERT INTO emp_name VALUES (emp_id, emp_lastname);
```

```
   INSERT INTO emp_sal VALUES (emp_id, emp_salary);
   INSERT INTO emp_job VALUES (emp_id, emp_jobid);
EXCEPTION
   WHEN DUP VAL ON INDEX THEN
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Inserts have been rolled back');
END;
/
```

6.3.- Deshacer cambios parcialmente.

Un punto de restauración (`SAVEPOINT`) es un marcador intermedio declarado por el usuario en el contexto de una transacción. Los puntos de restauración dividen una transacción grande en pequeñas partes.

Si usas puntos de restauración en una transacción larga, tendrás la opción de deshacer los cambios efectuados por la transacción antes de la sentencia actual en la que se encuentre, pero después del punto de restauración establecido. Así, si se produce un error, no es necesario rehacer todas las sentencias de la transacción completa, sino sólo aquellos posteriores al punto de restauración.

Para establecer un punto de restauración se utiliza la sentencia `SAVEPOINT` con la sintaxis:

```
SAVEPOINT nombre_punto_restauración;
```

La restauración de los cambios hasta ese punto se hará con un comando con el siguiente formato:

```
ROLLBACK TO SAVEPOINT nombre_punto_restauración;
```

Artículo sobre los puntos de restauración.
http://es.wikipedia.org/wiki/Savepoint

Ejemplo sobre el uso de los puntos de restauración.
http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14261/sqloperations.htm#BABGAAIG

```
CREATE TABLE emp name AS SELECT employee id, last name, salary FROM employees;
CREATE UNIQUE INDEX empname_ix ON emp_name (employee_id);

DECLARE
   emp id        employees.employee id%TYPE;
   emp_lastname  employees.last_name%TYPE;
   emp_salary    employees.salary%TYPE;
BEGIN
   SELECT employee_id, last_name, salary INTO emp_id, emp_lastname,
     emp salary FROM employees WHERE employee id = 120;
   UPDATE emp name SET salary = salary * 1.1 WHERE employee id = emp id;
   DELETE FROM emp name WHERE employee id = 130;
   SAVEPOINT do_insert;
   INSERT INTO emp_name VALUES (emp_id, emp_lastname, emp_salary);
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
      ROLLBACK TO do insert;
      DBMS_OUTPUT.PUT_LINE('Insert has been rolled back');
END;
/
```

7.- Problemas asociados al acceso simultáneo a los datos.

Caso práctico

Ana tiene una duda que le quiere preguntar a Juan, ya que se ha estado planteando qué ocurre en el supuesto caso de que dos operaciones simultáneas modifiquen un mismo registro. Por ejemplo, si se ofrece la posibilidad de que se puedan transferir créditos de un usuario a otro, qué ocurriría si justo en un mismo momento dos usuarios le regalan crédito a un tercero. ¿Podría ocurrir que sólo llegara a realizarse una de las dos operaciones?

Para explicarle su idea le plantea el siguiente supuesto: El usuario A no disponía de crédito antes de realizar esas operaciones. El usuario B le va a dar 100 y el C dará 50. Cuando se inicia la operación de B, observa que el saldo de A en ese momento es 0. Cuando todavía no ha terminado la operación de B, se inicia simultáneamente la de C, que consulta el saldo de A que sigue siendo 0 todavía. Cuando B termina de transferir el crédito a A, el saldo se pone a 100 puesto que tenía 0 y le suma sus 100. Pero C estaba haciendo lo mismo, y al saldo 0 que tenía cuando hizo la consulta, le suma 50, por lo que al final sólo le quedará a A como saldo 50 en vez de 150.

Juan le responde que ese tipo de problemas de acceso simultáneo a los datos están controlados en las bases de datos con lo que se denomina bloqueos.

En una base de datos a la que accede un solo usuario, un dato puede ser modificado sin tener en cuenta que otros usuarios puedan modificar el mismo dato al mismo tiempo. Sin embargo, en una base de datos multiusuario, las sentencias contenidas en varias transacciones simultáneas pueden actualizar los datos simultáneamente. Las transacciones ejecutadas simultáneamente, deben generar resultados consistentes. Por tanto, una base de datos multiusuario debe asegurar:

- ✓ **Concurrencia de datos**: asegura que los usuarios pueden acceder a los datos al mismo tiempo.
- ✓ **Consistencia de datos**: asegura que cada usuario tiene una vista consistente de los datos, incluyendo los cambios visibles realizados por las transacciones del mismo usuario y las transacciones finalizadas de otros usuarios.

En una base de datos monousuario, no son necesarios los bloqueos ya que sólo modifica la información un solo usuario. Sin embargo, cuando varios usuarios acceden y modifican datos, la base de datos debe proveer un mecanismo para prevenir la modificación concurrente del mismo dato. Los bloqueos permiten obtener los siguientes requerimientos fundamentales en la base de datos:

- ✓ **Consistencia**: Los datos que están siendo consultados o modificados por un usuario no pueden ser cambiados por otros hasta que el usuario haya finalizado la operación completa.
- ✓ **Integridad**: Los datos y sus estructuras deben reflejar todos los cambios efectuados sobre ellos en el orden correcto.

La base de datos Oracle proporciona concurrencia de datos, consistencia e integridad en las transacciones mediante sus mecanismos de bloqueo. Los bloqueos se realizan de forma automática y no requiere la actuación del usuario.

Interesante enlace sobre control de concurrencia.

Integridad: Control de concurrencia.

Introducción

En sistema multiusuario es imprescindible, un mecanismo de control de concurrencia para conservar la integridad de la BD.

- ✓ Todos los datos deben ser iguales para todos los usuarios.

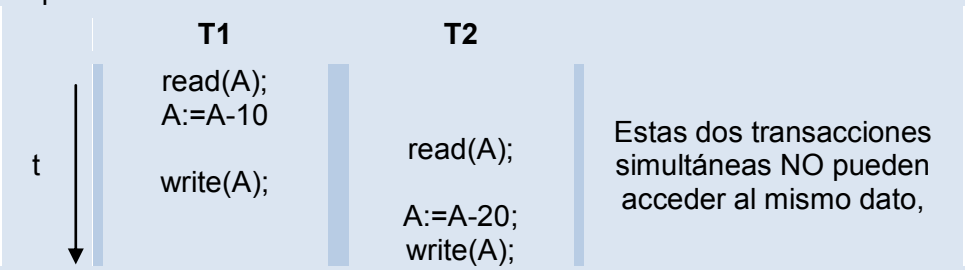
Cuando se ejecutan varias transacciones simultáneamente pueden producirse estados inconsistentes en la BD:

- ✓ Una transacción bancaria lee un importe y le resta 100 euros y antes de actualizar la BBDD otra transacción lee ese dato.

ORACLE es una BD multiusuario. Se necesita
- ✓ Concurrencia
- ✓ Consistencia

Maximización de concurrencia ➔ *maximiza productividad y desarrollo*

Ejemplo de "problema" de concurrencia:



OBJETIVO para controlar la concurrencia:
Garantizar que las transacciones sean seriables así, se garantizará la consistencia de las transacciones.

Problemas clásicos de concurrencia:
- ✓ Modificación perdida
- ✓ Modificación temporal
- ✓ Totalización incorrecta
- ✓ Lectura no repetible

En **ORACLE**los fenómenos prevenibles son:
- ✓ Lectura sucia
- ✓ Lectura no repetible (borrosa)
- ✓ Lectura fantasma

Para prevenir problemas ⇒ *Técnicas de Control de Concurrencia*
- ✓ ***Técnicas Pesimistas (Prevención)***
  - ➔ *Bloqueos, Marcas de Tiempo,…*
- ✓ ***Técnicas Optimistas (Corrección)***
  - ➔ *Técnicas de Validación*

ORACLE tiene como soluciones:
- ✓ Varios tipos de **bloqueo**
- ✓ Un **modelo** de consistencia **multiversión**.
- ✓ Niveles de **AISLAMIENTO**:
  - ➔ Aceptación de lectura (read commited)
  - ➔ Serializable
  - ➔ Modo de solo lectura (read-only mode)

ORACLE para prevenir interacción destructiva de datos entre usuarios:
- ✓ Consistencia: Asegura que los datos que estamos viendo no cambian (por otros usuarios) hasta que acabemos la transacción
- ✓ Integridad: Asegura que los datos y estructuras reflejan los cambios en una secuencia correcta.

Técnicas de Bloqueo.

Bloqueo. Variable cerrojo
Un bloqueo, asocia una variable (cerrojo) a cada elemento de datos que describe el estado de dicho elemento, respecto a las posibles operaciones que sobre él se puede realizar.
- ✓ Identificador del Elemento bloqueado
- ✓ Identificador de la Transacción que lo bloquea

Una transacción obtiene un bloqueo solicitándolo al Gestor de Bloqueos.

Un bloqueo es una garantía de ciertos derechos de exclusividad para la Transacción.

Tipos:

- ✓ Bloqueos exclusivos: Un único bloqueo por recurso
- ✓ Bloqueos Compartidos: Muchos bloqueos por recurso

ORACLE tiene los dos tipos de bloqueo así como control de consistencia multiversión para asegurar acceso concurrente a los datos.

Un **Protocolo de Bloqueo** indica cuando una transacción puede bloquear y desbloquear elementos. Si los bloqueos se realizaran de manera arbitraria se podrían producir inconsistencias.

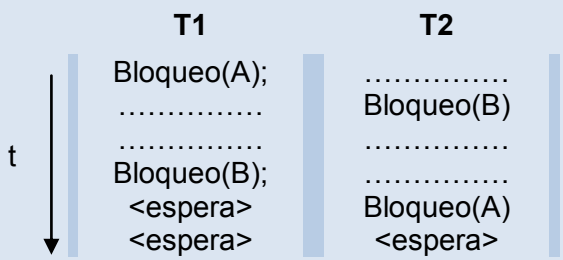
Protocolo de Bloqueo en dos Fases (Two-Phase-Locking)

- ✓ Fase de Crecimiento: Bloquea pero no Libera
- ✓ Fase de Encogimiento: Libera pero no Bloquea

Asegura la seriabilidad

Problema de Interbloqueo. DEADLOCK

Dos o más transacciones se quedan a la espera de que se liberen elementos que tiene bloqueados otra.



Puede producirse incluso con protocolos de bloqueo que garantizan la seriabilidad
Soluciones:

- ✓ **Prevención del interbloqueo**, obligando a que las transacciones bloqueen todos los elementos que necesitan por adelantado
- ✓ **Detectar el interbloqueo**, controlando de forma periódica si se ha producido, para lo que suele construir un grafo de espera:
  - ➔ Cada nodo representa una transacción
  - ➔ Existe un arco entre Ti y Tj si Ti está esperando un recurso que tiene bloqueado Tj
  - ➔ Existe interbloqueo si el Grafo tiene un ciclo.

Cada SGBD tiene su propia política para escoger las víctimas, aunque suelen ser las transacciones más recientes.

Prevención el interbloqueo:

- ✓ Bloquear todo lo necesario por adelantado
- ✓ Comprobar por adelantado la posibilidad de Interbloqueo, la transacción esperará en el caso en que de atenderse su petición se produciría Interbloqueo.
- ✓ Nuevos protocolos de bloqueo que además de garantizar la seriabilidad, eviten el interbloqueo. (ej. Protocolo del árbol).
- ✓ Utilización de Marcas de Tiempo para cada transacción MT(Ti):
  - ➔ Usar expropiación y retroceso de Transacciones
    - → esperar-morir (Wait – Die)

→ herir-esperar (Wound – Wait)

Detección del Interbloqueo y Recuperación
Periódicamente se comprueba la existencia de Interbloqueo:
- ✓ Si se detecta Interbloqueo, se elige una transacción como víctima y se la mata.
- ✓ Se puede producir inanición si se elige siempre a la misma víctima.

¿Cuándo usar prevención y cuando detección?
GRANULARIDAD del Bloqueo: para mejorar rendimiento
Son posibles diversos niveles de bloqueo
- ✓ un campo de un registro
- ✓ un registro
- ✓ un fichero
- ✓ la base de datos

Granularidad gruesa ➔ menor gestión de bloqueos y menor nivel de concurrencia.
Granularidad fina ➔ mayor gestión de bloqueos y mayor nivel de concurrencia.

ORACLE bloquea automáticamente:
- ✓ **Objetos de usuario** (tablas, filas, etc. (estructuras y datos))
- ✓ **Objetos del sistema** àinvisibles a los usuarios (estructuras de datos compartidas en memoria, filas de diccionario de datos, etc.)

Marcas de Tiempo
En lugar de determinar el orden entre las transacciones en conflicto en función del momento del acceso a los elementos, determinan por adelantado una ordenación de las transacciones.

El interbloqueo es imposible.

Una marca de tiempo es un identificador único asociado a cada transacción

Las actualizaciones físicas se retrasan hasta la confirmación de las transacciones. No se puede actualizar ningún elemento actualizado por otra transacción más reciente.

Protocolos:
- ✓ **Wait-die** que fuerza a una transacción a esperar en caso de que entre en conflicto con otra transacción cuya marca de tiempo sea más reciente, o a morir (abortar y reiniciar) si la transacción que se está ejecutando es más antigua.
- ✓ **Wound-wait**, que permite a una transacción matar a otra que posea una marca de tiempo más reciente, o que fuerza a la transacción peticionaria a esperar.

Marcas de tiempo multiversión
Varias transacciones leen y escriben diferentes versiones del mismo dato siempre que cada transacción sea un conjunto consistente de versiones de todos los datos a los que accede.

ORACLE utiliza un sistema multiversión

Protocolo:
- ✓ Está basado en marcas de tiempo
- ✓ El control de concurrencia es de varias versiones a la vez de un item de datos.

- ✓ Cuando una transacción requiere acceder a un item, la marca de tiempo de la transacción es comparada con las marcas de tiempo de las diferentes versiones del item.
- ✓ Se elige una versión de los items para mantener la serializabilidad del plan de items que se este ejecutando.

Modelo Multiversión de ORACLE:

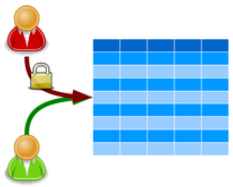
Consistencia en lectura:

- ✓ Imaginar cada usuario operando con una copia privada de la BD (modelo consistente multiversión)
- ✓ Características
  - ➔ garantiza que los datos no cambian durante la ejecución
  - ➔ Los lectores no esperan a los que escriben o a otros lectores
  - ➔ Los que escriben no esperan a los lectores, pero sí a otros escritores de los mismos datos
- ✓ Niveles de concurrencia
  - ➔ Sentencia
  - ➔ Transacción

Cuando se lee y modifica al mismo tiempo, Oracle crea:

- ✓ Conjunto de datos (vistas) consistente en lectura
- ✓ Cuando se modifica (antes del COMMIT)
  - ➔ se almacenan los valores antiguos de los datos en *Segmentos de Rollback*
  - ➔ Crea la Vista Consistente a partir de:
    - → § Información actual en el *Área Global del Sistema*
    - → § Información antigua en los *Segmentos de Rollback*
- ✓ Al hacer el COMMIT:
  - ➔ Se hacen los cambios permanentes para todas las vistas posteriores

http://ocw.uc3m.es/ingenieria-informatica/diseno-y-administracion-de-bases-de-datos/teoria/Tema4_6%28Administracion_Concurrencia%29.pdf



7.1.- Políticas de bloqueo.

La base de datos permite el uso de diferentes tipos de bloqueos, dependiendo de la operación que realiza el bloqueo.

Los bloqueos afectan a la interacción de lectores y escritores. Un lector es una consulta sobre un recurso, mientras que un escritor es una sentencia que realiza un modificación sobre un recurso. Las siguientes reglas resumen el comportamiento de la base de datos Oracle sobre lectores y escritores:

- ✓ Un registro es bloqueado sólo cuando es modificado por un escritor: Cuando una sentencia actualiza un registro, la transacción obtiene un bloqueo sólo para ese registro.
- ✓ Un escritor de un registro bloquea a otro escritor concurrente del mismo registro: Si una transacción está modificando una fila, un bloqueo del registro impide que otra transacción modifique el mismo registro simultáneamente.
- ✓ Un lector nunca bloquea a un escritor: Puesto que un lector de un registro no lo bloquea, un escritor puede modificar dicho registro. La única excepción es la sentencia `SELECT ... FOR UPDATE`, que es un tipo especial de sentencia `SELECT` que bloquea el registro que está siendo consultado.
- ✓ Un escritor nunca bloquea a un lector: Cuando un registro está siendo modificado, la base de datos proporciona al lector una vista del registro sin los cambios que se están realizando.

Hay dos mecanismos para el bloqueo de los datos en una base de datos: el bloqueo **pesimista** y bloqueo **optimista**. En el bloqueo pesimista de un registro o una tabla se realiza el bloqueo inmediatamente, en cuanto el bloqueo se solicita, mientras que en un bloqueo optimista el acceso al

registro o la tabla sólo está cerrado en el momento en que los cambios realizados a ese registro se actualizan en el disco. Esta última situación sólo es apropiada cuando hay menos posibilidad de que alguien necesite acceder al registro mientras está bloqueado, de lo contrario no podemos estar seguros de que la actualización tenga éxito, porque el intento de actualizar el registro producirá un error si otro usuario actualiza antes el registro. Con el bloqueo pesimista se garantiza que el registro será actualizado.

Supongamos que un usuario está en proceso de modificación de un registro, y otro en ese mismo momento quiere leer ese mismo registro. ¿Qué tipo de bloqueo debes establecer para que el segundo usuario obtenga los datos con los cambios que está efectuando el primero?

◉ **Bloqueo pesimista** ○ Bloqueo optimista

Este tipo de bloqueo impide que un usuario lea los datos del registro hasta que se finalice la modificación que ha empezado otro

Documento, en inglés, sobre los bloqueos optimistas y pesimistas en Oracle con algunos ejemplos.
http://www.orafaq.com/papers/locking.pdf

7.2.- Bloqueos compartidos y exclusivos.

En general, la base de datos usa dos tipos de bloqueos: bloqueos exclusivos y bloqueos compartidos. Un recurso, por ejemplo un registro de una tabla, sólo puede obtener un bloqueo exclusivo, pero puede conseguir varios bloqueos compartidos.

- ✓ **bloqueo exclusivo**: Este modo previene que sea compartido el recurso asociado. Una transacción obtiene un bloqueo exclusivo cuando modifica los datos. La primera transacción que bloquea un recurso exclusivamente, es la única transacción que puede modificar el recurso hasta que el bloqueo exclusivo es liberado.
- ✓ **bloqueo compartido**: Este modo permite que sea compartido el recurso asociado, dependiendo de la operación en la que se encuentra involucrado. Varios usuarios que estén leyendo datos pueden compartir los datos, realizando bloqueos compartidos para prevenir el acceso concurrente de un escritor que necesita un bloqueo exclusivo. Varias transacciones pueden obtener bloqueos compartidos del mismo recurso.

Por ejemplo, supongamos que transacción usa la sentencia `SELECT ... FOR UPDATE` para consultar un registro de una tabla. La transacción obtiene un bloqueo exclusivo del registro y un bloqueo compartido de la tabla. El bloqueo del registro permite a otras sesiones que modifiquen cualquier otro registro que no sea el registro bloqueado, mientras que el bloqueo de la tabla previene que otras sesiones modifiquen la estructura de la tabla. De esta manera, la base de datos permite la ejecución de todas las sentencias que sean posibles.

Definición y ejemplos del bloqueo exclusivo y compartido
http://dircompucv.ciens.ucv.ve/generador/sites/administracion-de-bd/archivos/Integridad.pdf

7.3.- Bloqueos automáticos.

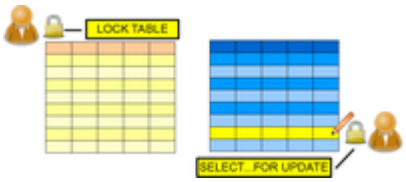
La base de datos Oracle bloquea automáticamente un recurso usado por una transacción para prevenir que otras transacciones realicen alguna acción que requiera acceso exclusivo sobre el mismo recurso. La base de datos adquiere automáticamente diferentes tipo de bloqueos con diferentes niveles de restricción dependiendo del recurso y de la operación que se realice.

Los bloqueos que realiza la base de datos Oracle están divididos en las siguientes categorías:

- ✓ **Bloqueos DML**: Protegen los datos, garantizando la integridad de los datos accedidos de forma concurrente por varios usuarios. Por ejemplo, evitan que dos clientes compren el último artículo disponible en una tienda online. Estos bloqueos pueden ser sobre un sólo registro o sobre la tabla completa.
- ✓ **Bloqueos DDL**: Protegen la definición del esquema de un objeto mientras una operación DDL actúa sobre él. Los bloqueos se realizan de manera automática por cualquier transacción DDL que lo requiera. Los usuarios no pueden solicitar explícitamente un bloqueo DDL.
- ✓ **Bloqueos del sistema**: La base de datos Oracle usa varios tipos de bloqueos del sistema para proteger la base de datos interna y las estructuras de memoria.

Definición de bloqueo automático, exclusivo y compartido.
http://books.google.com/books?id=zYBWm5-X5usC&lpg=PA166&ots=ha7MVfOo9n&pg=PA166#v=onepage&q&f=true

7.4.- Bloqueos manuales.



Hemos visto que la base de datos Oracle realiza bloqueos de forma automática para asegurar la concurrencia de datos, su integridad y consistencia en la consulta de datos. Sin embargo, también puedes omitir los mecanismos de bloqueo que realiza por defecto la base de datos Oracle. Esto puede ser útil en situaciones como las siguientes:

- ✓ En aplicaciones que requieren consistencia en la consulta de datos a nivel de transacciones o en lecturas repetitivas: En este caso, las consultas obtienen datos consistentes en la duración de la transacción, sin reflejar los cambios realizados por otras transacciones.
- ✓ En aplicaciones que requieren que una transacción tenga acceso exclusivo a un recurso con el fin de que no tenga que esperar que otras transacciones finalicen.

La cancelación de los bloqueos automáticos se puede realizar a nivel de sesión o de transacción. En el nivel de sesión, una sesión puede establecer el nivel requerido de aislamiento de la transacción con la sentencia `ALTER SESSION`. En el nivel de transacción, las transacciones que incluyan las siguientes sentencias SQL omiten el bloqueo por defecto:

- ✓ `SET TRANSACTION ISOLATION LEVEL`
- ✓ `LOCK TABLE`
- ✓ `SELECT...FOR UPDATE`

Los bloqueos que establecen las sentencias anteriores terminan una vez que la transacción ha finalizado.

Definición y ejemplos sobre transacciones y control de concurrencia con bloqueos manuales.
http://www.fdi.ucm.es/profesor/hector/DB-II/transacciones.pdf

TEMA 6

INDICE

1.- Introducción. 5
2.- Conceptos básicos. 6
2.1.- Unidades léxicas (I). 6
Delimitadores. 7
Identificadores. 7
Literales. 8
Comentarios 8
Ejercicio resuelto 8
2.2.- Tipos de datos simples, variables y constantes. 8
Numéricos. 8
Alfanuméricos. 9
Grandes objetos 9
Otros 9
2.2.1.- Subtipos. 9
2.2.2.- Variables y constantes. 10
Conversión de tipos. 11
Precedencia de operadores. 11
2.3.- El bloque PL/SQL. 11
2.4.- Estructuras de contro. 12
Control condicional. 12
Control iterativo. 13
2.5.- Manejo de errores. 14
Ejercicio resuelto 16
3.- Tipos de datos compuestos. 19
3.1.- Registros. 19
3.2.- Colecciones. Arrays de longitud variable. 20
Arrays de longitud variable. 20
3.2.1.- Colecciones. Tablas anidadas. 22
3.3.- Cursores. 23
Cursores implícitos. 24
Atributos de un cursor 24
3.3.1.- Cursores explícitos. 25
3.3.2.- Cursores variables. 26
4.- Abstracción en PL/SQL. 28
4.1.- Subprogramas. 28
4.1.1.- Almacenar subprogramas en la base de datos. 29
4.1.2.- Parámetros de los subprogramas 30
4.1.3.- Sobrecarga de subprogramas y recursividad. 32
4.2.- Paquetes. 35
4.2.1.- Ejemplos de utilización del paquete DBMS_OUTPUT. 39
Ejercicio resuelto 40
4.3.- Objetos 40
Ejercicio resuelto 42
4.3.1.- Objetos. Funciones mapa y funciones de orden. 42
5.- Disparadores. 44
5.1.- Definición de disparadores. 44
5.2.- Ejemplos de disparadores. 45
Ejercicio resuelto 46
Ejercicio resuelto 46
Ejercicio resuelto 47
6.- Interfaces de programación de aplicaciones para lenguajes externos. 48

Programación de bases de datos.

Caso práctico

__Juan__ recuerda, de cuando estudió el Ciclo de Desarrollo de Aplicaciones Informáticos, que había muchas tareas que se podían automatizar dentro de la base de datos mediante el uso de un lenguaje de programación, e incluso que se podían programar algunas restricciones a la hora de manipular los datos. __Juan__ se lo comenta a __María__ y ésta se muestra ilusionada con dicha idea ya que muchas veces repiten el trabajo con la base de datos de juegos on-line que tienen entre manos (consultas, inserciones, etc. que son muy parecidas y que se podrían automatizar).

Para ello, hablan con __Ada__ y ésta les comenta que claro que se puede hacer y que precisamente eso es lo que les toca hacer ahora. __Ada__ les dice que para ese propósito existe un lenguaje de programación llamado PL/SQL que permite hacer lo que ellos quieren, así que les pasa un manual para que se lo vayan leyendo y se vayan poniendo manos a la obra con la base de datos de juegos on-line.

Ahora que ya dominas el uso de SQL para la manipulación y consulta de datos, es el momento de dar una vuelta de tuerca adicional para mejorar las aplicaciones que utilicen nuestra base de datos. Para ello nos vamos a centrar en la programación de bases de datos, utilizando el lenguaje PL/SQL. En esta unidad conoceremos qué es PL/SQL, cuál es su sintaxis y veremos cómo podemos sacarle el máximo partido a nuestra base de datos mediante su uso.

La mayor parte de los ejemplos de esta unidad están basados en el modelo de datos extraído del siguiente caso de estudio:

Una empresa de telefonía tiene sus centros de llamadas distribuidos por la geografía española en diferentes oficinas. Estas oficinas están jerarquizadas en familias de agentes telefónicos. Cada familia, por tanto, podrá contener agentes u otras familias. Los agentes telefónicos, según su categoría, además se encargarán de supervisar el trabajo de todos los agentes de una oficina o de coordinar el trabajo de los agentes de una familia dada. El único agente que pertenecerá directamente a una oficina y que no formará parte de ninguna familia será el supervisor de dicha oficina, cuya categoría es la 2. Los coordinadores de las familias deben pertenecer a dicha familia y su categoría será 1 (no todas las familias tienen por qué tener un coordinador y dependerá del tamaño de la oficina, ya que de ese trabajo también se puede encargar el supervisor de la oficina). Los demás agentes deberán pertenecer a una familia, su categoría será 0 y serán los que principalmente se ocupen de atender las llamadas.

- ✓ De los agentes queremos conocer su nombre, su clave y contraseña para entrar al sistema, su categoría y su habilidad que será un número entre 0 y 9 indicando su habilidad para atender llamadas.
- ✓ Para las familias sólo nos interesa conocer su nombre.
- ✓ Finalmente, para las oficinas queremos saber su nombre, domicilio, localidad y código postal de la misma.

Un posible modelo entidad-relación para el problema expuesto podría ser el siguiente:

De este modelo de datos surgen tres tablas, que podrían ser creadas en Oracle con las siguientes sentencias:

```
create table oficinas (
     identificador     number(6) not null primary key,
     nombre          varchar2(40) not null unique,
     domicilio     varchar2(40),
     localidad     varchar2(20),
     codigo_postal     number(5)
);

create table familias (
     identificador     number(6) not null primary key,
     nombre          varchar2(40) not null unique,
     familia          number(6) references familias,
     oficina          number(6) references oficinas
);

create table agentes (
     identificador     number(6) not null primary key,
     nombre          varchar2(60) not null,
     usuario          varchar2(20) not null unique,
     clave          varchar2(20) not null,
     habilidad     number(1) not null,
     categoria     number(1) not null,
     familia          number(6) references familias,
     oficina          number(6) references oficinas
);
```

Vamos a insertar algunas filas para que las pruebas de nuestros ejemplos tengan algo de sentido. Para ello podemos utilizar las siguientes sentencias:

```
insert into oficinas values (1, 'Madrid', 'Gran vía, 37', 'Madrid', 28000);
insert into oficinas values (2, 'Granada', 'Camino Ronda, 50', 'Granada', 36000);
insert into oficinas values (3, 'Jaén', 'Gran Eje, 80', 'Jaén', 27000);

insert into familias values (11, 'Madrid-1', NULL, 1);
insert into familias values (111, 'Madrid-1.1', 11, NULL);
insert into familias values (112, 'Madrid-1.2', 11, NULL);
insert into familias values (1121, 'Madrid-1.2.1', 112, NULL);
insert into familias values (1122, 'Madrid-1.2.2', 112, NULL);
insert into familias values (1123, 'Madrid-1.2.3', 112, NULL);
insert into familias values (21, 'Granada-1', NULL, 2);
insert into familias values (211, 'Granada-1.1', 21, NULL);
insert into familias values (212, 'Granada-1.2', 21, NULL);
insert into familias values (213, 'Granada-1.3', 21, NULL);
insert into familias values (31, 'Jaén-1', NULL, 3);

insert into agentes values (31, 'José Ramón Jiménez Reyes', 'jrjr', 'sup31', 9, 2, NULL, 3);
insert into agentes values (311, 'Pedro Fernández Arias', 'pfa', 'ag311', 5, 0, 31, NULL);
insert into agentes values (312, 'Vanesa Sánchez Rojo', 'vsr', 'ag312', 5, 0, 31, NULL);
insert into agentes values (313, 'Francisco Javier García Escobedo', 'fjge', 'ag313', 5, 0,
31, NULL);
insert into agentes values (314, 'Pilar Ramirez Pérez', 'prp', 'ag314', 5, 0, 31, NULL);
insert into agentes values (315, 'José Luis García Martínez', 'jlgm', 'ag315', 5, 0, 31,
NULL);
insert into agentes values (21, 'Sebastián López Ojeda', 'slo', 'sup21', 9, 2, NULL, 2);
insert into agentes values (211, 'Diosdado Sánchez Hernández', 'dsh', 'ag211', 8, 1, 21,
NULL);
insert into agentes values (2111, 'José Juan Cano Pardo', 'jjcp', 'ag2111', 5, 0, 211, NULL);
insert into agentes values (2112, 'Flor Moncada Añón', 'ag2112', 'fma', 5, 0, 211, NULL);
insert into agentes values (2113, 'Juan Manuel Alcazar Donaire', 'jmad', 'ag2113', 5, 0, 211,
NULL);
insert into agentes values (2121, 'Manuel Jesús Rubia Mateos', 'mjrm', 'ag2121', 5, 0, 212,
NULL);
insert into agentes values (2122, 'Esther López Delgado', 'eld', 'ag2122', 5, 0, 212, NULL);
insert into agentes values (2123, 'Francisco Javier Cabrerizo Membrilla', 'fjcm', 'ag2123', 5,
0, 212, NULL);
insert into agentes values (2124, 'Verónica Cabrerizo Menbrilla', 'vcm', 'ag2124', 5, 0, 212,
NULL);
insert into agentes values (2125, 'María José Navascués Morales', 'mjnm', 'ag2125', 5, 0, 212,
NULL);
insert into agentes values (2131, 'Isabel Cruz Granados', 'icg', 'ag2131', 5, 0, 213, NULL);
insert into agentes values (2132, 'Antonio Casado Fernández', 'acf', 'ag2132', 5, 0, 213,
NULL);
insert into agentes values (2133, 'Gabriel Callejón García', 'gcg', 'ag2133', 5, 0, 213,
NULL);
insert into agentes values (2134, 'Enrique Cano Balsera', 'ecb', 'ag2134', 5, 0, 213, NULL);
insert into agentes values (11, 'Narciso Jáimez Toro', 'njt', 'sup11', 9, 2, NULL, 1);
```

```
insert into agentes values (111, 'Jesús Baños Sancho', 'jbs', 'ag111', 8, 1, 11, NULL);
insert into agentes values (1111, 'Salvador Romero Villegas', 'srv', 'ag1111', 7, 1, 111,
NULL);
insert into agentes values (1112, 'José Javier Bermúdez Hernández', 'jjbh', 'ag1112', 7, 1,
111, NULL);
insert into agentes values (1113, 'Alfonso Bonillo Sierra', 'abs', 'ag1113', 7, 1, 111, NULL);
insert into agentes values (1121, 'Silvia Thomas Barrós', 'stb', 'ag1121', 7, 1, 112, NULL);
insert into agentes values (11211, 'Ernesto Osoro Gorrotxategi', 'eog', 'ag11211', 5, 0, 1121,
NULL);
insert into agentes values (11212, 'Guillermo Campos Guillén', 'gcg', 'ag11212', 5, 0, 1121,
NULL);
insert into agentes values (11213, 'Antonio Fernández Ruíz', 'afr', 'ag11213', 5, 0, 1121,
NULL);
insert into agentes values (11214, 'María Luisa López Caballero', 'mllc', 'ag11214', 5, 0,
1121, NULL);
insert into agentes values (11221, 'Virginia Morenas Rubio', 'vmr', 'ag11221', 5, 0, 1121,
NULL);
insert into agentes values (11222, 'Rosario Castro García', 'rcg', 'ag11222', 5, 0, 1122,
NULL);
insert into agentes values (11223, 'Antonio Álvarez Palomeque', 'aap', 'ag11223', 5, 0, 1122,
NULL);
insert into agentes values (11224, 'David Martínez Martínez', 'dmm', 'ag11224', 5, 0, 1122,
NULL);
insert into agentes values (11225, 'Juan Corral González', 'jcg', 'ag11225', 5, 0, 1122,
NULL);
insert into agentes values (11226, 'Eduardo Alfada Pereira', 'eap', 'ag11226', 5, 0, 1122,
NULL);
insert into agentes values (11231, 'Cayetano García Herrera', 'cgh', 'ag11231', 5, 0, 1123,
NULL);
insert into agentes values (11232, 'José Antonio Sieres Vega', 'jasv', 'ag11232', 5, 0, 1123,
NULL);
insert into agentes values (11233, 'Juan Manuel Guzmán García', 'jmgg', 'ag11233', 5, 0, 1123,
NULL);

commit;
```

1.- Introducción.

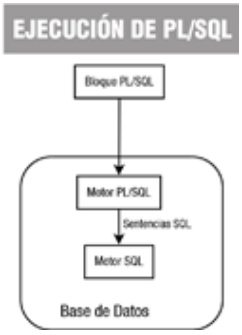
Caso práctico

__Juan__ y __María__ se han puesto a repasar el manual de PL/SQL que les ha pasado Ada. Aunque no han avanzado mucho con el mismo, ya saben a qué se van a enfrentar y los beneficios que pueden obtener del uso del mismo para su aplicación de juegos on-line. Cuando hacen la primera parada de la mañana para tomarse un café, ambos se ponen a comentar las primeras conclusiones que han sacado después de su primer acercamiento a este lenguaje. Ambos están deseosos de seguir avanzando en su aprendizaje y saben que para ello cuentan con la inestimable ayuda de __Ada__.

Estarás pensado que si no tenemos bastante con aprender SQL, sino que ahora tenemos que aprender otro lenguaje más que lo único que va a hacer es complicarnos la vida. Verás que eso no es cierto ya que lo más importante, que es el conocimiento de SQL, ya lo tienes. PL/SQL tiene una sintaxis muy sencilla y verás como pronto te acostumbras y luego no podrás vivir sin él.

Pero, **¿qué es realmente PL/SQL?**

PL/SQL es un lenguaje procedimental estructurado en bloques que amplía la funcionalidad de SQL. Con PL/SQL podemos usar sentencias SQL para manipular datos y sentencias de control de flujo para procesar los datos. Por tanto, PL/SQL combina la potencia de SQL para la manipulación de datos, con la potencia de los lenguajes procedimentales para procesar los datos.



Aunque PL/SQL fue creado por Oracle, hoy día todos los gestores de bases de datos utilizan un lenguaje procedimental muy parecido al ideado por Oracle para poder programar las bases de datos.

Como veremos, en PL/SQL podemos definir variables, constantes, funciones, procedimientos, capturar errores en tiempo de ejecución, anidar cualquier número de bloques, etc. como solemos hacer en cualquier otro lenguaje de programación. Además, por medio de PL/SQL programaremos los disparadores de nuestra base de datos, tarea que no podríamos hacer sólo con SQL.

El motor de PL/SQL acepta como entrada bloques PL/SQL o subprogramas, ejecuta sentencias procedimentales y envía sentencias SQL al servidor de bases de datos. En el esquema adjunto puedes ver su funcionamiento.

Una de las grandes ventajas que nos ofrece PL/SQL es un mejor rendimiento en entornos de red cliente-servidor, ya que permite mandar bloques PL/SQL desde el cliente al servidor a través de la red, reduciendo de esta forma el tráfico y así no tener que mandar una a una las sentencias SQL correspondientes.

En el siguiente enlace podrás encontrar una breve historia de PL/SQL.
http://plsqlenespanol.wordpress.com/2007/06/16/un-poco-de-historia/

En estos enlaces podrás comprobar como los gestores de bases de datos incluyen hoy día un lenguaje procedimental para programar la base de datos muy parecido a PL/SQL.
http://es.wikipedia.org/wiki/Procedimiento_almacenado
http://es.wikipedia.org/wiki/PL/PgSQL

2.- Conceptos básicos.

Caso práctico

__Juan__ ha avanzado muy rápido en la lectura del manual que le pasó __Ada__. __Juan__ ya sabe programar en otros lenguajes de programación y por tanto la lectura de los primeros capítulos que se centran en cómo se estructura el lenguaje, los tipos de datos, las estructuras de control, etc. le han resultado muy fáciles de comprender. Sabe que lo único que tendrá que tener en cuenta son algunos aspectos en cuanto a las reglas de escritura y demás, pero la lógica es la de cualquier otro lenguaje de programación.
Como __María__ está un poco más verde en el tema de la programación, __Juan__ se ha ofrecido a darle un repaso rápido a todo lo que él ha visto y a explicarle todo aquello en lo que tenga dudas y a ver si pronto se pueden poner manos a la obra con la base de datos de juegos on-line.

En este apartado nos vamos a ir introduciendo poco a poco en los diferentes conceptos que debemos tener claros para programar en PL/SQL. Como para cualquier otro lenguaje de programación, debemos conocer las reglas de sintaxis que podemos utilizar, los diferentes elementos de que consta, los tipos de datos de los que disponemos, las estructuras de control que nos ofrece (tanto iterativas como condicionales) y cómo se realiza el manejo de los errores.

Como podrás comprobar, es todo muy sencillo y pronto estaremos escribiendo fragmentos de código que realizan alguna tarea particular. ¡Vamos a ello!

Indica cuáles de las siguientes características que nos proporciona PL/SQL son ciertas.

- [x] **Permite reducir el tráfico en la red en entornos cliente-servidor**
- [ ] No podemos utilizar sentencias SQL dentro de un bloque PL/SQL
- [x] **Nos ofrece las ventajas de SQL y la potencia de un lenguaje procedimental**
- [ ] Para utilizar PL/SQL debemos instalar diferentes drivers en nuestra base de datos Oracle

2.1.- Unidades léxicas (I).

En este apartado nos vamos a centrar en conocer cuáles son las unidades léxicas que podemos utilizar para escribir código en PL/SQL. Al igual que en nuestra lengua podemos distinguir diferentes unidades léxicas como palabras, signos de puntuación, etc. En los lenguajes de programación también existen diferentes unidades léxicas que definen los elementos más pequeños que tienen sentido propio y que al combinarlos de manera adecuada, siguiendo las reglas de sintaxis, dan lugar a sentencias válidas sintácticamente.

PL/SQL es un lenguaje no sensible a las mayúsculas, por lo que será equivalente escribir en mayúsculas o minúsculas, excepto cuando hablemos de literales de tipo cadena o de tipo carácter.
Cada unidad léxica puede estar separada por espacios (debe estar separada por espacios si se trata de 2 identificadores), por saltos de línea o por tabuladores para aumentar la legibilidad del código escrito.

```
IF A=CLAVE THEN ENCONTRADO:=TRUE;ELSE ENCONTRADO:=FALSE;END IF;
```

Sería equivalente a escribir la siguiente línea:

```
if a=clave then encontrado:=true;else encontrado:=false;end if;
```

Y también sería equivalente a este otro fragmento:

```
IF a = clave THEN
     encontrado := TRUE;
ELSE
     encontrado := FALSE;
END IF;
```

Las unidades léxicas se pueden clasificar en:

- ✓ Delimitadores.
- ✓ Identificadores.
- ✓ Literales.
- ✓ Comentarios.

Vamos a verlas más detenidamente.

Delimitadores.

PL/SQL tiene un conjunto de símbolos denominados delimitadores utilizados para representar operaciones entre tipos de datos, delimitar comentarios, etc. En la siguiente tabla puedes ver un resumen de los mismos.

Delimitadores en PL/SQL.			
Delimitadores Simples.		Delimitadores Compuestos.	
Símbolo.	**Significado.**	**Símbolo.**	**Significado.**
+	Suma.	**	Exponenciación.
%	Indicador de atributo.	<>	Distinto.
.	Selector.	¡=	Distinto.
/	División.	<=	Menor o igual.
(	Delimitador de lista.	>=	Mayor o igual.
)	Delimitador de lista.	..	Rango.
:	Variable host.	||	Concatenación.
,	Separador de elementos.	<<	Delimitador de etiquetas.
*	Producto.	>>	Delimitador de etiquetas.
"	Delimitador de identificador acotado.	--	Comentario de una línea.
=	Igual relacional.	/*	Comentario de varias líneas.
<	Menor.	*/	Comentario de varias líneas.
>	Mayor.	:=	Asignación.
@	Indicador de acceso remoto.	=>	Selector de nombre de parámetro.
;	Terminador de sentencias.		
-	Resta/negación.		

Ya hemos visto qué son los delimitadores. Ahora vamos a continuar viendo el resto de unidades léxicas que nos podemos encontrar en PL/SQL.

Identificadores.

Los identificadores en PL/SQL, como en cualquier otro lenguaje de programación, son utilizados para nombrar elementos de nuestros programas. A la hora de utilizar los identificadores debemos tener en cuenta los siguientes aspectos:

- ✓ Un identificador es una **letra seguida** opcionalmente de **letras**, **números**, **$**, **_**, **#**.
- ✓ No podemos utilizar como identificador una palabra reservada.
  - ➔ Ejemplos válidos: X, A1, codigo_postal.
  - ➔ Ejemplos no válidos: rock&roll, on/off.
- ✓ PL/SQL nos permite además definir los identificadores acotados, en los que podemos usar cualquier carácter con una longitud máxima de 30 y deben estar delimitados por ". Ejemplo: "X*Y".
- ✓ En PL/SQL existen algunos identificadores predefinidos y que tienen un significado especial ya que nos permitirán darle sentido sintáctico a nuestros programas. Estos identificadores son las

palabras reservadas y no las podemos utilizar como identificadores en nuestros programas. Ejemplo: `IF`, `THEN`, `ELSE` ...

- ✓ Algunas palabras reservadas para PL/SQL no lo son para SQL, por lo que podríamos tener una tabla con una columna llamada 'type' por ejemplo, que nos daría un error de compilación al referirnos a ella en PL/SQL. La solución sería acotarlos. `SELECT "TYPE"` …

Literales.

Los literales se utilizan en las comparaciones de valores o para asignar valores concretos a los identificadores que actúan como variables o constantes. Para expresar estos literales tendremos en cuenta que:

- ✓ Los literales numéricos se expresarán por medio de notación decimal o de notación exponencial. Ejemplos: 234, +341, 2e3, -2E-3, 7.45, 8.1e3.
- ✓ Los literales de tipo carácter y de tipo cadena se deben delimitar con unas comillas simples.
- ✓ Los literales lógicos son `TRUE` y `FALSE`.
- ✓ El literal `NULL` que expresa que una variable no tiene ningún valor asignado.

Comentarios.

En los lenguajes de programación es muy conveniente utilizar comentarios en mitad del código. Los comentarios no tienen ningún efecto sobre el código pero sí ayudan mucho al programador o la programadora a recordar qué se está intentando hacer en cada caso (más aún cuando el código es compartido entre varias personas que se dedican a mejorarlo o corregirlo).

En PL/SQL podemos utilizar dos tipos de comentarios:

- ✓ Los comentarios de una línea se expresaran por medio del delimitador **--**. Ejemplo:
  `a:=b;  --asignación`
- ✓ Los comentarios de varias líneas se acotarán por medio de los delimitadores **/*** y ***/**. Ejemplo:

```
/* Primera línea de comentarios.
Segunda línea de comentarios. */
```

Ejercicio resuelto

Dada la siguiente línea de código, haz su descomposición en las diferentes unidades léxicas que contenga.

```
IF A <> B THEN iguales := FALSE; --No son iguales
```

Respuesta:

La descomposición en unidades léxicas sería la siguiente:

- ✓ Identificadores: **A**, **B**, **iguales**.
- ✓ Identificadores (*palabras reservadas*): **IF**, **THEN**.
- ✓ Delimitadores: **<>**, **:=**, **;**.
- ✓ Comentarios: **--No son iguales**.

2.2.- Tipos de datos simples, variables y constantes.

En cualquier lenguaje de programación, las variables y las constantes tienen un tipo de dato asignado (bien sea explícitamente o implícitamente). Dependiendo del tipo de dato el lenguaje de programación sabrá la estructura que utilizará para su almacenamiento, las restricciones en los valores que puede aceptar, la precisión del mismo, etc.

En PL/SQL contamos con todos los **tipos de datos simples** utilizados en SQL y algunos más. En este apartado vamos a enumerar los más utilizados.

Numéricos.

- ✓ `BINARY INTEGER`: Tipo de dato numérico cuyo rango es de -2147483647 .. 2147483647. PL/SQL además define algunos subtipos de éste: `NATURAL`, `NATURALN`, `POSITIVE`, `POSITIVEN`, `SIGNTYPE`.

- ✓ `NUMBER`: Tipo de dato numérico para almacenar números racionales. Podemos especificar su escala (-84 .. 127) y su precisión (1 .. 38). La escala indica cuándo se redondea y hacia dónde. Ejemplos. escala=2: 8.234 -> 8.23, escala=-3: 7689 -> 8000. PL/SQL también define algunos subtipos como: `DEC`, `DECIMAL`, `DOUBLE PRECISION`, `FLOAT`, `INTEGER`, `INT`, `NUMERIC`, `REAL`, `SMALLINT`.
- ✓ `PLS_INTEGER`: Tipo de datos numérico cuyo rango es el mismo que el del tipo de dato `BINARY INTEGER`, pero que su representación es distinta por lo que las operaciones aritméticas llevadas a cabo con los mismos serán más eficientes que en los dos casos anteriores.

Alfanuméricos.

- ✓ `CHAR(n)`: Array de n caracteres, máximo 2000 bytes. Si no especificamos longitud sería 1.
- ✓ `LONG`: Array de caracteres con un máximo de 32760 bytes.
- ✓ `RAW`: Array de bytes con un número máximo de 2000.
- ✓ `LONG RAW`: Array de bytes con un máximo de 32760.
- ✓ `VARCHAR2`: Tipo de dato para almacenar cadenas de longitud variable con un máximo de 32760.

Grandes objetos.

- ✓ `BFILE`: Puntero a un fichero del Sistema Operativo.
- ✓ `BLOB`: Objeto binario con una capacidad de 4 GB.
- ✓ `CLOB`: Objeto carácter con una capacidad de 2 GB.

Otros.

- ✓ `BOOLEAN`: `TRUE`/`FALSE`.
- ✓ `DATE`: Tipo de dato para almacenar valores día/hora desde el 1 enero de 4712 a.c. hasta el 31 diciembre de 4712 d.c.

Hemos visto los tipos de datos simples más usuales. Los tipos de datos compuestos los dejaremos para posteriores apartados.

En el siguiente enlace podrás ampliar información sobre los tipos de datos de los que disponemos en PL/SQL.
http://www.devjoker.com/contenidos/Tutorial-PLSQL/60/Tipos-de-datos-en-PLSQL.aspx

En PL/SQL cuando vamos a trabajar con enteros es preferible utilizar el tipo de dato BINARY_INTEGER, en vez de PLS_INTEGER.

○ Verdadero ◉ **Falso**

Nuestros programas serán más eficientes al utilizar este tipo de dato (PLS_INTEGER) debido a su representación interna.

2.2.1.- Subtipos.

Cuántas veces no has deseado cambiarle el nombre a las cosas por alguno más común para ti. Precisamente, esa es la posibilidad que nos ofrece PL/SQL con la utilización de los subtipos.
PL/SQL nos permite definir subtipos de tipos de datos para darles un nombre diferente y así aumentar la legibilidad de nuestros programas. Los tipos de operaciones aplicables a estos subtipos serán las mismas que los tipos de datos de los que proceden. La sintaxis será:

```
SUBTYPE subtipo IS tipo_base;
```

Donde subtipo será el nombre que le demos a nuestro subtipo y tipo_base será cualquier tipo de dato en PL/SQL.
A la hora de especificar el tipo base, podemos utilizar el modificador `%TYPE` para indicar el tipo de dato de una variable o de una columna de la base de datos y `%ROWTYPE` para especificar el tipo de un cursor o tabla de una base de datos.

```
SUBTYPE id_familia IS familias.identificador%TYPE;
```

```
SUBTYPE agente IS agentes%ROWTYPE;
```

Los subtipos no podemos restringirlos, pero podemos usar un truco para conseguir el mismo efecto y es por medio de una variable auxiliar:

```
SUBTYPE apodo IS varchar2(20);          --ilegal
aux varchar2(20);
SUBTYPE apodo IS aux%TYPE;              --legal
```

Los subtipos son intercambiables con su tipo base. También son intercambiables si tienen el mismo tipo base o si su tipo base pertenece a la misma familia:

```
DECLARE
     SUBTYPE numero IS NUMBER;
     numero_tres_digitos NUMBER(3);
     mi_numero_de_la_suerte numero;
     SUBTYPE encontrado IS BOOLEAN;
     SUBTYPE resultado IS BOOLEAN;
     lo_he_encontrado encontrado;
     resultado_busqueda resultado;
     SUBTYPE literal IS CHAR;
     SUBTYPE sentencia IS VARCHAR2;
     literal nulo literal;
     sentencia vacia sentencia;
BEGIN
     ...
     numero_tres_digitos := mi_numero_de_la_suerte;     --legal
     ...
     lo he encontrado := resultado busqueda;            --legal
     ...
     sentencia_vacia := literal_nulo;                   --legal
     ...
END;
```

Indica la afirmación correcta.

- ○ Los subtipos lo único que hacen es añadir complejidad a nuestros programas.
- ○ No hay manera de restringir los subtipos con respecto a su tipo base.
- ◉ **Podemos definir un subtipo cuyo tipo base sea una tabla de la base de datos.**
- ○ Podemos definir un subtipo de una variable pero no de una columna de la base de datos.

2.2.2.- Variables y constantes.

Llevamos un buen rato hablando de tipos de datos, variables e incluso de constantes y te estarás preguntando cuál es la forma adecuada de definirlas. En este apartado vamos a ver las diferentes posibilidades a la hora de definirlas y dejaremos para el apartado siguiente ver cuál es el lugar adecuado para hacerlo dentro de un bloque PL/SQL.

Para declarar variables o constantes pondremos el nombre de la variable, seguido del tipo de datos y opcionalmente una asignación. Si es una constante antepondremos la palabra `CONSTANT` al tipo de dato (lo que querrá decir que no podemos cambiar su valor). Podremos sustituir el operador de asignación en las declaraciones por la palabra reservada `DEFAULT`. También podremos forzar a que no sea nula utilizando la palabra `NOT NULL` después del tipo y antes de la asignación. Si restringimos una variable con `NOT NULL` deberemos asignarle un valor al declararla, de lo contrario PL/SQL lanzará la excepción `VALUE_ERROR` (no te asustes que más adelante veremos lo que son las excepciones, pero como adelanto te diré que es un error en tiempo de ejecución).

```
id SMALLINT;
hoy DATE := sysdate;
pi CONSTANT REAL:= 3.1415;
id SMALLINT NOT NULL;          --ilegal, no está inicializada
id SMALLINT NOT NULL := 9999;  --legal
```

El alcance y la visibilidad de las variables en PL/SQL será el típico de los lenguajes estructurados basados en bloques, aunque eso lo veremos más adelante.

Conversión de tipos.

Aunque en PL/SQL existe la conversión implícita de tipos para tipos parecidos, siempre es aconsejable utilizar la conversión explícita de tipos por medio de funciones de conversión (`TO_CHAR`, `TO_DATE`, `TO_NUMBER`, …) y así evitar resultados inesperados.

Precedencia de operadores.

Al igual que en nuestro lenguaje matemático se utiliza una precedencia entre operadores a la hora de realizar las operaciones aritméticas, en PL/SQL también se establece dicha precedencia para evitar confusiones. Si dos operadores tienen la misma precedencia lo aconsejable es utilizar los paréntesis (al igual que hacemos en nuestro lenguaje matemático) para alterar la precedencia de los mismos ya que las operaciones encerradas entre paréntesis tienen mayor precedencia. En la tabla siguiente se muestra la precedencia de los operadores de mayor a menor.

Precedencia de operadores.	
Operador.	**Operación.**
****, NOT**	Exponenciación, negación lógica.
+, -	Identidad, negación.
***, /**	Multiplicación, división.
+, -, ||	Suma, resta y concatenación.
=, ¡=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparaciones.
AND	Conjunción lógica
OR	Disyunción lógica.

Rellena el hueco con el resultado de las siguientes operaciones.

5+3*22** es igual a: .	**17**
23+6/3** es igual a: .	**10**
2(3+6/3)** es igual a: .	**32**

2.3.- El bloque PL/SQL.

Ya hemos visto las unidades léxicas que componen PL/SQL, los tipos de datos que podemos utilizar y cómo se definen las variables y las constantes. Ahora vamos a ver la unidad básica en PL/SQL que es el bloque.

Un bloque PL/SQL consta de tres zonas:

- ✓ **Declaraciones**: definiciones de variables, constantes, cursores y excepciones.
- ✓ **Proceso**: zona donde se realizará el proceso en sí, conteniendo las sentencias ejecutables.
- ✓ **Excepciones**: zona de manejo de errores en tiempo de ejecución.

La sintaxis es la siguiente:

```
[DECLARE
     [Declaración de variables, constantes, cursores y excepciones]]
BEGIN
     [Sentencias ejecutables]
[EXCEPTION
     Manejadores de excepciones]
END;
```

Los bloques PL/SQL pueden anidarse a cualquier nivel. Como hemos comentado anteriormente el ámbito y la visibilidad de las variables es la normal en un lenguaje procedimental. Por ejemplo, en el siguiente fragmento de código se declara la variable aux en ambos bloques, pero en el bloque anidado aux con valor igual a 10 actúa de variable global y aux con valor igual a 5 actúa como

variable local, por lo que en la comparación evaluaría a `FALSE`, ya que al tener el mismo nombre la visibilidad dominante sería la de la variable local.

```
DECLARE
     aux number := 10;
BEGIN
     DECLARE
          aux number := 5;
     BEGIN
          ...
          IF aux = 10 THEN     --evalúa a FALSE, no entraría
          ...
     END;
END;
```

En el siguiente enlace podrás ampliar información sobre el ámbito y la visibilidad de las variables en PL/SQL.
http://franjaviersanzsanz.comyr.com/programacion_pl_sql.php#enlace2_2_4

En PL/SQL el bloque es la unidad básica, por lo que éstos no pueden anidarse.

○ Verdadero ◉ **Falso**

2.4.- Estructuras de contro.

En la vida constantemente tenemos que tomar decisiones que hacen que llevemos a cabo unas acciones u otras dependiendo de unas circunstancias o repetir una serie de acciones un número dado de veces o hasta que se cumpla una condición. En PL/SQL también podemos imitar estas situaciones por medio de las estructuras de control que son sentencias que nos permiten manejar el flujo de control de nuestro programa, y éstas son dos: condicionales e iterativas.

Control condicional.

Las estructuras de control condicional nos permiten llevar a cabo una acción u otra dependiendo de una condición. Vemos sus diferentes variantes:

- ✓ `IF-THEN`: Forma más simple de las sentencias de control condicional. Si la evaluación de la condición es `TRUE`, entonces se ejecuta la secuencia de sentencias encerradas entre el `THEN` y el final de la sentencia.

Sentencia IF-THEN.	
Sintaxis.	**Ejemplo.**

IF condicion THEN
secuencia_de_sentencias;
END IF;
	`IF (b<>0) THEN`
	`c:=a/b;`
	`END IF;`

- ✓ `IF-THEN-ELSE`: Con esta forma de la sentencia ejecutaremos la primera secuencia de sentencias si la condición evalúa a `TRUE` y en caso contrario ejecutaremos la segunda secuencia de sentencias.

Sentencia IF-THEN-ELSE.	
Sintaxis.	**Ejemplo.**

IF condicion THEN
Secuencia_de_sentencias1;
ELSE
Secuencia_de_sentencias2;
END IF;
	`IF (b<>0) THEN`
	`c:=a/b;`
	`END IF;`

- ✓ `IF-THEN-ELSIF`: Con esta última forma de la sentencia condicional podemos hacer una selección múltiple. Si la evaluación de la condición 1 da `TRUE`, ejecutamos la secuencia de sentencias 1, sino evaluamos la condición 2. Si esta evalúa a `TRUE` ejecutamos la secuencia de sentencias 2 y así

para todos los `ELSIF` que haya. El último `ELSE` es opcional y es por si no se cumple ninguna de las condiciones anteriores.

Sentencia IF-THEN-ELSIF.

Sintaxis.	Ejemplo.

IF condicion1 THEN
Secuencia_de_sentencias1;
ELSIF condicion2 THEN
Secuencia_de_sentencias2;
...
[ELSE
Secuencia_de_sentencias;]
END IF;
	`IF (operacion = 'SUMA') THEN`
	`  resultado := arg1 + arg2;`
	`ELSIF (operacion = 'RESTA') THEN`
	`  resultado := arg1 - arg2;`
	`ELSIF (operacion = 'PRODUCTO') THEN`
	`  resultado := arg1 * arg2;`
	`ELSIF (arg2 <> 0) AND (operacion = 'DIVISION') THEN`
	`  resultado := arg1 / arg2;`
	`ELSE`
	`  RAISE operacion_no_permitida;`
	`END IF;`

En PL/SQL no existen sentencias que nos permitan tomar una acción u otra dependiendo de una condición.

○ Verdadero ◉ **Falso**

Para eso existen las sentencias de control condicional.

Ya que hemos visto las estructuras de control condicional, veamos ahora las estructuras de control iterativo.

Control iterativo.

Estas estructuras nos permiten ejecutar una secuencia de sentencias un determinado número de veces.

✓ `LOOP`: La forma más simple es el bucle infinito, cuya sintaxis es:

```
LOOP
   secuencia_de_sentencias;
END LOOP;
```

✓ `EXIT`: Con esta sentencia forzamos a un bucle a terminar y pasa el control a la siguiente sentencia después del bucle. Un `EXIT` no fuerza la salida de un bloque PL/SQL, sólo la salida del bucle.

```
LOOP
   ...
   IF encontrado = TRUE THEN
      EXIT;
   END IF;
END LOOP;
```

✓ `EXIT WHEN condicion`: Fuerza a salir del bucle cuando se cumple una determinada condición.

```
LOOP
   ...
   EXIT WHEN encontrado;
END LOOP;
```

✓ `WHILE LOOP`: Este tipo de bucle ejecuta la secuencia de sentencias mientras la condición sea cierta.

Sentencia WHILE LOOP.

Sintaxis.	Ejemplo.

WHILE condicion LOOP
Secuencia_de_sentencias;
END LOOP;
	`WHILE (not encontrado) LOOP`
	`  ...`
	`END LOOP;`

✓ `FOR LOOP`: Este bucle itera mientras el contador se encuentre en el rango definido.

Sentencia FOR LOOP.

Sintaxis.	Ejemplo.

FOR contador IN [REVERSE]
limite_inferior..limite_superior LOOP
Secuencia_de_sentencias;
END LOOP;
	`FOR i IN 1..3 LOOP --i=1, i=2, i=3`
	`...`
	`END LOOP;`
	
	`SELECT count(*) INTO num_agentes FROM agentes;`
	`FOR i IN 1..num agentes LOOP`
	`...`
	`END LOOP;`
	
	`FOR i IN REVERSE 1..3 LOOP --i=3, i=2, i=1`
	`...`
	`END LOOP;`

Al utilizar REVERSE en un bucle FOR, en el rango debemos poner el número mayor el primero y el menor el último.

○ Verdadero ◉ **Falso**

2.5.- Manejo de errores.

Muchas veces te habrá pasado que surgen situaciones inesperadas con las que no contabas y a las que tienes que hacer frente. Pues cuando programamos con PL/SQL pasa lo mismo, que a veces tenemos que manejar errores debidos a situaciones diversas. Vamos a ver cómo tratarlos.

Cualquier situación de error es llamada **excepción** en PL/SQL. Cuando se detecta un error, una excepción es lanzada, es decir, la ejecución normal se para y el control se transfiere a la parte de manejo de excepciones. La parte de manejo de excepciones es la parte etiquetada como `EXCEPTION` y constará de sentencias para el manejo de dichas excepciones, llamadas **manejadores de excepciones**.

Manejadores de excepciones.	
Sintaxis.	**Ejemplo.**

WHEN nombre_excepcion THEN
<sentencias para su manejo>
....
WHEN OTHERS THEN
<sentencias para su manejo>
	`DECLARE`
	`supervisor agentes%ROWTYPE;`
	`BEGIN`
	`SELECT * INTO supervisor FROM agentes`
	`WHERE categoria = 2 AND oficina = 3;`
	`...`
	`EXCEPTION`
	`WHEN NO DATA FOUND THEN`
	`--Manejamos el no haber encontrado datos`
	`WHEN OTHERS THEN`
	`--Manejamos cualquier error inesperado`
	`END;`

La parte `OTHERS` captura cualquier excepción no capturada.

Las excepciones pueden estar definidas por el usuario o definidas internamente. Las excepciones predefinidas se lanzarán automáticamente asociadas a un error de Oracle. Las excepciones definidas por el usuario deberán definirse y lanzarse explícitamente.

En PL/SQL nosotros podemos definir nuestras propias excepciones en la parte `DECLARE` de cualquier bloque. Estas excepciones podemos lanzarlas explícitamente por medio de la sentencia `RAISE nombre_excepción`.

Excepciones definidas por el usuario.	
Sintaxis.	**Ejemplo.**
DECLARE	`DECLARE`

```
    nombre_excepcion EXCEPTION;
BEGIN
    ...
    RAISE nombre_excepcion;
    ...
END;
```

```
    categoria_erronea EXCEPTION;
BEGIN
    ...
    IF categoria<0 OR categoria>3 THEN
        RAISE categoria erronea;
    END IF;
    ...
EXCEPTION
    WHEN categoria_erronea THEN
        --manejamos la categoria errónea
END;
```

En el siguiente enlace podrás ver las diferentes excepciones predefinidas en Oracle, junto a su código de error asociado (que luego veremos lo que es) y una explicación de cuándo son lanzadas.

Excepciones predefinidas en Oracle.		
Excepción.	**SQLCODE**	**Lanzada cuando ...**
ACCES_INTO_NULL	-6530	Intentamos asignar valor a atributos de objetos no inicializados.
COLECTION_IS_NULL	-6531	Intentamos asignar valor a elementos de colecciones no inicializadas, o acceder a métodos distintos de `EXISTS`.
CURSOR_ALREADY_OPEN	-6511	Intentamos abrir un cursor ya abierto.
DUP_VAL_ON_INDEX	-1	Índice único violado.
INVALID_CURSOR	-1001	Intentamos hacer una operación con un cursor que no está abierto.
INVALID_NUMBER	-1722	Conversión de cadena a número falla.
LOGIN_DENIED	-1403	El usuario y/o contraseña para conectarnos a Oracle no es válido.
NO_DATA_FOUND	+100	Una sentencia `SELECT` no devuelve valores, o intentamos acceder a un elemento borrado de una tabla anidada.
NOT_LOGGED_ON	-1012	No estamos conectados a Oracle.
PROGRAM_ERROR	-6501	Ha ocurrido un error interno en PL/SQL.
ROWTYPE_MISMATCH	-6504	Diferentes tipos en la asignación de 2 cursores.
STORAGE_ERROR	-6500	Memoria corrupta.
SUBSCRIPT_BEYOND_COUNT	-6533	El índice al que intentamos acceder en una colección sobrepasa su límite superior.
SUBSCRIPT_OUTSIDE_LIMIT	-6532	Intentamos acceder a un rango no válido dentro de una colección (-1 por ejemplo).
TIMEOUT_ON_RESOURCE	-51	Un timeout ocurre mientras Oracle espera por un recurso.
TOO_MANY_ROWS	-1422	Una sentencia `SELECT...INTO...` devuelve más de una fila.
VALUE_ERROR	-6502	Ocurre un error de conversión, aritmético, de truncado o de restricción de tamaño.
ZERO_DIVIDE	-1476	Intentamos dividir un número por 0.

Ahora que ya sabemos lo que son las excepciones, cómo capturarlas y manejarlas y cómo definir y lanzar las nuestras propias. Es la hora de comentar algunos detalles sobre el uso de las mismas.

- ✓ El alcance de una excepción sigue las mismas reglas que el de una variable, por lo que si nosotros redefinimos una excepción que ya es global para el bloque, la definición local prevalecerá y no podremos capturar esa excepción a menos que el bloque en la que estaba definida esa excepción fuese un bloque nombrado, y podremos capturarla usando la sintaxis: `nombre_bloque.nombre_excepcion`.

- ✓ Las excepciones predefinidas están definidas globalmente. No necesitamos (ni debemos) redefinir las excepciones predefinidas.

```
DECLARE
     no_data_found EXCEPTION;
BEGIN
     SELECT * INTO ...
EXCEPTION
     WHEN no data found THEN     --captura la excepción local, no
                          --la global
END;
```

- ✓ Cuando manejamos una excepción no podemos continuar por la siguiente sentencia a la que la lanzó.

```
DECLARE
     ...
BEGIN
     ...
     INSERT INTO familias VALUES
(id fam, nom fam, NULL, oficina);
     INSERT INTO agentes VALUES
(id_ag, nom_ag, login, password, 0, 0, id_fam, NULL);
     ...
EXCEPTION
     WHEN DUP VAL ON INDEX THEN
          --manejamos la excepción debida a que el nombre de
          --la familia ya existe, pero no podemos continuar por
          --el INSERT INTO agentes, a no ser que lo pongamos
          --explícitamente en el manejador
END;
```

- ✓ Pero sí podemos encerrar la sentencia dentro de un bloque, y ahí capturar las posibles excepciones, para continuar con las siguientes sentencias.

```
DECLARE
     id fam NUMBER;
     nom fam VARCHAR2(40);
     oficina NUMBER;
     id_ag NUMBER;
     nom_ag VARCHAR2(60);
     usuario VARCHAR2(20);
     clave VARCHAR2(20);
BEGIN
     ...
     BEGIN
          INSERT INTO familias VALUES (id_fam, nom_fam, NULL, oficina);
     EXCEPTION
          WHEN DUP_VAL_ON_INDEX THEN
SELECT identificador INTO id fam FROM familias WHERE nombre = nom fam;
     END;
     INSERT INTO agentes VALUES (id_ag, nom_ag, login, password, 1, 1, id_fam, null);
     ...
END;
```

Ejercicio resuelto

Supongamos que queremos reintentar una transacción hasta que no nos dé ningún error. Para ello deberemos encapsular la transacción en un bloque y capturar en éste las posibles excepciones. El bloque lo metemos en un bucle y así se reintentará la transacción hasta que sea posible llevarla a cabo.

Respuesta:

```
DECLARE
     id_fam NUMBER;
     nombre VARCHAR2(40);
     oficina NUMBER;
BEGIN
     ...
     LOOP
          BEGIN
               SAVEPOINT inicio;
```

```
                INSERT INTO familias VALUES
(id_fam, nombre, NULL, oficina);
                ...
                COMMIT;
                EXIT;
           EXCEPTION
                WHEN DUP_VAL_ON_INDEX THEN
                     ROLLBACK TO inicio;
                     id_fam := id_fam + 1;
           END;
     END LOOP;
     ...
END;
```

Continuemos viendo algunos detalles a tener en cuenta, relativos al uso de las excepciones.

- ✓ Cuando ejecutamos varias sentencias seguidas del mismo tipo y queremos capturar alguna posible excepción debida al tipo de sentencia, podemos encapsular cada sentencia en un bloque y manejar en cada bloque la excepción, o podemos utilizar una variable localizadora para saber qué sentencia ha sido la que ha lanzado la excepción (aunque de esta manera no podremos continuar por la siguiente sentencia).

```
DECLARE
     sentencia NUMBER := 0;
BEGIN
     ...
     SELECT * FROM agentes ...
     sentencia := 1;
     SELECT * FROM familias ...
     sentencia := 2;
     SELECT * FROM oficinas ...
     ...
EXCEPTION
     WHEN NO_DATA_FOUND THEN
          IF sentencia = 0 THEN
               RAISE agente_no_encontrado;
          ELSIF sentencia = 1 THEN
               RAISE familia no encontrada;
          ELSIF sentencia = 2 THEN
               RAISE oficina_no_encontrada;
          END IF;
END;
```

- ✓ Si la excepción es capturada por un manejador de excepción apropiado, ésta es tratada y posteriormente el control es devuelto al bloque superior. Si la excepción no es capturada y no existe bloque superior, el control se devolverá al entorno. También puede darse que la excepción sea manejada en un bloque superior a falta de manejadores para ella en los bloques internos, la excepción se propaga de un bloque al superior y así hasta que sea manejada o no queden bloques superiores con lo que el control se devuelve al entorno. Una excepción también puede ser relanzada en un manejador.

Todas las excepciones están predefinidas y nosotros no podemos definir nuevas excepciones.

○ Verdadero ◉ **Falso**

Las excepciones definidas por el usuario deben ser lanzadas explícitamente.

◉ **Verdadero** ○ Falso

Es obligatorio declarar todas las excepciones predefinidas que vamos a usar en nuestros bloques.

○ Verdadero ◉ **Falso**

Oracle también permite que nosotros lancemos nuestros propios mensajes de error a las aplicaciones y asociarlos a un código de error que Oracle reserva, para no interferir con los demás códigos de error. Lo hacemos por medio del procedimiento:

```
RAISE_APPLICATION_ERROR(error_number, message [, (TRUE|FALSE)]);
```

Donde error_number es un entero negativo comprendido entre –20000..-20999 y message es una cadena que devolvemos a la aplicación. El tercer parámetro especifica si el error se coloca en la pila de errores (`TRUE`) o se vacía la pila y se coloca únicamente el nuestro (`FALSE`). Sólo podemos llamar a este procedimiento desde un subprograma.

No hay excepciones predefinidas asociadas a todos los posibles errores de Oracle, por lo que nosotros podremos asociar excepciones definidas por nosotros a errores Oracle, por medio de la directiva al compilador (o pseudoinstrucción):

```
PRAGMA_INIT( nombre_excepcion, error_Oracle )
```

Donde nombre_excepcion es el nombre de una excepción definida anteriormente, y error_Oracle es el número negativo asociado al error.

```
DECLARE
     no_null EXCEPTION;
     PRAGMA EXCEPTION_INIT(no_null, -1400);
     id familias.identificador%TYPE;
     nombre familias.nombre%TYPE;
BEGIN
     ...
     nombre := NULL;
     ...
     INSERT INTO familias VALUES (id, nombre, null, null);
EXCEPTION
     WHEN no null THEN
          ...
END;
```

Oracle asocia 2 funciones para comprobar la ejecución de cualquier sentencia. `SQLCODE` nos devuelve el código de error y `SQLERRM` devuelve el mensaje de error asociado. Si una sentencia es ejecutada correctamente, `SQLCODE` nos devuelve 0 y en caso contrario devolverá un número negativo asociado al error (excepto `NO_DATA_FOUND` que tiene asociado el +100).

```
DECLARE
cod number;
msg varchar2(100);
BEGIN
...
EXCEPTION
WHEN OTHERS THEN
     cod := SQLCODE;
     msg := SUBSTR(SQLERRM, 1, 1000);
     INSERT INTO errores VALUES (cod, msg);
END;
```

De las siguientes afirmaciones marca las que creas que son correctas.

- [x] **Podemos lanzar nuestros propios mensajes de error a las aplicaciones.**
- [ ] Podemos acceder al código de error generado por la ejecución de una sentencia pero no a su mensaje asociado.
- [x] **Podemos asociar excepciones definidas por nosotros a códigos de error de Oracle.**

3.- Tipos de datos compuestos.

Caso práctico

__María__, gracias a la ayuda de Juan, ha comprendido muy bien el manejo básico de PL/SQL. __Juan__ le comenta que en la mayoría de los lenguajes de programación, además de los tipos de datos simples, existen tipos de datos compuestos que le dan mucha más versatibilidad a los mismos y una gran potencia. __Juan__ no conoce bien si PL/SQL dispone de estos tipos de datos y si dispone de ellos ¿de cuáles?

__María__ y __Juan__ van a hablar con __Ada__ para ver si les puede aclarar un poco la situación y dar unas pequeñas pautas para empezar. __Ada__ les comenta que claro que PL/SQL cuenta con este tipo de datos, pero que hoy tiene una reunión importantísima y que tiene que terminar de preparársela, por lo que los remite al capítulo sobre tipos de datos compuestos del manual que les pasó. __Juan__ y __María__ le dicen que no se preocupe y que ya verá como a la vuelta de esa reunión son capaces de saber cuáles son e incluso de dominarlos. __Ada__, para motivarlos, les dice que si es así los invita a un aperitivo a su vuelta. Así que __Juan__ y __María__ se ponen con el capítulo de tipos de datos compuestos del manual para ver si pueden sorprender a __Ada__ a su vuelta.

En el capítulo anterior, entre otras cosas, hemos conocido los tipos de datos simples con los que cuenta PL/SQL. Pero dependiendo de la complejidad de los problemas, necesitamos disponer de otras estructuras en las que apoyarnos para poder modelar nuestro problema. En este capítulo nos vamos a centrar en conocer los tipos de datos complejos que nos ofrece PL/SQL y cómo utilizarlos para así poder sacarle el mayor partido posible.

"Todo lo complejo puede dividirse en partes simples".*René Descartes.*

3.1.- Registros.

El uso de los registros es algo muy común en los lenguajes de programación. PL/SQL también nos ofrece este tipo de datos. En este apartado veremos qué son y cómo definirlos y utilizarlos.

Un registro es un grupo de elementos relacionados almacenados en campos, cada uno de los cuales tiene su propio nombre y tipo de dato.

Por ejemplo, una dirección podría ser un registro con campos como calle, número, piso, puerta, código postal, ciudad, provincia y país. Los registros hacen que la información sea más fácil de organizar y representar. Para declarar un registro seguiremos la siguiente sintaxis:

```
TYPE nombre_tipo IS RECORD (decl_campo[, decl_campo] ...);
```

donde:

```
decl_campo := nombre tipo [[NOT NULL] {:=|DEFAULT} expresion]
```

El tipo del campo será cualquier tipo de dato válido en PL/SQL excepto REF CURSOR. La expresión será cualquier expresión que evalúe al tipo de dato del campo.

```
TYPE direccion IS RECORD
(
   calle          VARCHAR2(50),
   numero          INTEGER(4),
   piso           INTEGER(4),
   puerta          VARCHAR2(2),
   codigo_postal     INTEGER(5),
   ciudad          VARCHAR2(30),
   provincia      VARCHAR2(20),
   pais           VARCHAR2(20) := 'España'
);
mi_direccion direccion;
```

Para acceder a los campos usaremos la notación del punto.

```
...
mi_direccion.calle := 'Ramirez Arellano';
```

```
mi_direccion.numero := 15;
...
```

Para asignar un registro a otro, éstos deben ser del mismo tipo, no basta que tengan el mismo número de campos y éstos emparejen uno a uno. Tampoco podemos comparar registros aunque sean del mismo tipo, ni tampoco comprobar si éstos son nulos. Podemos hacer `SELECT` en registros, pero no podemos hacer `INSERT` desde registros.

```
DECLARE
TYPE familia IS RECORD
(
     identificador   NUMBER,
     nombre          VARCHAR2(40),
     padre           NUMBER,
     oficina         NUMBER
);
TYPE familia_aux IS RECORD
(
     identificador   NUMBER,
     nombre          VARCHAR2(40),
     padre           NUMBER,
     oficina         NUMBER
     );
     SUBTYPE familia_fila IS familias%ROWTYPE;
     mi_fam familia;
     mi fam aux familia aux;
     mi_fam_fila familia_fila;
BEGIN
...
mi_fam := mi_fam_aux;                              --ilegal
mi_fam := mi_fam_fila;                             --legal
IF mi fam IS NULL THEN ...                         --ilegal
IF mi fam = mi fam fila THEN ...                   --ilegal
SELECT * INTO mi_fam FROM familias ...             --legal
INSERT INTO familias VALUES (mi_fam_fila);         --ilegal
...
END;
```

Un registro se puede asignar a otro siempre que tenga el mismo número de campos y éstos emparejen uno a uno.

○ Verdadero ◉ **Falso**

3.2.- Colecciones. Arrays de longitud variable.

Una colección es un grupo ordenado de elementos, todos del mismo tipo. Cada elemento tiene un subíndice único que determina su posición en la colección.

En PL/SQL las colecciones sólo pueden tener una dimensión. PL/SQL ofrece 2 clases de colecciones: arrays de longitud variable y tablas anidadas.

Arrays de longitud variable.

Los elementos del tipo `VARRAY` son los llamados arrays de longitud variable. Son como los arrays de cualquier otro lenguaje de programación, pero con la salvedad de que a la hora de declararlos, nosotros indicamos su tamaño máximo y el array podrá ir creciendo dinámicamente hasta alcanzar ese tamaño. Un `VARRAY` siempre tiene un límite inferior igual a 1 y un límite superior igual al tamaño máximo.

Para declarar un `VARRAY` usaremos la sintaxis:

```
TYPE nombre IS {VARRAY | VARYING} (tamaño_máximo) OF tipo_elementos [NOT NULL];
```

Donde tamaño_máximo será un entero positivo y tipo_elementos será cualquier tipo de dato válido en PL/SQL, excepto `BINARY INTEGER`, `BOOLEAN`, `LONG`, `LONG RAW`, `NATURAL`, `NATURALN`, `NCHAR`, `NCLOB`, `NVARCHAR2`, objetos que tengan como atributos `TABLE` o `VARRAY`, `PLS INTEGER`, `POSITIVE`, `POSITIVEN`, `SIGNTYPE`, `STRING`, `TABLE`, `VARRAY`. Si tipo_elementos es un registro, todos los campos deberían ser de un tipo escalar.

Cuando definimos un `VARRAY`, éste es automáticamente nulo, por lo que para empezar a utilizarlo deberemos inicializarlo. Para ello podemos usar un constructor:

```
TYPE familias hijas IS VARRAY(100) OF familia;
familias_hijas1 familias_hijas := familias_hijas( familia(100, 'Fam100', 10, null), ...,
familia(105, 'Fam105', 10, null));
```

También podemos usar constructores vacíos.

```
familias_hijas2 familias_hijas := familias_hijas();
```

Para referenciar elementos en un `VARRAY` utilizaremos la sintaxis `nombre_colección(subíndice)`. Si una función devuelve un `VARRAY`, podemos usar la sintaxis:

```
nombre funcion(lista parametros)(subindice).
IF familias_hijas1(i).identificador = 100 THEN ...
IF dame_familias_hijas(10)(i).identificador = 100 THEN ...
```

Un VARRAY puede ser asignado a otro si ambos son del mismo tipo.

```
DECLARE
TYPE tabla1 IS VARRAY(10) OF NUMBER;
TYPE tabla2 IS VARRAY(10) OF NUMBER;
mi_tabla1 tabla1 := tabla1();
mi tabla2 tabla2 := tabla2();
mi tabla tabla1 := tabla1();
BEGIN
...
mi_tabla := mi_tabla1;          --legal
mi_tabla1 := mi_tabla2;     --ilegal
...
END;
```

Para extender un `VARRAY` usaremos el método `EXTEND`. Sin parámetros, extendemos en 1 elemento nulo el `VARRAY. EXTEND(n)` añade n elementos nulos al `VARRAY` y `EXTEND(n,i)` añade n copias del i-ésimo elemento.

`COUNT` nos dirá el número de elementos del `VARRAY`. `LIMIT` nos dice el tamaño máximo del `VARRAY`. `FIRST` siempre será 1. `LAST` siempre será igual a `COUNT`. `PRIOR` y `NEXT` devolverá el antecesor y el sucesor del elemento.

Al trabajar con `VARRAY` podemos hacer que salte alguna de las siguientes excepciones, debidas a un mal uso de los mismos: `COLECTION IS NULL`, `SUBSCRIPT BEYOND COUNT`, `SUBSCRIPT OUTSIDE LIMIT` y `VALUE_ERROR`.

Ejemplos de uso de los VARRAY.

Extender un VARRAY.

```
DECLARE
  TYPE tab_num IS
VARRAY(10) OF NUMBER;
  mi_tab tab_num;
BEGIN
  mi_tab := tab_num();
  FOR i IN 1..10 LOOP
    mi_tab.EXTEND;
```

Consultar propiedades VARRAY.

```
DECLARE
   TYPE numeros IS VARRAY(20)
OF NUMBER;
   tabla_numeros numeros :=
numeros();
   num NUMBER;
BEGIN
   num :=
tabla numeros.COUNT;
--num := 0
   FOR i IN 1..10 LOOP
```

Posibles excepciones.

```
DECLARE
  TYPE numeros IS VARRAY(20) OF
INTEGER;
   v_numeros numeros := numeros(
10, 20, 30, 40 );
    v_enteros numeros;
BEGIN
    v enteros(1) := 15; --
lanzaría COLECTION IS NULL
    v_numeros(5) := 20; --
lanzaría SUBSCRIPT_BEYOND_COUNT
```

```
    mi_tab(i) :=
calcular_elemento(i);
  END LOOP;
  ...
END;
```

```
      tabla_numeros.EXTEND;
      tabla_numeros(i) := i;
   END LOOP;
   num :=
tabla numeros.COUNT; --num :=
10
   num :=
tabla_numeros.LIMIT; --num :=
20
   num :=
tabla numeros.FIRST; --num :=
1;
   num := tabla_numeros.LAST;
--num := 10;
   ...
END;
```

```
    v_numeros(-1) := 5; --
lanzaría SUBSCRIPT_OUTSIDE_LIMIT
   v_numeros('A') := 25; --
lanzaría VALUE_ERROR
    ....
END;
```

Indica, de entre las siguientes, cuál es la afirmación correcta referida a VARRAY.

- ○ Un VARRAY no hace falta inicializarlo.
- ○ COUNT y LIMIT siempre nos devolverán el mismo valor.
- ◉ LAST y COUNT siempre nos devolverán el mismo valor.

3.2.1.- Colecciones. Tablas anidadas.

Las tablas anidadas son colecciones de elementos, que no tienen límite superior fijo, y pueden aumentar dinámicamente su tamaño. Además podemos borrar elementos individuales.

Para declararlos utilizaremos la siguiente sintaxis:

```
TYPE nombre IS TABLE OF tipo_elementos [NOT NULL];
```

Donde tipo_elementos tendrá las mismas restricciones que para los `VARRAY`.

Al igual que pasaba con los `VARRAY`, al declarar una tabla anidada, ésta es automáticamente nula, por lo que deberemos inicializarla antes de usarla.

```
TYPE hijos IS TABLE OF agente;
hijos_fam hijos := hijos( agente(...) ...);
```

También podemos usar un constructor nulo.

Para referenciar elementos usamos la misma sintaxis que para los `VARRAY`.

Para extender una tabla usamos `EXTEND` exactamente igual que para los `VARRAY`. `COUNT` nos dirá el número de elementos, que no tiene por qué coincidir con `LAST`. `LIMIT` no tiene sentido y devuelve `NULL`. `EXISTS(n)` devuelve `TRUEsi` el elemento existe, y `FALSE` en otro caso (el elemento ha sido borrado). `FIRST` devuelve el primer elemento que no siempre será 1, ya que hemos podido borrar elementos del principio. `LAST` devuelve el último elemento. `PRIOR yNEXT` nos dicen el antecesor y sucesor del elemento (ignorando elementos borrados). `TRIM` sin argumentos borra un elemento del final de la tabla. `TRIM(n)` borra n elementos del final de la tabla. `TRIM` opera en el tamaño interno, por lo que si encuentra un elemento borrado con `DELETE`, lo incluye para ser eliminado de la colección. `DELETE(n)` borra el n-ésimo elemento. `DELETE(n, m)` borra del elemento n al m. Si después de hacer `DELETE`, consultamos si el elemento existe nos devolverá `FALSE`.

Al trabajar con tablas anidadas podemos hacer que salte alguna de las siguientes excepciones, debidas a un mal uso de las mismas: `COLECTION_IS_NULL`, `NO_DATA_FOUND`, `SUBSCRIPT_BEYOND_COUNT` y `VALUE ERROR`.

Ejemplos de uso de las tablas anidadas.

Diferentes operaciones sobre tablas anidadas.

```
DECLARE
TYPE numeros IS TABLE OF NUMBER;
tabla_numeros numeros := numeros();
num NUMBER;
BEGIN
num := tabla_numeros.COUNT;    --num := 0
FOR i IN 1..10 LOOP
   tabla_numeros.EXTEND;
   tabla_numeros(i) := i;
END LOOP;
num := tabla_numeros.COUNT;    --num := 10
tabla_numeros.DELETE(10);
num := tabla_numeros.LAST;    --num := 9
num := tabla_numeros.FIRST;    --num := 1
tabla_numeros.DELETE(1);
num := tabla_numeros.FIRST;    --num := 2
FOR i IN 1..4 LOOP
   tabla_numeros.DELETE(2*i);
END LOOP;
num := tabla_numeros.COUNT;    --num := 4
num := tabla_numeros.LAST;    --num := 9
...
END;
```

Posibles excepciones en su uso.

```
DECLARE
TYPE numeros IS TABLE OF NUMBER;
tabla_num numeros := numeros();
tabla1 numeros;
BEGIN
tabla1(5) := 0;     --lanzaría COLECTION_IS_NULL
tabla_num.EXTEND(5);
tabla_num.DELETE(4);
tabla_num(4) := 3;     --lanzaría NO_DATA_FOUND
tabla_num(6) := 10;     --lanzaría
SUBSCRIPT BEYOND COUNT
tabla_num(-1) := 0;     --lanzaría
SUBSCRIPT_OUTSIDE_LIMIT
tabla_num('y') := 5;--lanzaría VALUE_ERROR
END;
```

Las tablas anidadas podemos hacer que crezcan dinámicamente, pero no podemos borrar elementos.

○ Verdadero ◉ Falso

3.3.- Cursores.

En los apartados anteriores hemos visto algunos tipos de datos compuestos cuyo uso es común en otros lenguajes de programación. Sin embargo, en este apartado vamos a ver un tipo de dato, que aunque se puede asemejar a otros que ya conozcas, su uso es exclusivo en la programación de las bases de datos y que es el **cursor**.

Un cursor no es más que una estructura que almacena el conjunto de filas devuelto por una consulta a la base de datos.

Oracle usa áreas de trabajo para ejecutar sentencias SQL y almacenar la información procesada. Hay 2 clases de cursores: **implícitos** y **explícitos**. PL/SQL declara implícitamente un cursor para todas las sentencias SQL de manipulación de datos, incluyendo consultas que devuelven una sola fila. Para las consultas que devuelven más de una fila, se debe declarar explícitamente un cursor para procesar las filas individualmente.

En este primer apartado vamos a hablar de los cursores implícitos y de los atributos de un cursor (estos atributos tienen sentido con los cursores explícitos, pero los introducimos aquí para ir abriendo boca), para luego pasar a ver los cursores explícitos y terminaremos hablando de los cursores variables.

Cursores implícitos.

Oracle abre implícitamente un cursor para procesar cada sentencia SQL que no esté asociada con un cursor declarado explícitamente.
Con un cursor implícito no podemos usar las sentencias `OPEN`, `FETCH` y `CLOSE` para controlar el cursor. Pero sí podemos usar los atributos del cursor para obtener información sobre las sentencias SQL más recientemente ejecutadas.

Atributos de un cursor.

Cada cursor tiene 4 atributos que podemos usar para obtener información sobre la ejecución del mismo o sobre los datos. Estos atributos pueden ser usados en PL/SQL, pero no en SQL. Aunque estos atributos se refieren en general a cursores explícitos y tienen que ver con las operaciones que hayamos realizado con el cursor, es deseable comentarlas aquí y en el siguiente apartado tomarán pleno sentido.

- ✓ `%FOUND`: Después de que el cursor esté abierto y antes del primer `FETCH`, `%FOUND` devuelve `NULL`. Después del primer `FETCH`, `%FOUND` devolverá `TRUE` si el último `FETCH` ha devuelto una fila y `FALSE` en caso contrario. Para cursores implícitos `%FOUND` devuelve `TRUE` si un `INSERT`, `UPDATE` o `DELETE` afectan a una o más de una fila, o un `SELECT ... INTO ...` devuelve una o más filas. En otro caso `%FOUND` devuelve `FALSE`.
- ✓ `%NOTFOUND`: Es lógicamente lo contrario a `%FOUND`.
- ✓ `%ISOPEN`: Evalúa a `TRUE` si el cursor está abierto y `FALSE` en caso contrario. Para cursores implícitos, como Oracle los cierra automáticamente, `%ISOPEN` evalúa siempre a `FALSE`.
- ✓ `%ROWCOUNT`: Para un cursor abierto y antes del primer `FETCH`, `%ROWCOUNT` evalúa a 0. Después de cada `FETCH`, `%ROWCOUNT` es incrementado y evalúa al número de filas que hemos procesado. Para cursores implícitos `%ROWCOUNT` evalúa al número de filas afectadas por un `INSERT`, `UPDATE` o `DELETE` o el número de filas devueltas por un `SELECT ... INTO ...`

Aunque todavía no hemos visto las operaciones que se pueden realizar con un cursor explícito, es conveniente que te vayas familiarizando con la evaluación de sus atributos según las operaciones que hayamos realizado con el cursor y que tomarán pleno sentido cuando veamos el siguiente apartado.

Evaluación de los atributos de un cursor explícito según las operaciones realizadas con él.

Operación realizada.	%FOUND	%NOTFOUND	%ISOPEN	%ROWCOUNT
Antes del OPEN	Excepción.	Excepción.	FALSE	Excepción.
Después del OPEN	NULL	NULL	TRUE	0
Antes del primer FETCH	NULL	NULL	TRUE	0
Después del primer FETCH	TRUE	FALSE	TRUE	1
Antes de los siguientes FETCH	TRUE	FALSE	TRUE	1
Después de los siguientes FETCH	TRUE	FALSE	TRUE	Depende datos.
Antes del último FETCH	TRUE	FALSE	TRUE	Depende datos.
Después del último FETCH	FALSE	TRUE	TRUE	Depende datos.
Antes del CLOSE	FALSE	TRUE	TRUE	Depende datos.
Después del CLOSE	Excepción.	Excepción.	FALSE	Excepción.

3.3.1.- Cursores explícitos.

Cuando una consulta devuelve múltiples filas, podemos declarar explícitamente un cursor para procesar las filas devueltas. Cuando declaramos un cursor, lo que hacemos es darle un nombre y asociarle una consulta usando la siguiente sintaxis:

```
CURSOR nombre_cursor [(parametro [, parametro] ...)] [RETURN tipo_devuelto] IS
sentencia_select;
```

Donde tipo_devuelto debe representar un registro o una fila de una tabla de la base de datos, y parámetro sigue la siguiente sintaxis:

```
parametro := nombre_parametro [IN] tipo_dato [{:= | DEFAULT} expresion]
```

Ejemplos:

```
CURSOR cAgentes IS SELECT * FROM agentes;
CURSOR cFamilias RETURN familias%ROWTYPE IS SELECT * FROM familias WHERE ...
```

Además, como hemos visto en la declaración, un cursor puede tomar parámetros, los cuales pueden aparecer en la consulta asociada como si fuesen constantes. Los parámetros serán de entrada, un cursor no puede devolver valores en los parámetros actuales. A un parámetro de un cursor no podemos imponerle la restricción `NOT NULL`.

```
CURSOR c1 (cat INTEGER DEFAULT 0) IS SELECT * FROM agentes WHERE categoria = cat;
```

Cuando abrimos un cursor, lo que se hace es ejecutar la consulta asociada e identificar el conjunto resultado, que serán todas las filas que emparejen con el criterio de búsqueda de la consulta. Para abrir un cursor usamos la sintaxis:

```
OPEN nombre_cursor [(parametro [, parametro] ...)];
```

Ejemplos:

```
OPEN cAgentes;
OPEN c1(1);
OPEN c1;
```

La sentencia `FETCH` devuelve una fila del conjunto resultado. Después de cada `FETCH`, el cursor avanza a la próxima fila en el conjunto resultado.

```
FETCH cFamilias INTO mi_id, mi_nom, mi_fam, mi_ofi;
```

Para cada valor de columna devuelto por la consulta asociada al cursor, debe haber una variable que se corresponda en la lista de variables después del `INTO`.

Para procesar un cursor entero deberemos hacerlo por medio de un bucle.

```
BEGIN
...
OPEN cFamilias;
LOOP
     FETCH cFamilias INTO mi_id, mi_nom, mi_fam, mi_ofi;
     EXIT WHEN cFamilias%NOTFOUND;
     ...
END LOOP;
...
END;
```

Una vez procesado el cursor, deberemos cerrarlo, con lo que desabilitamos el cursor y el conjunto resultado queda indefinido.

```
CLOSE cFamilias;
```

Una vez cerrado el cursor podemos reabrirlo, pero cualquier otra operación que hagamos con el cursor cerrado lanzará la excepción `INVALID CURSOR`.

También podemos simplificar la operación de procesamiento de un cursor, por medio de los bucles para cursores, los cuales declaran implícitamente una variable índice definida como `%ROWTYPE` para el cursor, abren el cursor, se van trayendo los valores de cada fila del cursor, almacenándolas en la variable índice, y finalmente cierran el cursor.

```
BEGIN
     ...
     FOR cFamilias rec IN cFamilias LOOP
          --Procesamos las filas accediendo a
          --cFamilias_rec.identificador, cFamilias_rec.nombre,
          --cFamilias_rec.familia, ...
     END LOOP;
     ...
END;
```

En PL/SQL los cursores son abiertos al definirlos.

○ Verdadero ◉ Falso

Debemos abrirlos por medio de la sentencia OPEN.

3.3.2.- Cursores variables.

Oracle, además de los cursores vistos anteriormente, nos permite definir cursores variables que son como punteros a cursores, por lo que podemos usarlos para referirnos a cualquier tipo de consulta. Los cursores serían estáticos y los cursores variables serían dinámicos.

Para declarar un cursor variable debemos seguir 2 pasos:

- ✓ Definir un tipo `REF CURSOR` y entonces declarar una variable de ese tipo.

```
TYPE tipo_cursor IS REF CURSOR RETURN agentes%ROWTYPE;
cAgentes tipo_cursor;
```

- ✓ Una vez definido el cursor variable debemos asociarlo a una consulta (notar que esto no se hace en la parte declarativa, sino dinámicamente en la parte de ejecución) y esto lo hacemos con la sentencia `OPEN-FOR` utilizando la siguiente sintaxis:

```
OPEN nombre_variable_cursor FOR sentencia_select;

OPEN cAgentes FOR SELECT * FROM agentes WHERE oficina = 1;
```

Un cursor variable no puede tomar parámetros. Podemos usar los atributos de los cursores para cursores variables.

Además, podemos usar varios `OPEN-FOR` para abrir el mismo cursor variable para diferentes consultas. No necesitamos cerrarlo antes de reabrirlo. Cuando abrimos un cursor variable para una consulta diferente, la consulta previa se pierde.

Una vez abierto el cursor variable, su manejo es idéntico a un cursor. Usaremos `FETCH` para traernos las filas, usaremos sus atributos para hacer comprobaciones y lo cerraremos cuando dejemos de usarlo.

```
DECLARE
TYPE cursor_Agentes IS REF CURSOR RETURN agentes%ROWTYPE;
cAgentes cursor Agentes;
agente cAgentes%ROWTYPE;
BEGIN
...
OPEN cAgentes FOR SELECT * FROM agentes WHERE oficina = 1;
LOOP
     FETCH cAgentes INTO agente;
     EXIT WHEN cAgentes%NOTFOUND;
     ...
END LOOP;
CLOSE cAgentes;
...
END;
```

A los cursores variables no podemos pasarles parámetros al abrirlos.

(•) **Verdadero** ( ) Falso

Los cursores variables se abren exactamente igual que los cursores explícitos.

( ) Verdadero (•) **Falso**

Debemos abrirlo por medio de la sentencia OPEN-FOR con la que le asociamos la consulta.

4.- Abstracción en PL/SQL.

Caso práctico

__María__, gracias a la ayuda de __Juan__, tiene bastante claro cómo programar en PL/SQL pero no entiende muy bien cómo integrar todo esto con la base de datos de juegos on-line. Sabe que puede utilizar unos tipos de datos, que hay unas estructuras de control y que se pueden manejar los errores que surjan, pero lo que no sabe es cómo y donde utilizar todo eso.

__Juan__ le explica que lo que hasta ahora ha aprendido es el comienzo, pero que ahora viene lo bueno y que será donde le va a encontrar pleno sentido a lo aprendido anteriormente. Le explica que PL/SQL permite crear funciones y procedimientos y además agruparlos en paquetes y que eso será lo que realmente van a hacer con la base de datos de juegos on-line. Deberán ver qué es lo que utilizan más comúnmente e implementarlo en PL/SQL utilizando funciones y procedimientos según convenga. __Juan__ la tranquiliza y le dice que lo primero que va a hacer es explicarle cómo se escriben dichas funciones y procedimientos y luego pasarán a implementar alguno y que así verá la potencia real de PL/SQL. __María__ se queda más tranquila y está deseando implementar esa primera función o procedimiento que le resolverá la gran duda que tiene.

Hoy día cualquier lenguaje de programación permite definir diferentes grados de abstracción en sus programas. La abstracción permite a los programadores crear unidades lógicas y posteriormente utilizarlas pensando en qué hace y no en cómo lo hace. La abstracción se consigue utilizando funciones, procedimientos, librerías, objetos, etc.

PL/SQL nos permite definir funciones y procedimientos. Además nos permite agrupar todas aquellas que tengan relación en paquetes. También permite la utilización de objetos. Todo esto es lo que veremos en este apartado y conseguiremos darle modularidad a nuestras aplicaciones, aumentar la reusabilidad y mantenimiento del código y añadir grados de abstracción a los problemas.

En los siguientes enlaces puedes ampliar información sobre la abstracción en programación.
http://es.wikipedia.org/wiki/Abstracci%C3%B3n_%28inform%C3%A1tica%29
http://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos

4.1.- Subprogramas.

Los subprogramas son bloques nombrados a los cuales les podemos pasar parámetros y los podemos invocar. Además, los subprogramas pueden estar almacenados en la base de datos o estar encerrados en otros bloques. Si el programa está almacenado en la base de datos, podremos invocarlo si tenemos permisos suficientes y si está encerrado en otro bloque lo podremos invocar si tenemos visibilidad sobre el mismo.

Hay dos clases de subprogramas: las funciones y los procedimientos. Las funciones devuelven un valor y los procedimientos no.

Para declarar un subprograma utilizamos la siguiente sintaxis:

Sintaxis para la declaración de subprogramas.

Funciones.	Procedimientos.

FUNCTION nombre [(parametro [, parametro] ...)]
RETURN tipo_dato IS
[declaraciones_locales]
BEGIN
sentencias_ejecutables
[EXCEPTION
manejadores_de_excepciones]
END [nombre];
	`PROCEDURE nombre [( parametro [, parametro] ... )] IS`
	`  [declaraciones_locales]`
	`BEGIN`
	`sentencias ejecutables`
	`[EXCEPTION`
	`manejadores_de_excepciones]`
	`END [nombre];`

Donde:

```
parametro := nombre_parametro [IN|OUT|IN OUT] tipo_dato [{:=|DEFAULT} expresion]
```

Algunas consideraciones que debes tener en cuenta son las siguientes:

- ✓ No podemos imponer una restricción `NOT NULL` a un parámetro.
- ✓ No podemos especificar una restricción del tipo:

```
PROCEDURE KK(a NUMBER(10)) IS ...          --ilegal
```

- ✓ Una función siempre debe acabar con la sentencia `RETURN`.

Podemos definir subprogramas al final de la parte declarativa de cualquier bloque. En Oracle, cualquier identificador debe estar declarado antes de usarse, y eso mismo pasa con los subprogramas, por lo que deberemos declararlos antes de usarlos.

```
DECLARE
hijos NUMBER;
FUNCTION hijos familia( id familia NUMBER )
     RETURN NUMBER IS
     hijos NUMBER;
BEGIN
     SELECT COUNT(*) INTO hijos FROM agentes
          WHERE familia = id_familia;
     RETURN hijos;
END hijos_familia;
BEGIN
...
END;
```

Si quisiéramos definir subprogramas en orden alfabético o lógico, o necesitamos definir subprogramas mutuamente recursivos (uno llama a otro, y éste a su vez llama al anterior), deberemos usar la definición hacia delante, para evitar errores de compilación.

```
DECLARE
     PROCEDURE calculo(...);          --declaración hacia delante
     --Definimos subprogramas agrupados lógicamente
     PROCEDURE inicio(...) IS
     BEGIN
          ...
          calculo(...);
          ...
     END;
     ...
BEGIN
     ...
END;
```

Una función siempre debe devolver un valor.

(•) **Verdadero** ( ) Falso

En PL/SQL no podemos definir subprogramas mutuamente recursivos.

( ) Verdadero (•) **Falso**

4.1.1.- Almacenar subprogramas en la base de datos.

Para almacenar un subprograma en la base de datos utilizaremos la misma sintaxis que para declararlo, anteponiendo `CREATE [OR REPLACE]` a `PROCEDURE` o `FUNCTION`, y finalizando el subprograma con una línea que simplemente contendrá el carácter '**/**' para indicarle a Oracle que termina ahí. Si especificamos `OR REPLACE` y el subprograma ya existía, éste será reemplazado. Si no lo especificamos y el subprograma ya existe, Oracle nos devolverá un error indicando que el nombre ya está siendo utilizado por otro objeto de la base de datos.

```
CREATE OR REPLACE FUNCTION hijos_familia(id_familia NUMBER)
RETURN NUMBER IS
```

```
hijos NUMBER;
BEGIN
SELECT COUNT(*) INTO hijos FROM agentes
WHERE familia = id familia;
RETURN hijos;
END;
/
```

Cuando los subprogramas son almacenados en la base de datos, para ellos no podemos utilizar las declaraciones hacia delante, por lo que cualquier subprograma almacenado en la base de datos deberá conocer todos los subprogramas que utilice.

Para invocar un subprograma usaremos la sintaxis:

```
nombre_procedimiento [(parametro [,parametro] ...)];
variable := nombre_funcion [(parametro [, parametro] ...)];

BEGIN
...
hijos := hijos_familia(10);
...
END;
```

Si el subprograma está almacenado en la base de datos y queremos invocarlo desde SQL*Plus usaremos la sintaxis:

```
EXECUTE nombre_procedimiento [(parametros)];
EXECUTE :variable_sql := nombre_funcion [(parametros)];
```

Cuando almacenamos un subprograma en la base de datos éste es compilado antes. Si hay algún error se nos informará de los mismos y deberemos corregirlos por medio de la cláusula `OR REPLACE`, antes de que el subprograma pueda ser utilizado.

Hay varias vistas del diccionario de datos que nos ayudan a llevar un control de los subprogramas, tanto para ver su código, como los errores de compilación. También hay algunos comandos de SQL*Plus que nos ayudan a hacer lo mismo pero de forma algo menos engorrosa.

Vistas y comandos asociados a los subprogramas.		
Información almacenada.	**Vista del diccionario.**	**Comando.**
Código fuente.	USER_SOURCE	DESCRIBE
Errores de compilación.	USER_ERRORS	SHOW ERRORS
Ocupación de memoria.	USER_OBJECT_SIZE	

También existe la vista `USER OBJECTS` de la cual podemos obtener los nombres de todos los subprogramas almacenados.

Una vez que hemos almacenado un subprograma en la base de datos podemos consultar su código mediante la vista USER_OBJECTS.

○ Verdadero ◉ **Falso**

4.1.2.- Parámetros de los subprogramas.

Ahora vamos a profundizar un poco más en los parámetros que aceptan los subprogramas y cómo se los podemos pasar a la hora de invocarlos.

Las variables pasadas como parámetros a un subprograma son llamadas **parámetros actuales**. Las variables referenciadas en la especificación del subprograma como parámetros, son llamadas **parámetros formales**.

Cuando llamamos a un subprograma, los parámetros actuales podemos escribirlos utilizando notación posicional o notación nombrada, es decir, la asociación entre parámetros actuales y formales podemos hacerla por posición o por nombre.

En la notación posicional, el primer parámetro actual se asocia con el primer parámetro formal, el segundo con el segundo, y así para el resto. En la notación nombrada usamos el operador => para asociar el parámetro actual al parámetro formal. También podemos usar notación mixta.

Los parámetros pueden ser de **entrada** al subprograma, de **salida**, o de **entrada/salida**. Por defecto si a un parámetro no le especificamos el modo, éste será de entrada. Si el parámetro es de salida o de entrada/salida, el parámetro actual debe ser una variable.

Un parámetro de entrada permite que pasemos valores al subprograma y no puede ser modificado en el cuerpo del subprograma. El parámetro actual pasado a un subprograma como parámetro formal de entrada puede ser una constante o una variable.

Un parámetro de salida permite devolver valores y dentro del subprograma actúa como variable no inicializada. El parámetro formal debe ser siempre una variable.

Un parámetro de entrada-salida se utiliza para pasar valores al subprograma y/o para recibirlos, por lo que un parámetro formal que actúe como parámetro actual de entrada-salida siempre deberá ser una variable.

Los parámetros de entrada los podemos inicializar a un valor por defecto. Si un subprograma tiene un parámetro inicializado con un valor por defecto, podemos invocarlo prescindiendo del parámetro y aceptando el valor por defecto o pasando el parámetro y sobreescribiendo el valor por defecto. Si queremos prescindir de un parámetro colocado entre medias de otros, deberemos usar notación nombrada o si los parámetros restantes también tienen valor por defecto, omitirlos todos.

Veamos ahora algunos ejemplos sobre el paso de parámetros a nuestros subprogramas.

Notación mixta.

```
DECLARE
     PROCEDURE prueba( formal1 NUMBER, formal2 VARCHAR2) IS
     BEGIN
          ...
     END;
     actual1 NUMBER;
     actual2 VARCHAR2;
BEGIN
     ...
     prueba(actual1, actual2);               --posicional
     prueba(formal2=>actual2,formal1=>actual1);     --nombrada
     prueba(actual1, formal2=>actual2);          --mixta
END;
```

Parámetros de entrada.

```
FUNCTION categoria( id_agente IN NUMBER )
RETURN NUMBER IS
     cat NUMBER;
BEGIN
     ...
     SELECT categoria INTO cat FROM agentes
WHERE identificador = id agente;
RETURN cat;
EXCEPTION
     WHEN NO_DATA_FOUND THEN
          id_agente := -1; --ilegal, parámetro de entrada
END;
```

Parámetros de salida.

```
PROCEDURE nombre( id_agente NUMBER, nombre OUT VARCHAR2) IS
```

```
BEGIN
     IF (nombre = 'LUIS') THEN     --error de sintaxis
     END IF;
     ...
END;
```

Parámetros con valor por defecto de los que podemos prescindir.

```
DECLARE
     SUBTYPE familia IS familias%ROWTYPE;
     SUBTYPE agente IS agentes%ROWTYPE;
     SUBTYPE tabla agentes IS TABLE OF agente;
     familia1 familia;
     familia2 familia;
     hijos_fam tabla_agentes;
     FUNCTION inserta_familia( mi_familia familia,
          mis_agentes tabla_agentes := tabla_agentes() )
RETURN NUMBER IS
     BEGIN
          INSERT INTO familias VALUES (mi_familia);
          FOR i IN 1..mis agentes.COUNT LOOP
               IF   (mis agentes(i).oficina   IS   NOT   NULL)   or   (mis agentes(i).familia   ¡=
mi familia.identificador) THEN
                    ROLLBACK;
                    RETURN -1;
               END IF;
               INSERT INTO agentes VALUES (mis_agentes(i));
          END LOOP;
          COMMIT;
          RETURN 0;
     EXCEPTION
          WHEN DUP_VAL_ON_INDEX THEN
               ROLLABACK;
               RETURN -1;
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END inserta_familia;
BEGIN
     ...
     resultado := inserta familia( familia1 );
     ...
     resultado := inserta_familia( familia2, hijos_fam2 );
     ...
END;
```

Indica de entre las siguientes afirmaciones las que creas que son correctas.

- [x] **En PL/SQL podemos usar la notación posicional para pasar parámetros**
- [ ] No existen los parámetros de salida ya que para eso existen las funciones
- [x] **Los parámetros de entrada los podemos inicializar a un valor por defecto**

En PL/SQL podemos usar la notación posicional para pasar parámetros, los parámetros de entrada podemos inicializarlos a un valor por defecto y también podemos utilizar los parámetros de salida

4.1.3.- Sobrecarga de subprogramas y recursividad.

PL/SQL también nos ofrece la posibilidad de sobrecargar funciones o procedimientos, es decir, llamar con el mismo nombre subprogramas que realizan el mismo cometido y que aceptan distinto número y/o tipo de parámetros. No podemos sobrecargar subprogramas que aceptan el mismo número y tipo de parámetros y sólo difieren en el modo. Tampoco podemos sobrecargar subprogramas con el mismo número de parámetros y que los tipos de los parámetros sean diferentes, pero de la misma familia, o sean subtipos basados en la misma familia.

A continuación podrás ver un ejemplo de una función que es sobrecarga tres veces dependiendo del tipo de parámetros que acepta.

```
DECLARE
     TYPE agente IS agentes%ROWTYPE;
     TYPE familia IS familias%ROWTYPE;
     TYPE tAgentes IS TABLE OF agente;
```

```
     TYPE tFamilias IS TABLE OF familia;

     FUNCTION inserta familia( mi familia familia )
     RETURN NUMBER IS
     BEGIN
          INSERT   INTO   familias   VALUES   (mi_familia.identificador,   mi_familia.nombre,
mi_familia.familia, mi_familia.oficina );
          COMMIT;
          RETURN 0;
     EXCEPTION
          WHEN DUP_VAL_ON_INDEX THEN
               ROLLBACK;
               RETURN -1;
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END inserta_familia;

     FUNCTION inserta familia( mi familia familia, hijas tFamilias )
     RETURN NUMBER IS
     BEGIN
          INSERT   INTO   familias   VALUES   (mi famila.identificador,   mi familia.nombre,
mi_familia.familia, mi_familia.oficina );
          IF (hijas IS NOT NULL) THEN
               FOR i IN 1..hijas.COUNT LOOP
                    IF   (hijas(i).oficina   IS   NOT   NULL)   or   (hijas(i).familia   ¡=
mi familia.identificador) THEN
                         ROLLBACK;
                         RETURN -1;
                    END IF;
                    INSERT  INTO  familias  VALUES  (hijas(i).identificador,  hijas(i).nombre,
hijas(i).familia, hijas(i).oficina );
               END LOOP;
          END IF;
          COMMIT;
          RETURN 0;
     EXCEPTION
          WHEN DUP VAL ON INDEX THEN
               ROLLBACK;
               RETURN -1;
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN -1;
     END inserta familia;

     FUNCTION inserta_familia( mi_familia familia, hijos tAgentes )
     RETURN NUMBER IS
     BEGIN
          INSERT   INTO   familias   VALUES   (mi familia.identificador,   mi familia.nombre,
mi_familia.familia, mi_familia.oficina );
          IF (hijos IS NOT NULL) THEN
               FOR i IN 1..hijos.COUNT LOOP
                    IF   (hijos(i).oficina   IS   NOT   NULL)   or   (hijos(i).familia   ¡=
mi familia.identificador) THEN
                         ROLLBACK;
                         RETURN -1;
                    END IF;
                    INSERT  INTO  agentes  VALUES  (hijos(i).identificador,  hijos(i).nombre,
hijos(i).usuario,  hijos(i).clave,  hijos(i).habilidad,  hijos(i).categoria,  hijos(i).familia,
hijos(i).oficina );
               END LOOP;
          END IF;
          COMMIT;
          RETURN 0;
     EXCEPTION
          WHEN DUP_VAL_ON_INDEX THEN
               ROLLBACK;
               RETURN -1;
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN -1;
     END inserta familias;

     mi familia familia;
     mi_familia1 familia;
     familias_hijas tFamilias;
     mi_familia2 familia;
```

```
     hijos tAgentes;
BEGIN
     ...
     resultado := inserta familia(mi familia);
     ...
     resultado := inserta_familia(mi_familia1, familias_hijas);
     ...
     resultado := inserta_familia(mi_familia2, hijos);
     ...
END;
```

PL/SQL también nos ofrece la posibilidad de utilizar la recursión en nuestros subprogramas. Un subprograma es recursivo si éste se invoca a él mismo.

En el siguiente enlace podrás ampliar información sobre la recursividad.
http://es.wikipedia.org/wiki/Recursi%C3%B3n
A continuación te mostramos un ejemplo del uso de la recursividad en nuestros subprogramas.

```
DECLARE
     TYPE agente IS agentes%ROWTYPE;
     TYPE tAgentes IS TABLE OF agente;
     hijos10 tAgentes;
     PROCEDURE dame_hijos( id_familia NUMBER,
                    hijos IN OUT tAgentes ) IS
          CURSOR hijas IS SELECT identificador FROM familias WHERE familia = id familia;
hija NUMBER;
     CURSOR cHijos IS SELECT * FROM agentes WHERE familia = id_familia;
hijo agente;
     BEGIN
          --Si la tabla no está inicializada -> la inicializamos
          IF hijos IS NULL THEN
               hijos = tAgentes();
          END IF;
          --Metemos en la tabla los hijos directos de esta familia
          OPEN cHijos;
          LOOP
               FETCH cHijos INTO hijo;
               EXIT WHEN cHijos%NOTFOUND;
               hijos.EXTEND;
               hijos(hijos.LAST) := hijo;
          END LOOP;
          CLOSE cHijos;
          --Hacemos lo mismo para todas las familias hijas de la actual
          OPEN hijas;
          LOOP
               FETCH hijas INTO hija;
               EXIT WHEN hijas%NOTFOUND;
               dame hijos( hija, hijos );
          END LOOP;
          CLOSE hijas;
     END dame_hijos;
BEGIN
     ...
     dame hijos( 10, hijos10 );
     ...
END;
```

En PL/SQL no podemos sobrecargar subprogramas que aceptan el mismo número y tipo de parámetros, pero sólo difieren en el modo.

(•) **Verdadero** ( ) Falso

En PL/SQL no podemos utilizar la recursión y tenemos que imitarla mediante la iteración.

( ) Verdadero (•) **Falso**

4.2.- Paquetes.

Un paquete es un objeto que agrupa tipos, elementos y subprogramas. Suelen tener dos partes: la especificación y el cuerpo, aunque algunas veces el cuerpo no es necesario.

En la parte de especificación declararemos la interfaz del paquete con nuestra aplicación y en el cuerpo es donde implementaremos esa interfaz.

Para crear un paquete usaremos la siguiente sintaxis:

```
CREATE [OR REPLACE] PACKAGE nombre AS
     [declaraciones públicas y especificación de subprogramas]
END [nombre];

CREATE [OR REPLACE] PACKAGE BODY nombre AS
     [declaraciones privadas y cuerpo de los subprogramas especificados]
[BEGIN
     sentencias de inicialización]
END [nombre];
```

La parte de inicialización sólo se ejecuta una vez, la primera vez que el paquete es referenciado.

Veamos ahora un ejemplo de un paquete que agrupa las principales tareas que realizamos con nuestra base de datos de ejemplo.

```
CREATE OR REPLACE PACKAGE call_center AS     --inicialización
     --Definimos los tipos que utilizaremos
     SUBTYPE agente IS agentes%ROWTYPE;
     SUBTYPE familia IS familias%ROWTYPE;
     SUBTYPE oficina IS oficinas%ROWTYPE;
     TYPE tAgentes IS TABLE OF agente;
     TYPE tFamilias IS TABLE OF familia;
     TYPE tOficinas IS TABLE OF oficina;

     --Definimos las excepciones propias
     referencia no encontrada exception;
     referencia encontrada exception;
     no null exception;
     PRAGMA EXCEPTION_INIT(referencia_no_encontrada, -2291);
     PRAGMA EXCEPTION_INIT(referencia_encontrada, -2292);
     PRAGMA EXCEPTION_INIT(no null, -1400);

     --Definimos los errores que vamos a tratar
     todo_bien               CONSTANT NUMBER := 0;
     elemento_existente          CONSTANT NUMBER:= -1;
     elemento_inexistente          CONSTANT NUMBER:= -2;
     padre_existente          CONSTANT NUMBER:= -3;
     padre inexistente          CONSTANT NUMBER:= -4;
     no null violado          CONSTANT NUMBER:= -5;
     operacion_no_permitida          CONSTANT NUMBER:= -6;

     --Definimos los subprogramas públicos
     --Nos devuelve la oficina padre de un agente
     PROCEDURE oficina padre( mi agente agente, padre OUT oficina );

     --Nos devuelve la oficina padre de una familia
     PROCEDURE oficina_padre( mi_familia familia, padre OUT oficina );

     --Nos da los hijos de una familia
     PROCEDURE dame hijos( mi familia familia, hijos IN OUT tAgentes );

     --Nos da los hijos de una oficina
     PROCEDURE dame_hijos( mi_oficina oficina, hijos IN OUT tAgentes );

     --Inserta un agente
     FUNCTION inserta agente ( mi agente agente )
     RETURN NUMBER;

     --Inserta una familia
     FUNCTION inserta familia( mi familia familia )
     RETURN NUMBER;

     --Inserta una oficina
```

```
     FUNCTION inserta_oficina ( mi_oficina oficina )
     RETURN NUMBER;

     --Borramos una oficina
     FUNCTION borra_oficina( id_oficina NUMBER )
     RETURN NUMBER;

     --Borramos una familia
     FUNCTION borra_familia( id_familia NUMBER )
     RETURN NUMBER;

     --Borramos un agente
     FUNCTION borra agente( id agente NUMBER )
     RETURN NUMBER;
END call center;
/

CREATE OR REPLACE PACKAGE BODY call_center AS          --cuerpo
     --Implemento las funciones definidas en la especificación

     --Nos devuelve la oficina padre de un agente
     PROCEDURE oficina padre( mi agente agente, padre OUT oficina ) IS
          mi_familia familia;
     BEGIN
          IF (mi_agente.oficina IS NOT NULL) THEN
               SELECT * INTO padre FROM oficinas
               WHERE identificador = mi agente.oficina;
          ELSE
               SELECT * INTO mi_familia FROM familias
               WHERE identificador = mi_agente.familia;
               oficina_padre( mi_familia, padre );
          END IF;
     EXCEPTION
          WHEN OTHERS THEN
               padre := NULL;
     END oficina_padre;

     --Nos devuelve la oficina padre de una familia
     PROCEDURE oficina padre( mi familia familia, padre OUT oficina ) IS
          madre familia;
     BEGIN
          IF (mi familia.oficina IS NOT NULL) THEN
               SELECT * INTO padre FROM oficinas
               WHERE identificador = mi familia.oficina;
          ELSE
               SELECT * INTO madre FROM familias
               WHERE identificador = mi_familia.familia;
               oficina padre( madre, padre );
          END IF;
     EXCEPTION
          WHEN OTHERS THEN
               padre := NULL;
     END oficina_padre;

     --Nos da los hijos de una familia
     PROCEDURE dame hijos( mi familia familia, hijos IN OUT tAgentes ) IS
          CURSOR cHijos IS SELECT * FROM agentes
          WHERE familia = mi_familia.identificador;
          CURSOR cHijas IS SELECT * FROM familias
          WHERE familia = mi_familia.identificador;
          hijo agente;
          hija familia;
     BEGIN
          --inicializamos la tabla si no lo está
          if (hijos IS NULL) THEN
               hijos := tAgentes();
          END IF;
          --metemos en la tabla los hijos directos
          OPEN cHijos;
          LOOP
               FETCH cHijos INTO hijo;
               EXIT WHEN cHijos%NOTFOUND;
               hijos.EXTEND;
               hijos(hijos.LAST) := hijo;
          END LOOP;
          CLOSE cHijos;
          --hacemos lo mismo para las familias hijas
```

```
          OPEN cHijas;
          LOOP
               FETCH cHijas INTO hija;
               EXIT WHEN cHijas%NOTFOUND;
               dame_hijos( hija, hijos );
          END LOOP;
          CLOSE cHijas;
     EXCEPTION
          WHEN OTHERS THEN
               hijos := tAgentes();
     END dame_hijos;

     --Nos da los hijos de una oficina
     PROCEDURE dame hijos( mi oficina oficina, hijos IN OUT tAgentes ) IS
          CURSOR cHijos IS SELECT * FROM agentes
          WHERE oficina = mi oficina.identificador;
          CURSOR cHijas IS SELECT * FROM familias
          WHERE oficina = mi_oficina.identificador;
          hijo agente;
          hija familia;
     BEGIN
          --inicializamos la tabla si no lo está
          if (hijos IS NULL) THEN
               hijos := tAgentes();
          END IF;
          --metemos en la tabla los hijos directos
          OPEN cHijos;
          LOOP
               FETCH cHijos INTO hijo;
               EXIT WHEN cHijos%NOTFOUND;
               hijos.EXTEND;
               hijos(hijos.LAST) := hijo;
          END LOOP;
          CLOSE cHijos;
          --hacemos lo mismo para las familias hijas
          OPEN cHijas;
          LOOP
               FETCH cHijas INTO hija;
               EXIT WHEN cHijas%NOTFOUND;
               dame_hijos( hija, hijos );
          END LOOP;
          CLOSE cHijas;
     EXCEPTION
          WHEN OTHERS THEN
               hijos := tAgentes();
     END dame_hijos;

     --Inserta un agente
     FUNCTION inserta agente ( mi agente agente )
     RETURN NUMBER IS
     BEGIN
          IF (mi_agente.familia IS NULL and mi_agente.oficina IS NULL) THEN
               RETURN operacion_no_permitida;
          END IF;
          IF (mi agente.familia IS NOT NULL and mi agente.oficina IS NOT NULL) THEN
               RETURN operacion no permitida;
          END IF;
          INSERT    INTO    agentes    VALUES    (mi_agente.identificador,    mi_agente.nombre,
mi_agente.usuario,       mi_agente.clave,       mi_agente.habilidad,       mi_agente.categoria,
mi_agente.familia, mi_agente.oficina );
          COMMIT;
          RETURN todo_bien;
     EXCEPTION
          WHEN referencia_no_encontrada THEN
               ROLLBACK;
               RETURN padre_inexistente;
          WHEN no null THEN
               ROLLBACK;
               RETURN no_null_violado;
          WHEN DUP VAL ON INDEX THEN
               ROLLBACK;
               RETURN elemento existente;
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END inserta_agente;
```

```
     --Inserta una familia
     FUNCTION inserta_familia( mi_familia familia )
     RETURN NUMBER IS
     BEGIN
          IF (mi_familia.familia IS NULL and mi_familia.oficina IS NULL) THEN
               RETURN operacion_no_permitida;
          END IF;
          IF (mi_familia.familia IS NOT NULL and mi_familia.oficina IS NOT NULL) THEN
               RETURN operacion_no_permitida;
          END IF;
          INSERT   INTO   familias   VALUES   (   mi_familia.identificador,   mi_familia.nombre,
mi_familia.familia, mi_familia.oficina );
          COMMIT;
          RETURN todo bien;
     EXCEPTION
          WHEN referencia no encontrada THEN
               ROLLBACK;
               RETURN padre_inexistente;
          WHEN no null THEN
               ROLLBACK;
               RETURN no null violado;
           WHEN DUP VAL ON INDEX THEN
               ROLLBACK;
               RETURN elemento_existente;
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END inserta_familia;

     --Inserta una oficina
     FUNCTION inserta_oficina ( mi_oficina oficina )
     RETURN NUMBER IS
     BEGIN
          INSERT   INTO   oficinas   VALUES   (mi_oficina.identificador,   mi_oficina.nombre,
mi_oficina.domicilio, mi_oficina.localidad, mi_oficina.codigo_postal );
          COMMIT;
          RETURN todo_bien;
     EXCEPTION
          WHEN no null THEN
               ROLLBACK;
               RETURN no_null_violado;
          WHEN DUP VAL ON INDEX THEN
               ROLLBACK;
               RETURN elemento existente;
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END inserta oficina;

     --Borramos una oficina
     FUNCTION borra_oficina( id_oficina NUMBER )
     RETURN NUMBER IS
          num_ofi NUMBER;
     BEGIN
          SELECT COUNT(*) INTO num ofi FROM oficinas
          WHERE identificador = id oficina;
          IF (num_ofi = 0) THEN
               RETURN elemento_inexistente;
          END IF;
          DELETE oficinas WHERE identificador = id_oficina;
          COMMIT;
          RETURN todo_bien;
     EXCEPTION
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END borra oficina;

     --Borramos una familia
     FUNCTION borra familia( id familia NUMBER )
     RETURN NUMBER IS
          num fam NUMBER;
     BEGIN
          SELECT COUNT(*) INTO num fam FROM familias
          WHERE identificador = id_familia;
          IF (num_fam = 0) THEN
               RETURN elemento_inexistente;
```

```
          END IF;
          DELETE familias WHERE identificador = id_familia;
          COMMIT;
          RETURN todo bien;
     EXCEPTION
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END borra_familia;

     --Borramos un agente
     FUNCTION borra_agente( id_agente NUMBER )
     RETURN NUMBER IS
          num ag NUMBER;
     BEGIN
          SELECT COUNT(*) INTO num ag FROM agentes
          WHERE identificador = id_agente;
          IF (num_ag = 0) THEN
               RETURN elemento inexistente;
          END IF;
          DELETE agentes WHERE identificador = id agente;
          COMMIT;
          RETURN todo_bien;
     EXCEPTION
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END borra_agente;
END call_center;
/
```

Para referenciar las partes visibles de un paquete, lo haremos por medio de la notación del punto.

```
BEGIN
     ...
     call_center.borra_agente( 10 );
     ...
END;
```

Oracle nos suministra varios paquetes para simplificarnos algunas tareas. En el siguiente enlace puedes encontrar más información sobre los mismos.
http://docs.oracle.com/cd/B19306_01/appdev.102/b14258/toc.htm

4.2.1.- Ejemplos de utilización del paquete DBMS_OUTPUT.

Oracle nos suministra un paquete público con el cual podemos enviar mensajes desde subprogramas almacenados, paquetes y disparadores, colocarlos en un buffer y leerlos desde otros subprogramas almacenados, paquetes o disparadores.

SQL*Plus permite visualizar los mensajes que hay en el buffer, por medio del comando `SET SERVEROUTPUT ON`. La utilización fundamental de este paquete es para la depuración de nuestros subprogramas.
Veamos uno a uno los subprogramas que nos suministra este paquete:

- ✓ Habilita las llamadas a los demás subprogramas. No es necesario cuando está activada la opción `SERVEROUTPUT`. Podemos pasarle un parámetro indicando el tamaño del buffer.

```
ENABLE
ENABLE( buffer_size IN INTEGER DEFAULT 2000);
```

- ✓ Deshabilita las llamadas a los demás subprogramas y purga el buffer. Como con `ENABLE` no es necesario si estamos usando la opción `SERVEROUTPUT`.

```
DISABLE
DISABLE();
```

- ✓ Coloca elementos en el buffer, los cuales son convertidos a `VARCHAR2`.

```
PUT
PUT(item IN NUMBER);
PUT(item IN VARCHAR2);
PUT(item IN DATE);
```

- ✓ Coloca elementos en el buffer y los termina con un salto de línea.

```
PUT_LINE
PUT_LINE(item IN NUMBER);
PUT_LINE(item IN VARCHAR2);
PUT_LINE(item IN DATE);
```

- ✓ Coloca un salto de línea en el buffer. Utilizado cuando componemos una línea usando varios `PUT`.

```
NEW_LINE
NEW_LINE();
```

- ✓ Lee una línea del buffer colocándola en el parámetro line y obviando el salto de línea. El parámetro status devolverá 0 si nos hemos traído alguna línea y 1 en caso contrario.

```
GET_LINE
GET_LINE(line OUT VARCHAR2, status OUT VARCHAR2);
```

- ✓ Intenta leer el número de líneas indicado en numlines. Una vez ejecutado, numlines contendrá el número de líneas que se ha traído. Las líneas traídas las coloca en el parámetro lines del tipo `CHARARR`, tipo definido el paquete DBMS_OUTPUT como una tabla de `VARCHAR2(255)`.

```
GET_LINES
GET_LINES(lines OUT CHARARR, numlines IN OUT INTEGER);
```

Ejercicio resuelto

Debes crear un procedimiento que visualice todos los agentes, su nombre, nombre de la familia y/o nombre de la oficina a la que pertenece.

Respuesta:

```
CREATE OR  REPLACE PROCEDURE lista agentes IS
     CURSOR cAgentes IS SELECT identificador, nombre,familia, oficina FROM agentes;
     fam familias.nombre%TYPE;
     ofi oficinas.nombre%TYPE;
     num ag INTEGER := 0;
BEGIN
     DBMS_OUTPUT.ENABLE( 1000000 );
     DBMS_OUTPUT.PUT_LINE('Agente              |Familia             |Oficina              ');
     DBMS_OUTPUT.PUT_LINE('-----------------------------------------------------------------------');
     FOR ag_rec IN cAgentes LOOP
          IF (ag rec.familia IS NOT NULL) THEN
               SELECT nombre INTO fam FROM familias WHERE identificador = ag rec.familia;
               ofi := NULL;
               DBMS_OUTPUT.PUT_LINE(rpad(ag_rec.nombre,20)  ||  '|'  ||  rpad(fam,20)  ||  '|'  ||
rpad(ofi,20));
               num_ag := num_ag + 1;
          ELSIF (ag_rec.oficina IS NOT NULL) THEN
               SELECT nombre INTO ofi FROM oficinas WHERE identificador = ag rec.oficina;
               fam := NULL;
               DBMS_OUTPUT.PUT_LINE(rpad(ag_rec.nombre,20)  ||  '|'  ||  rpad(fam,20)  ||  '|'  ||
rpad(ofi,20));
               num_ag := num_ag + 1;
          END IF;
     END LOOP;
DBMS_OUTPUT.PUT_LINE('-----------------------------------------------------------------------');
     DBMS_OUTPUT.PUT_LINE('Número de agentes: ' || num_ag);
END lista_agentes;
/
```

Recuerda que para ejecutarlo desde SQL*Plus debes ejecutar las siguientes sentencias:

```
SQL>SET SERVEROUTPUT ON;
SQL>EXEC lista_agentes;
```

4.3.- Objetos.

Hoy día, la programación orientada a objetos es uno de los paradigmas más utilizados y casi todos los lenguajes de programación la soportan. En este apartado vamos a dar unas pequeñas pinceladas de su uso en PL/SQL que serán ampliados en la siguiente unidad de trabajo.

Un tipo de objeto es un tipo de dato compuesto, que encapsula unos datos y las funciones y procedimientos necesarios para manipular esos datos. Las variables son los atributos y los subprogramas son llamados métodos. Podemos pensar en un tipo de objeto como en una entidad que posee unos atributos y un comportamiento (que viene dado por los métodos).

- ✓ Cuando creamos un tipo de objeto, lo que estamos creando es una entidad abstracta que especifica los atributos que tendrán los objetos de ese tipo y define su comportamiento.
- ✓ Cuando instanciamos un objeto estamos particularizando la entidad abstracta a una en particular, con los atributos que tiene el tipo de objeto, pero con un valor dado y con el mismo comportamiento.

Los tipos de objetos tiene 2 partes: una **especificación** y un **cuerpo**. La parte de especificación declara los atributos y los métodos que harán de interfaz de nuestro tipo de objeto. En el cuerpo se implementa la parte de especificación. En la parte de especificación debemos declarar primero los atributos y después los métodos. Todos los atributos son públicos (visibles). No podemos declarar atributos en el cuerpo, pero sí podemos declarar subprogramas locales que serán visibles en el cuerpo del objeto y que nos ayudarán a implementar nuestros métodos.

Los atributos pueden ser de cualquier tipo de datos Oracle, excepto:

- ✓ `LONG` y `LONG RAW`.
- ✓ `NCHAR`, `NCLOB` y `NVARCHAR2`.
- ✓ `MLSLABEL` y `ROWID`.
- ✓ Tipos específicos de PL/SQL: `BINARY INTEGER`, `BOOLEAN`, `PLS INTEGER`, `RECORD`, `REF CURSOR`, `%TYPE` y `%ROWTYPE`.
- ✓ Tipos definidos dentro de un paquete PL/SQL.

No podemos inicializar un atributo en la declaración. Tampoco podemos imponerle la restricción `NOT NULL`.

Un **método** es un subprograma declarado en la parte de especificación de un tipo de objeto por medio de: `MEMBER`. Un método no puede llamarse igual que el tipo de objeto o que cualquier atributo. Para cada método en la parte de especificación, debe haber un método implementado en el cuerpo con la misma cabecera.

Todos los métodos en un tipo de objeto aceptan como primer parámetro una instancia de su tipo. Este parámetro es `SELF` y siempre está accesible a un método. Si lo declaramos explícitamente debe ser el primer parámetro, con el nombre `SELF` y del tipo del tipo de objeto. Si `SELF` no está declarado explícitamente, por defecto será `IN` para las funciones e `IN OUT` para los procedimientos.

Los métodos dentro de un tipo de objeto pueden sobrecargarse. No podemos sobrecargarlos si los parámetros formales sólo difieren en el modo o pertenecen a la misma familia. Tampoco podremos sobrecargar una función miembro si sólo difiere en el tipo devuelto.

Una vez que tenemos creado el objeto, podemos usarlo en cualquier declaración. Un objeto cuando se declara sigue las mismas reglas de alcance y visibilidad que cualquier otra variable.

Cuando un objeto se declara éste es automáticamente `NULL`. Dejará de ser nulo cuando lo inicialicemos por medio de su constructor o cuando le asignemos otro. Si intentamos acceder a los atributos de un objeto `NULL` saltará la excepción `ACCES_INTO_NULL`.

Todos los objetos tienen constructores por defecto con el mismo nombre que el tipo de objeto y acepta tantos parámetros como atributos del tipo de objeto y con el mismo tipo. PL/SQL no llama implícitamente a los constructores, deberemos hacerlo nosotros explícitamente.

```
DECLARE
     familia1 Familia;
BEGIN
     ...
     familia1 := Familia( 10, 'Fam10', 1, NULL );
     ...
END;
```

Un tipo de objeto puede tener a otro tipo de objeto entre sus atributos. El tipo de objeto que hace de atributo debe ser conocido por Oracle. Si 2 tipos de objetos son mutuamente dependientes, podemos usar una declaración hacia delante para evitar errores de compilación.

Ejercicio resuelto

Cómo declararías los objetos para nuestra base de datos de ejemplo.

Respuesta:

```
CREATE OBJECT Oficina;     --Definición hacia delante

CREATE OBJECT Familia AS OBJECT (
     identificador     NUMBER,
     nombre          VARCHAR2(20),
     familia      Familia,
     oficina      Oficina,
     ...
);

CREATE OBJECT Agente AS OBJECT (
     identificador     NUMBER,
     nombre          VARCHAR2(20),
     familia_     Familia,
     oficina_     Oficina,
     ...
);

CREATE OBJECT Oficina AS OBJECT (
     identificador     NUMBER,
     nombre          VARCHAR2(20),
     jefe          Agente,
     ...
);
```

4.3.1.- Objetos. Funciones mapa y funciones de orden.

En la mayoría de los problemas es necesario hacer comparaciones entre tipos de datos, ordenarlos, etc. Sin embargo, los tipos de objetos no tienen orden predefinido por lo que no podrán ser comparados ni ordenados (`x>y`,`DISTINCT`, `ORDER BY`, ...). Nosotros podemos definir el orden que seguirá un tipo de objeto por medio de las funciones mapa y las funciones de orden.

Una función miembro mapa es una función sin parámetros que devuelve un tipo de dato: `DATE`, `NUMBER` o `VARCHAR2` y sería similar a una función hash. Se definen anteponiendo la palabra clave `MAP` y sólo puede haber una para el tipo de objeto.

```
CREATE TYPE Familia AS OBJECT (
identificador      NUMBER,
nombre           VARCHAR2(20),
familia_          NUMBER,
oficina_          NUMBER,
MAP MEMBER FUNCTION orden RETURN NUMBER,
...
);

CREATE TYPE BODY Familia AS
MAP MEMBER FUNCTION orden RETURN NUMBER IS
BEGIN
     RETURN identificador;
END;
...
```

```
END;
```

Una función miembro de orden es una función que acepta un parámetro del mismo tipo del tipo de objeto y que devuelve un número negativo si el objeto pasado es menor, cero si son iguales y un número positivo si el objeto pasado es mayor.

```
CREATE TYPE Oficina AS OBJECT (
     identificador          NUMBER,
     nombre              VARCHAR2(20),
     ...
     ORDER MEMBER FUNCTION igual ( ofi Oficina ) RETURN INTEGER,
     ...
);

CREATE TYPE BODY Oficina AS
     ORDER MEMBER FUNCTION igual ( ofi Oficina ) RETURN INTEGER IS
     BEGIN
          IF (identificador < ofi.identificador) THEN
               RETURN -1;
          ELSIF (identificador = ofi.identificador) THEN
               RETURN 0;
          ELSE
               RETURN 1;
          END IF;
     END;
     ...
END;
```

Los métodos de un objeto sólo pueden ser procedimientos.

- ○ Verdadero
- ◉ **Falso**

Efectivamente, ya que las funciones también pueden ser métodos.

El orden de un objeto se consigue:

- ○ Al crearlo.
- ○ En PL/SQL los objetos no pueden ser ordenados.
- ◉ **Mediante las funciones mapa y las funciones de orden.**

5.- Disparadores.

Caso práctico

__Juan__ y __María__ ya han hecho un paquete en el que tienen agrupadas las funciones más usuales que realizan sobre la base de datos de juegos on-line. Sin embargo, __María__ ve que hay cosas que aún no pueden controlar. Por ejemplo,__María__ quiere que la clave y el usuario de un jugador o jugadora no puedan ser la misma y eso no sabe cómo hacerlo. __Juan__ le dice que para ese cometido están los disparadores y sin más dilaciones se pone a explicarle a __María__ para qué sirven y cómo utilizarlos.

En este apartado vamos a tratar una herramienta muy potente proporcionada por PL/SQL para programar nuestra base de datos que son los disparadores o triggers en inglés.

Pero, ¿qué es un disparador?

Un disparador no es más que un procedimiento que es ejecutado cuando se realiza alguna sentencia de manipulación de datos sobre una tabla dada y bajo unas circunstancias establecidas a la hora de definirlo.

Por lo que un disparador puede ser usado para:
- ✓ Llevar a cabo auditorías sobre la historia de los datos en nuestra base de datos.
- ✓ Garantizar complejas reglas de integridad.
- ✓ Automatizar la generación de valores derivados de columnas.
- ✓ Etc.

Cuando diseñamos un disparador debemos tener en cuenta que:
- ✓ No debemos definir disparadores que dupliquen la funcionalidad que ya incorpora Oracle.
- ✓ Debemos limitar el tamaño de nuestros disparadores, y si estos son muy grandes codificarlos por medio de subprogramas que sean llamados desde el disparador.
- ✓ Cuidar la creación de disparadores recursivos.

Un disparador puede ser lanzado antes o después de realizar la operación que lo lanza. Por lo que tendremos disparadores `BEFORE` y disparadores `AFTER`.

Un disparador puede ser lanzado una vez por sentencia o una vez por cada fila a la que afecta. Por lo que tendremos disparadores de sentencia y disparadores de fila.

Un disparador puede ser lanzado al insertar, al actualizar o al borrar de una tabla, por lo que tendremos disparadores `INSERT`, `UPDATE` o `DELETE` (o mezclados).

En PL/SQL sólo podemos definir disparadores de fila.

○ Verdadero ◉ **Falso**

Efectivamente, ya que también podemos definir disparador de sentencia.

La diferente entre un disparador de fila y uno de sentencia es que el de fila es lanzado una vez por fila a la que afecta la sentencia y el de sentencia es lanzado una sola vez.

◉ **Verdadero** ○ Falso

5.1.- Definición de disparadores.

Por lo visto anteriormente, para definir un disparador deberemos indicar si será lanzado antes o después de la ejecución de la sentencia que lo lanza, si se lanzará una vez por sentencia o una vez por fila a la que afecta, y si será lanzado al insertar y/o al actualizar y/o al borrar. La sintaxis que seguiremos para definir un disparador será la siguiente:

```
CREATE [OR REPLACE] TRIGGER nombre
momento acontecimiento ON tabla
[[REFERENCING (old AS alias old|new AS alias new)
FOR EACH ROW
[WHEN condicion]]
bloque_PL/SQL;
```

Donde nombre nos indica el nombre que le damos al disparador, momento nos dice cuando será lanzado el disparador (`BEFORE` o `AFTER`), acontecimiento será la acción que provoca el lanzamiento del disparador (`INSERT` y/`oDELETE` y/o `UPDATE`). `REFERENCING` y `WHEN` sólo podrán ser utilizados con disparadores para filas. `REFERENCING` nos permite asignar un alias a los valores `NEW` o/y `OLD` de las filas afectadas por la operación, y `WHEN` nos permite indicar al disparador que sólo sea lanzado cuando sea `TRUE` una cierta condición evaluada para cada fila afectada.

En un disparador de fila, podemos acceder a los valores antiguos y nuevos de la fila afectada por la operación, referenciados como `:old` y `:new` (de ahí que podamos utilizar la opción `REFERENCING` para asignar un alias). Si el disparador es lanzado al insertar, el valor antiguo no tendrá sentido y el valor nuevo será la fila que estamos insertando. Para un disparador lanzado al actualizar el valor antiguo contendrá la fila antes de actualizar y el valor nuevo contendrá la fila que vamos actualizar. Para un disparador lanzado al borrar sólo tendrá sentido el valor antiguo.

En el cuerpo de un disparador también podemos acceder a unos predicados que nos dicen qué tipo de operación se está llevando a cabo, que son: `INSERTING`, `UPDATING` y `DELETING`.

Un disparador de fila no puede acceder a la tabla asociada. Se dice que esa tabla está mutando. Si un disparador es lanzado en cascada por otro disparador, éste no podrá acceder a ninguna de las tablas asociadas, y así recursivamente.

```
CREATE TRIGGER prueba BEFORE UPDATE ON agentes
FOR EACH ROW
BEGIN
     ...
     SELECT identificador FROM agentes WHERE ...
/*devolvería el error ORA-04091: table AGENTES is mutating, trigger/function may not see it*/
     ...
END;
/
```

Si tenemos varios tipos de disparadores sobre una misma tabla, el orden de ejecución será:

- ✓ Triggers before de sentencia.
- ✓ Triggers before de fila.
- ✓ Triggers after de fila.
- ✓ Triggers after de sentencia.

Existe una vista del diccionario de datos con información sobre los disparadores: `USER_TRIGGERS;`

```
SQL>DESC USER_TRIGGERS;
Name                           Null?    Type
------------------------------ -------- ----
TRIGGER_NAME                   NOT NULL VARCHAR2(30)
TRIGGER_TYPE                            VARCHAR2(16)
TRIGGERING_EVENT                        VARCHAR2(26)
TABLE_OWNER                    NOT NULL VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
REFERENCING_NAMES                       VARCHAR2(87)
WHEN_CLAUSE                             VARCHAR2(4000)
STATUS                                  VARCHAR2(8)
DESCRIPTION                             VARCHAR2(4000)
TRIGGER_BODY                            LONG
```

5.2.- Ejemplos de disparadores.

Ya hemos visto qué son los disparadores, los tipos que existen, cómo definirlos y algunas consideraciones a tener en cuenta a la hora de trabajar con ellos.

Ahora vamos a ver algunos ejemplos de su utilización con los que podremos comprobar la potencia que éstos nos ofrecen.

Ejercicio resuelto

Como un agente debe pertenecer a una familia o una oficina pero no puede pertenecer a una familia y a una oficina a la vez, deberemos implementar un disparador para llevar a cabo esta restricción que Oracle no nos permite definir.

Respuesta:

Para este cometido definiremos un disparador de fila que saltará antes de que insertemos o actualicemos una fila en la tabla agentes, cuyo código podría ser el siguiente:

```
CREATE OR REPLACE TRIGGER integridad_agentes
BEFORE INSERT OR UPDATE ON agentes
FOR EACH ROW
BEGIN
     IF (:new.familia IS NULL and :new.oficina IS NULL) THEN
          RAISE_APPLICATION_ERROR(-20201, 'Un agente no puede ser huérfano');
     ELSIF (:new.familia IS NOT NULL and :new.oficina IS NOT NULL) THEN
          RAISE_APPLICATION_ERROR(-20202, 'Un agente no puede tener dos padres');
     END IF;
END;
/
```

Ejercicio resuelto

Supongamos que tenemos una tabla de históricos para agentes que nos permita auditar las familias y oficinas por la que ha ido pasando un agente. La tabla tiene la fecha de inicio y la fecha de finalización del agente en esa familia u oficina, el identificador del agente, el nombre del agente, el nombre de la familia y el nombre de la oficina. Queremos hacer un disparador que inserte en esa tabla.

Respuesta:

Para llevar a cabo esta tarea definiremos un disparador de fila que saltará después de insertar, actualizar o borrar una fila en la tabla agentes, cuyo código podría ser el siguiente:

```
CREATE OR REPLACE TRIGGER historico_agentes
AFTER INSERT OR UPDATE OR DELETE ON agentes
FOR EACH ROW
DECLARE
     oficina VARCHAR2(40);
     familia VARCHAR2(40);
     ahora DATE := sysdate;
BEGIN
     IF INSERTING THEN
          IF (:new.familia IS NOT NULL) THEN
               SELECT nombre INTO familia FROM familias WHERE identificador = :new.familia;
               oficina := NULL;
          ELSIF
               SELECT nombre INTO oficina FROM oficinas WHERE identificador = :new.oficina;
               familia := NULL;
          END IF;
          INSERT INTO histagentes VALUES (ahora, NULL, :new.identificador, :new.nombre,
familia, oficina);
          COMMIT;
     ELSIF UPDATING THEN
          UPDATE histagentes SET fecha hasta = ahora WHERE identificador = :old.identificador
and fecha hasta IS NULL;
          IF (:new.familia IS NOT NULL) THEN
               SELECT nombre INTO familia FROM familias WHERE identificador = :new.familia;
               oficina := NULL;
          ELSE
```

```
               SELECT nombre INTO oficina FROM oficinas WHERE identificador = :new.oficina;
               familia := NULL;
          END IF;
          INSERT INTO histagentes VALUES (ahora, NULL, :new.identificador, :new.identificador,
familia, oficina);
          COMMIT;
     ELSE
          UPDATE histagentes SET fecha_hasta = ahora WHERE identificador = :old.identificador
and fecha_hasta IS NULL;
          COMMIT;
     END IF;
END;
/
```

Ejercicio resuelto

Queremos realizar un disparador que no nos permita llevar a cabo operaciones con familias si no estamos en la jornada laboral.

Respuesta:

```
CREATE OR REPLACE TRIGGER jornada_familias
BEFORE INSERT OR DELETE OR UPDATE ON familias
DECLARE
     ahora DATE := sysdate;
BEGIN
     IF (TO_CHAR(ahora, 'DY') = 'SAT' OR TO_CHAR(ahora, 'DY') = 'SUN') THEN
          RAISE_APPLICATION_ERROR(-20301, 'No podemos manipular familias en fines de semana');
     END IF;
     IF (TO CHAR(ahora, 'HH24') < 8 OR TO CHAR(ahora, 'HH24') > 18) THEN
          RAISE_APPLICATION_ERROR(-20302, 'No podemos manipular familias fuera del horario de
trabajo');
     END IF;
END;
/
```

6.- Interfaces de programación de aplicaciones para lenguajes externos.

Caso práctico

__Juan__ y __María__ tienen la base de datos de juegos on-line lista, pero no tienen claro si todo lo que han hecho lo tienen que utilizar desde dentro de la base de datos o si pueden reutilizar todo ese trabajo desde otro lenguaje de programación desde el que están creando la interfaz web desde la que los jugadores y jugadoras se conectarán para jugar y desde la que también se conectará la administradora para gestionar la base de datos de una forma más amigable.

En ese momento pasa __Ada__ al lado y ambos la asaltan con su gran duda. __Ada__, muy segura, les responde que para eso existen las interfaces de programación de aplicaciones o APIs, que les permiten acceder desde otros lenguajes de programación a la base de datos.

Llegados a este punto ya sabemos programar nuestra base de datos, pero al igual que a __Juan__ y __María__ te surgirá la duda de si todo lo que has hecho puedes aprovecharlo desde cualquier otro lenguaje de programación. La respuesta es la misma que dio __Ada__: SÍ.

En este apartado te vamos a dar algunas referencias sobre las APIs para acceder a las bases de datos desde un lenguaje de programación externo. No vamos a ser exhaustivos ya que eso queda fuera del alcance de esta unidad e incluso de este módulo. Todo ello lo vas a estudiar con mucha más profundidad en otros módulos de este ciclo (o incluso ya lo conoces). Aquí sólo pretendemos que sepas que existen y que las puedes utilizar.

Las primeras APIs utilizadas para acceder a bases de datos Oracle fueron las que el mismo Oracle proporcionaba: Pro*C, Pro*Fortran y Pro*Cobol. Todas permitían embeber llamadas a la base de datos en nuestro programa y a la hora de compilarlo, primero debíamos pasar el precompilador adecuado que trasladaba esas llamadas embebidas en llamadas a una librería utilizada en tiempo de ejecución. Sin duda, el más utilizado fue Pro*C, ya que el lenguaje C y C++ tuvieron un gran auge.

En los siguientes enlaces podrás obtener más información sobre Pro*C y cómo crear un programa para acceder a nuestra base de datos.
http://infolab.stanford.edu/~ullman/fcdb/oracle/or-proc.html
http://www.helloworldexample.net/pro-c-hello-world-example.html

Hoy día existen muchas más APIs para acceder a las bases de datos ya que tanto los lenguajes de programación como las tecnologías han evolucionado mucho. Antes la programación para la web casi no existía y por eso todos los programas que accedían a bases de datos lo hacían bien en local o bien a través de una red local. Hoy día eso sería impensable, de ahí que las APIs también hayan evolucionado mucho y tengamos una gran oferta a nuestro alcance para elegir la que más se adecue a nuestras necesidades.

En los siguientes enlaces podrás ampliar información sobre algunas APIs muy comunes para acceder a bases de datos.
http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html
http://en.wikipedia.org/wiki/ODBC
http://casidiablo.net/java-database-connectivity/

TEMA 7

INDICE

1.- Características de las bases de datos objeto-relacionales. 3
2.- Tipos de datos objeto. 5
3.- Definición de tipos de objeto. 6
3.1.- Declaración de atributos. 7
3.2.- Definición de métodos. 8
3.3.- Parámetro SELF. 9
3.4.- Sobrecarga. 9
3.5.- Métodos Constructores. 10
4.- Utilización de objetos. 11
4.1.- Declaración de objetos. 11
4.2.- Inicialización de objetos. 12
4.3.- Acceso a los atributos de objetos. 13
4.4.- Llamada a los métodos de los objetos. 14
4.5.- Herencia. 14
5.- Métodos MAP y ORDER. 16
5.1.- Métodos ORDER. 16
6.- Tipos de datos colección. 18
6.1.- Declaración y uso de colecciones. 19
7.- Tablas de objetos. 21
7.1.- Tablas con columnas tipo objeto. 22
7.2.- Uso de la sentencia Select. 22
7.3.- Inserción de objetos. 23
7.4.- Modificación de objetos. 23
7.5.- Borrado de objetos. 24
7.6.- Consultas con la función VALUE. 25
7.7.- Referencias a objetos. 26
7.8.- Navegación a través de referencias. 27
Anexo I - Soporte objeto-relacional en Oracle 8 28
1. Registros PL/SQL 28
1.1. Declaración de tipos y de variables registro 28
1.2. Acceso a un campo de una variable de tipo registro 28
1.3. Asignación de valores 28
1.4. Declaración de registros con el atributo %ROWTYPE 28
2. Tablas PL/SQL 29
2.1. Creación de una tabla 29
2.2. Referenciar un elemento de la tabla 29
3. Tablas PL/SQL de registros 29
4. Funciones para el manejo de tablas PL/SQL 29
5. Tablas anidadas (Nested tables) 30
5.1. Sintaxis 30
5.2. Inicialización de una tabla 30
5.3. Claves en la inicialización 31
5.4. Adición de elementos a una tabla existente 31
5.5. Tablas anidadas en la base de datos 32
5.6. Manipulación de tablas completas 32
Inserción 32
Modificación 32
Eliminación 32
Selección 33
6. VARRAYS 33
6.1. Declaración de un varray 33
6.2. Inicialización de un varray 33
6.3. Manipulación de los elementos de un varray 34
6.4. Varrays en la base de datos 34
6.5. Manipulación de varrays almacenados 34
7. VARRAYS y Tablas Anidadas 34
8. Métodos de colecciones 35
EXISTS 35
COUNT 35
LIMIT 35
FIRST y LAST 35
NEXT y PRIOR 35
DELETE 36
TRIM 36
9. Paquetes y resolución del problema de las tablas mutantes 36
Anexo II - Tipos de Objeto en PL/SQL 39
Introducción 39
Estructura de un tipo de objeto 39
Tablas de objetos 40
Métodos 40
Constructores 41
Métodos constructor 41
Declarar e inicializar objetos 41
Acceso a los atributos de un objeto 41
Variables de correlación 41
OID - Object IDentifier 42
Tipos referencia 43
Operador VALUE 44
Gestión de objetos 44
Forward Type Definitions 44
Tipos Colección 45
Creación 45
Consulta 45
Operaciones DML 45
Operadores 46
PL/SQL 46
Cursores y colecciones 46
Colecciones de tipo REF 46
Tipos colección 47
Tipos de Objeto en PL/SQL 47
Capture 47
Resolución de nombres: uso de alias 47
Anexo III - Bases de datos objeto-relacionales 49
Tecnología objeto-relacional 49
Tipos de objetos 49
Estructura de un tipo de objeto 49
Características: 50
Ejemplo: 50
Componentes de un tipo de objeto 50
Atributos 50
Métodos 51
El parámetro SELF 52

Sobrecarga 52
Métodos MAP y ORDER 53
Constructores 54
Pragma RESTRIC_REFERENCES 54
Declaración e inicialización de objetos 55
Declaración de objetos 55
Inicialización de objetos 56
Objetos sin inicializar en PL/SQL 56
Acceso a los atributos 57
Invocación de constructores y métodos 57
Paso de parámetros a un constructor 57
Invocación de métodos 58
Compartición de objetos 58
Utilización de referencias 59
Limitaciones en la definición de tipos 60
Manipulación de objetos 61
Selección de objetos 61
El operador VALUE 61
El operador REF 62
Referencias colgadas (dangling refs) 62
El operador DEREF 63
Inserción de objetos 64
Actualización de objetos 65
Borrado de objetos 65
Anexo IV - Bases de Datos Objeto-Relacionales en Oracle 8 66
1. Introducción 66
2. Tipos de Datos Definidos por el Usuario 66
2.1 Tipos de objetos 66
2.2 Métodos 67
2.2.1 Constructores de tipo 67
2.2.2 Métodos de comparación 68
2.3 Tablas de objetos 69
2.4 Referencias entre objetos 69
2.5 Tipos para colecciones 70
2.5.1 El tipo VARRAY 70
2.5.2 Tablas anidadas 71
3. Inserción y Acceso a los Datos 72
3.1 Alias 72
3.2 Inserción de referencias 73
3.3 Llamadas a métodos 73
3.4 Inserción en tablas anidadas 74
4. Una Base de Datos Ejemplo 74
4.1 Modelo lógico para una base de datos relacional 74
4.1.1 Implementación relacional con Oracle 8 75
4.2 Modelo lógico para una base de datos orientada a objetos 75
4.2.1 Implementación objeto-relacional con Oracle 8 76
4.2.2 Creación de tablas de objetos 76
4.2.3 Inserción de objetos en las tablas 76
4.2.4 Borrado de los objetos, las tablas y los tipos de usuario 78
4.2.5 Definición de métodos para los tipos 79
4.2.6 Consultas a la base de datos anterior 79

Uso de bases de datos objeto-relacionales.

Caso práctico

Juan *ha realizado un curso de perfeccionamiento sobre programación con bases de datos, y ha conocido las ventajas que pueden ofrecer el uso de las bases de datos objeto-relacionales. Hasta ahora siempre había usado las bases de datos relacionales, pero el conocimiento de las bases de datos orientadas a objetos le ha ofrecido nuevas perspectivas para aprovechar de manera más óptima la reutilización de código, el trabajo colaborativo, y la interconexión de las bases de datos con otros lenguajes de programación orientados a objetos.*

1.- Características de las bases de datos objeto-relacionales.

Caso práctico

__Ana__ ha sabido que __Juan__ ha realizado el curso de formación de bases de datos objeto-relacionales, y éste se ha ofrecido a compartir los conocimientos que ha adquirido. Lo primero va a ser es que conozca las características que tienen estos tipos de bases de datos, para que compruebe las diferencias que tienen respecto a las bases de datos relaciones que ha usado hasta ahora. Usarán la base de datos de Oracle que han utilizado anteriormente, pero ahora van a darle el enfoque de este nuevo tipo de bases de datos.

Las **bases de datos objeto-relacionales** que vas a conocer en esta unidad son las referidas a aquellas que han evolucionado desde el modelo relacional tradicional a un modelo híbrido que utiliza además la tecnología orientada a objetos. Las **clases, objetos,** y **herencia** son directamente soportados en los esquemas de la base de datos y el lenguaje de consulta y manipulación de datos. Además da soporte a una extensión del modelo de datos con la creación personalizada de **tipos de datos y métodos**.

La base de datos de Oracle implementa el modelo orientado a objetos como una **extensión del modelo relacional**, siguiendo soportando la funcionalidad estándar de las bases de datos relacionales.

El modelo objeto-relacional ofrece las **ventajas** de las técnicas orientadas a objetos en cuanto a mejorar la reutilización y el uso intuitivo de los objetos, a la vez que se mantiene la alta capacidad de concurrencia y el rendimiento de las bases de datos relacionales.

Los tipos de objetos que vas a conocer, así como las características orientadas a objeto, te van a proporcionar un mecanismo para **organizar los datos y acceder a ellos a alto nivel**. Por debajo de la capa de objetos, los datos seguirán estando almacenados en columnas y tablas, pero vas a poder trabajar con ellos de manera más parecida a las entidades que puedes encontrar en la vida real, dando así más significado a los datos. En vez de pensar en términos de columnas y tablas cuando realices consultas a la base de datos, simplemente deberás seleccionar entidades que habrás creado, por ejemplo clientes o pedidos.

Podrás utilizar las características orientadas a objetos que vas a aprender en esta unidad a la vez que puedes seguir trabajando con los datos relacionalmente, o bien puedes usar de manera más exclusiva las técnicas orientadas a objetos.

En general, el modelo orientado a objetos es **similar al que puedes encontrar en lenguajes** como C++ y Java. La **reutilización** de objetos permite desarrollar aplicaciones de bases de datos más rápidamente y de manera más eficiente. Al ofrecer la base de datos de Oracle soporte nativo para los tipos de objetos, permite a los desarrolladores de aplicaciones con lenguajes orientados a objetos, acceder directamente a las **mismas estructuras de datos**creadas en la base de datos.

La programación orientada a objetos está especialmente enfocada a la construcción de componentes reutilizables y aplicaciones complejas. En **PL/SQL**, la programación orientada a objetos está basada en **tipos de objetos**. Estos tipos te permiten modelar objetos de la vida real, separar los detalles de los interfaces de usuarios de la implementación, y almacenar los datos orientados a objetos de forma permanente en una base de datos. Encontraras especialmente útil los tipos de objetos cuando realizas programas interconectados con Java u otros lenguajes de programación orientados a objetos.

Las tablas de bases de datos relacionales sólo contienen datos. En cambio, los objetos pueden incluir la posibilidad de realizar determinadas **acciones sobre los datos**. Por ejemplo, un objeto "Compra" puede incluir un método para calcular el importe de todos los elementos comprados. O un objeto

"Cliente" puede tener métodos que permitan obtener su historial de compras. De esta manera, una aplicación tan sólo debe realizar una llamada a dichos métodos para obtener esa información.

Si deseas conocer más en detalle la definición de las bases de datos objeto-relacionales puedes visitar la web de wikipedia:
http://es.wikipedia.org/wiki/Base_de_datos_objeto-relacional

2.- Tipos de datos objeto.

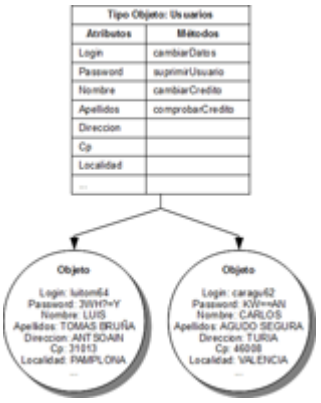
Caso práctico

__Juan__ comienza a explicarle a __Ana__ uno de los conceptos básicos en las bases de datos orientadas a objetos. Se trata de los tipos de datos objeto. Le hace ver que cualquier objeto que manejamos a diario se puede considerar un posible tipo de objeto para una base de datos.

Un **tipo de dato objeto** es un tipo de dato compuesto definido por el usuario. Representa una **estructura de datos** así como **funciones y procedimientos** para manipular datos. Como ya sabemos, las variables de un determinado tipo de dato escalar (`NUMBER`, `VARCHAR`, `BOOLEAN`, `DATE`, etc.) pueden almacenar un único valor. Las coleccionas permiten, como ya veremos, almacenar varios datos en una misma variable siendo todos del mismo tipo. Los tipos de datos objetos nos permitirán almacenar datos de distintos tipos, posibilitando además asociar código a dichos datos.

En el mundo real solemos pensar en un objeto, entidad, persona, animal, etc. como un conjunto de propiedades y acciones. Por ejemplo, el alumnado tiene una serie de propiedades como el nombre, fecha de nacimiento, DNI, curso, grupo, calificaciones, nombre del padre y de la madre, sexo, etc., y tiene asociadas una serie de acciones como matricular, evaluar, asistir a clase, presentar tareas, etc. Los tipos de datos objeto nos permiten representar esos valores y estos comportamientos de la vida real en una aplicación informática.

Piensa en un objeto y enumera algunas de sus propiedades y acciones que se pueden realizar sobre él.



Las variables que formen la estructura de datos de un tipo de dato objeto reciben el nombre de **atributos** (que se corresponde con sus propiedades). Las funciones y procedimientos del tipo de dato objeto se denominan **métodos**(que se corresponde con sus acciones).

Cuando se define un tipo de objeto, se crea una **plantilla abstracta** de un objeto de la vida real. La plantilla especifica los atributos y comportamientos que el objeto necesita en el entorno de la aplicación. Dependiendo de la aplicación a desarrollar se utilizarán sólo determinados atributos y comportamiento del objeto. Por ejemplo, en la gestión de la evaluación del alumnado es muy probable que no se necesite conocer su altura, peso, etc. o utilizar comportamientos como desplazarse, comer, etc., aunque formen todos ellos parte de las características del alumnado.

Aunque los atributos son públicos, es decir, visibles desde otros programas cliente, los programas deberían **manipular los datos únicamente a través de los métodos** (funciones y procedimientos) que se hayan declarado en el tipo objeto, en vez de asignar u obtener sus valores directamente. Esto es debido a que los métodos pueden hacer un chequeo de los datos de manera que se mantenga un estado apropiado en los mismos. Por ejemplo, si se desea asignar un curso a un miembro del alumnado, sólo debe permitirse que se asigne un curso existente. Si se permitiera modificar directamente el curso, se podría asignar un valor incorrecto (curso inexistente).

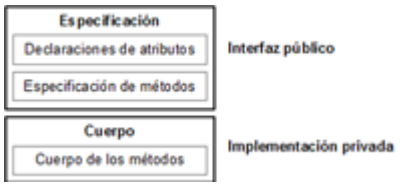
Durante la ejecución, la aplicación creará **instancias** de un tipo objeto, es decir, referencias a objetos reales con valores asignados en sus atributos. Por ejemplo, una instancia será un determinado miembro del alumnado con sus datos personales correspondientes.

3.- Definición de tipos de objeto.

Caso práctico

*Ya conoce **Ana** el concepto de las bases de datos orientadas a objetos y que los tipos de datos objeto son la base para ellas. Así que para asentar esos conocimientos se plantea modificar, con la ayuda de **Juan**, la base de datos de la plataforma de juegos on-line que hasta ahora era relacional para convertirla en una base de datos orientada a objetos.*

Para ello, se plantea crear el tipo de objeto Usuario, pero necesita conocer cómo se declaran los tipos de datos objeto en Oracle.



La estructura de la definición o declaración de un tipo de objeto está dividida en una especificación y un cuerpo. La **especificación** define el interfaz de programación, donde se declaran los atributos así como las operaciones (métodos) para manipular los datos. En el **cuerpo** se implementa el código fuente de los métodos.

Toda la información que un programa necesita para usar los métodos lo encuentra en la especificación. Se puede modificar el cuerpo sin cambiar la especificación, sin que ello afecte a los programas cliente.

En la especificación de un tipo de objeto, todos los **atributos debes declararlos antes que los métodos**. Si la especificación de un tipo de objeto sólo declara atributos, no es necesario que declares el cuerpo. Debes tener en cuenta también que no puedes declarar atributos en el cuerpo. Además, todas las declaraciones realizadas en la especificación del tipo de objeto son públicas, es decir, visibles fuera del tipo de objeto.

Por tanto, un tipo de objeto contiene (encapsula) datos y operaciones. Puedes declarar atributos y métodos en la especificación, pero no constantes (`CONSTANTS`), excepciones (`EXCEPTIONS`), cursores (`CURSORS`) o tipos (`TYPES`). Al menos debe tener un atributo declarado, y un máximo de 1000. En cambio los métodos son opcionales, por lo que se puede crear un tipo de objeto sin métodos.
Para definir un objeto en Oracle debes utilizar la sentencia `CREATE TYPE` que tiene el siguiente formato:

```
CREATE TYPE nombre_tipo AS OBJECT (
Declaración_atributos
Declaración_métodos
);
```

Siendo nombre_tipo el nombre deseado para el nuevo tipo de objeto. La forma de declarar los atributos y los métodos se verá en los siguientes apartados de esta unidad didáctica.

En caso de que el **nombre del tipo de objeto ya estuviera siendo usado** para otro tipo de objeto se obtendría un error. Si se desea reemplazar el tipo anteriormente creado, por el nuevo que se va a declarar, se puede añadir la cláusula `OR REPLACE` en la declaración del tipo de objeto:

```
CREATE OR REPLACE TYPE nombre_tipo AS OBJECT
```

Por ejemplo, para crear el tipo de objeto Usuario, **reemplazando la declaración** que tuviera anteriormente se podría hacer algo similar a lo siguiente:

```
CREATE OR REPLACE TYPE Usuario AS OBJECT (
Declaración_atributos
Declaración_métodos
);
```

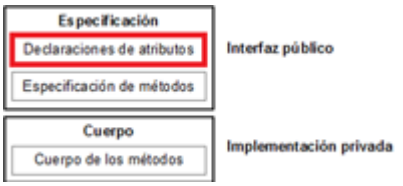
Si en algún momento deseas **eliminar el tipo de objeto** que has creado puedes utilizar la sentencia `DROP TYPE`:

```
DROP TYPE nombre_tipo;
```

Donde nombre_tipo debe ser el nombre del tipo de dato objeto que deseas eliminar. Por ejemplo, para el tipo de objetos anterior, deberías indicar:

```
DROP TYPE Usuario;
```

3.1.- Declaración de atributos.



La declaración de los atributos la puedes realizar de forma muy similar a declaración de las variables, es decir, utilizando un nombre y un tipo de dato. Dicho nombre debe ser único dentro del tipo de objeto, aunque puede ser reutilizado en otros tipos de objeto. El tipo de dato que puede almacenar un determinado atributo puede ser **cualquiera de los tipos de Oracle excepto** los siguientes:

- ✓ `LONG` y `LONG RAW`.
- ✓ `ROWID` y `UROWID`.
- ✓ Los tipos específicos PL/SQL `BINARY INTEGER` (y sus subtipos), `BOOLEAN`, `PLS INTEGER`, `RECORD`, `REF CURSOR`, `%TYPE`, y `%ROWTYPE`.
- ✓ Los tipos definidos dentro de un paquete PL/SQL.

Debes tener en cuenta que no puedes inicializar los atributos usando el operador de asignación, ni la cláusula `DEFAULT`, ni asignar la restricción `NOT NULL`.

El **tipo de dato de un atributo puede ser otro tipo de objeto**, por lo que la estructura de datos puede ser tan complicada como sea necesario.

```
CREATE OR REPLACE TYPE Usuario AS OBJECT (
   login VARCHAR2(10),
   nombre VARCHAR2(30),
   f_ingreso DATE,
   credito NUMBER
);
/
```

Después de haber sido creado el tipo de objeto, se pueden **modificar sus atributos** utilizando la sentencia `ALTER TYPE`. Si se desean añadir nuevos atributos se añadirá la cláusula `ADD ATTRIBUTE` seguida de la lista de nuevos atributos con sus correspondientes tipos de dato. Utilizando `MODIFY ATTRIBUTE` se podrán modificar los atributos existentes, y para eliminar atributos se dispone de manera similar de `DROP ATTRIBUTE`.

Aquí tienes varios ejemplos de modificación del tipo de objeto Usuario creado anteriormente:

```
ALTER TYPE Usuario DROP ATTRIBUTE f_ingreso;
```

```
ALTER TYPE Usuario ADD ATTRIBUTE (apellidos VARCHAR2(40), localidad VARCHAR2(50));
```

```
ALTER TYPE Usuario
   ADD ATTRIBUTE cp VARCHAR2(5),
   MODIFY ATTRIBUTE nombre VARCHAR2(35);
```

En la web de Oracle puedes encontrar la sintaxis completa de la instrucción CREATE TYPE. Ahí podrás conocer que hay más posibilidades de uso:
http://download.oracle.com/docs/cd/B13789_01/server.101/b10759/statements_8001.htm

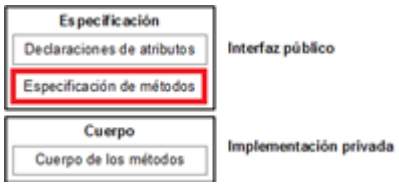
En la web de Oracle puedes encontrar la sintaxis completa de la instrucción ALTER TYPE. Ahí podrás conocer que hay más posibilidades de uso:
http://docs.oracle.com/cd/B14117_01/server.101/b10759/statements_4002.htm

3.2.- Definición de métodos.

Un **método es un subprograma** que declaras en la especificación de un tipo de objeto usando las palabras clave `MEMBER` o `STATIC`. Debes tener en cuenta que el nombre de un determinado método no puede puede ser el mismo nombre que el tipo de objeto ni el de ninguno de sus atributos. Como se verá más adelante, se pueden crear métodos con el mismo nombre que el tipo de objeto, pero dichos métodos tendrán una función especial.

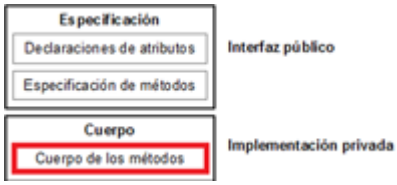
Al igual que los subprogramas empaquetados, los métodos tienen **dos partes: una especificación y un cuerpo**.

En la **especificación** o declaración se debe encontrar el nombre del método, una lista opcional de parámetros, y, en el caso de las funciones, un tipo de dato de retorno. Por ejemplo, observa la especificación del método incrementoCredito que se encuentra detrás de las declaraciones de los atributos:




```
CREATE OR REPLACE TYPE Usuario AS OBJECT (
   login VARCHAR2(10),
   nombre VARCHAR2(30),
   f_ingreso DATE,
   credito NUMBER,
   MEMBER PROCEDURE incrementoCredito(inc NUMBER)
);
/
```

En el **cuerpo** debes indicar el código que se debe ejecutar para realizar una determinada tarea cuando el método es invocado. En el siguiente ejemplo se desarrolla el cuerpo del método que se ha declarado antes:




```
CREATE OR REPLACE TYPE BODY Usuario AS
   MEMBER PROCEDURE incrementoCredito(inc NUMBER) IS
      BEGIN
         credito := credito + inc;
      END incrementoCredito;
END;
/
```

Por cada especificación de método que se indique en el bloque de especificación del tipo de objeto, debe existir su correspondiente cuerpo del método, o bien, el método debe declararse como `NOT INSTANTIABLE`, para indicar que el cuerpo del método se encontrará en un subtipo de ese tipo de objeto. Además, debes tener en cuenta que las cabeceras de los métodos deben coincidir exactamente en la especificación y en el cuerpo.

Al igual que los atributos, los parámetros formales se declaran con un nombre y un tipo de dato. Sin embargo, el tipo de dato de un parámetro no puede tener restricciones de tamaño. El tipo de datos puede ser cualquiera de los empleados por Oracle salvo los indicados anteriormente para los atributos. Las mismas restricciones se aplican para los tipos de retorno de las funciones.

El código fuente de los métodos no sólo puede escribirse en el lenguaje PL/SQL. También con otros lenguajes de programación como Java o C.

Puedes usar la sentencia `ALTER TYPE` para **añadir, modificar o eliminar métodos** de un tipo de objeto existente, de manera similar a la utilizada para modificar los atributos de un tipo de objeto.

3.3.- Parámetro SELF.

Un parámetro especial que puedes utilizar con los métodos `MEMBER` es el que recibe el nombre `SELF`. Este parámetro **hace referencia a una instancia (objeto) del mismo tipo de objeto**. Aunque no lo declares explícitamente, este parámetro siempre se declara automáticamente.

El **tipo de dato** correspondiente al parámetro `SELF` será el mismo que el del objeto original. En las funciones `MEMBER`, si no declaras el parámetro `SELF`, su modo por defecto se toma como `IN`. En cambio, en los procedimientos`MEMBER`, si no se declara, se toma como `IN OUT`. Ten en cuenta que no puedes especificar el modo `OUT` para este parámetro `SELF`, y que los métodos `STATIC` no pueden utilizar este parámetro especial.

Si se hace referencia al parámetro `SELF` **dentro del cuerpo de un método**, realmente se está haciendo referencia al objeto que ha invocado a dicho método. Por tanto, si utilizas `SELF.nombre atributo` o `SELF.nombre método`, estarás utilizando un atributo o un método del mismo objeto que ha llamado al método donde se encuentra utilizado el parámetro `SELF`.

```
MEMBER PROCEDURE setNombre(Nombre VARCHAR2) IS
   BEGIN
      /* El primer elemento (SELF.Nombre) hace referencia al atributo del tipo de objeto
mientras que el
      segundo (Nombre) hace referencia al parámetro del método */
      SELF.Nombre := Nombre;
   END setNombre;
```

¿Cómo se denonima a los elementos que realizan determinadas acciones sobre los objetos?

- ○ Atributos
- ◉ **Métodos**
- ○ Tipos de datos objeto
- ○ Parámetros

3.4.- Sobrecarga.

Al igual que ocurre con los subprogramas empaquetados, los métodos pueden ser sobrecargados, es decir, puedes **utilizar el mismo nombre para métodos diferentes** siempre que sus parámetros formales sean diferentes (en cantidad o tipo de dato).

Cuando se hace una llamada a un método, se comparan los parámetros actuales con los parámetros formales de los métodos que se han declarado, y se ejecutará aquél en el que haya una coincidencia entre ambos tipos de parámetros.

Ten en cuenta que no es válida la sobrecarga de dos métodos cuyos parámetros formales se diferencian únicamente en su modo, así como tampoco en funciones que se diferencien únicamente en el valor de retorno.

Estos dos ejemplos son correctos, ya que se diferencian en el número de parámetros:

```
MEMBER PROCEDURE setNombre(Nombre VARCHAR2)
MEMBER PROCEDURE setNombre(Nombre VARCHAR2, Apellidos VARCHAR2)
```

No es válido crear un nuevo método en el que se utilicen los mismos tipos de parámetros formales aunque se diferencien los nombres de los parámetros:

```
MEMBER PROCEDURE setNombre(Name VARCHAR2, Surname VARCHAR2)
```

¿Son correctas las siguientes declaraciones de métodos?

```
MEMBER FUNCTION getResultado(Valor VARCHAR2)
MEMBER FUNCTION getResultado(Valor INTEGER)
```

(•) **Verdadero** ( ) Falso

Aunque ambos métodos tengan el mismo nombre, incluso aunque el nombre de los parámetros sean iguales, los tipos de los parámetros son diferentes, por lo que el método getResultado está sobrecargado correctamente

3.5.- Métodos Constructores.

Cada tipo de objeto tiene un método constructor, que se trata de una **función con el mismo nombre que el tipo de objeto** y que se encarga de **inicializar los atributos y retornar una nueva instancia** de ese tipo de objeto.

Oracle crea un **método constructor por defecto** para cada tipo de objeto declarado, cuyos parámetros formales coinciden en orden, nombres y tipos de datos con los atributos del tipo de objeto.

También puedes declarar **tus propios métodos constructores**, reescribiendo ese método declarado por el sistema, o bien, definiendo un nuevo método con otros parámetros. Una de las ventajas de crear un nuevo método constructor personalizado es que se puede hacer una verificación de que los datos que se van a asignar a los atributos son correctos (por ejemplo, que cumplen una determinada restricción).

Si deseas reemplazar el método constructor por defecto, debes utilizar la sentencia `CONSTRUCTOR FUNCTION` seguida del nombre del tipo de objeto en el que se encuentra (recuerda que los métodos constructores tienen el mismo nombre que el tipo de objeto). A continuación debes indicar los parámetros que sean necesarios de la manera habitual. Por último, debes indicar que el valor de retorno de la función es el propio objeto utilizando la cláusula `RETURN SELF AS RESULT`.

Puedes crear **varios métodos constructores** siguiendo las restricciones indicadas para la sobrecarga de métodos.

En el siguiente ejemplo puedes ver la declaración y el cuerpo de un método constructor para el tipo de objeto Usuario. Como puedes comprobar, utiliza dos parámetros: login y crédito inicial. El cuerpo del método realiza un pequeño control para que en caso de que el crédito indicado sea negativo, se deje en cero:

```
CONSTRUCTOR FUNCTION Usuario(login VARCHAR2, credito NUMBER)
   RETURN SELF AS RESULT

CREATE OR REPLACE TYPE BODY Usuario AS
   CONSTRUCTOR FUNCTION Usuario(login VARCHAR2, credito NUMBER)
      RETURN SELF AS RESULT
   IS
      BEGIN
         IF (credito >= 0) THEN
            SELF.credito := credito;
         ELSE
            SELF.credito := 0;
         END IF;
         RETURN;
      END;
END;
```

4.- Utilización de objetos.

Caso práctico

__Ada__ ha observado que __Juan__ y __Ana__ están muy volcados en el desarrollo de algo. Ellos le cuentan que están realizando pruebas para modificar sus bases de datos relacionales para darles funcionalidades orientadas a objetos.

De momento han realizado la declaración de los tipos de datos objeto, pero no pueden enseñarle a __Ada__ su funcionamiento práctico, puesto que todavía no han creado objetos de esos tipos que ya tienen creados.

Le piden un poco de paciencia, porque pronto podrán enseñarle los beneficios que puede reportar esta técnica de trabajo con bases de datos y para el desarrollo de aplicaciones.

Una vez que dispones de los conocimientos necesarios para tener declarado un tipo de dato objeto, vamos a conocer la forma de utilizar los objetos que vayas a crear de ese tipo.

En los siguientes apartados vas a ver cómo puedes **declarar variables** que permitan almacenar objetos, darle unos **valores iniciales a sus atributos**, acceder al **contenido** de dichos atributos en cualquier momento, y **llamar a los métodos** que ofrece el tipo de objeto utilizado.

4.1.- Declaración de objetos.

Una vez que el tipo de objeto ha sido definido, éste puede ser utilizado para **declarar variables de objetos** de ese tipo en cualquier bloque PL/SQL, subprograma o paquete. Ese tipo de objeto lo puedes utilizar como tipo de dato para una variable, atributo, elemento de una tabla, parámetro formal, o resultado de una función, de igual manera que se utilizan los tipos de datos habituales como `VARCHAR` o `NUMBER`.

Por ejemplo, para declarar una variable denominada u1, que va a permitir almacenar un objeto del tipo Usuario, debes hacer la siguiente declaración:

```
u1 Usuario;
```

En la declaración de cualquier **procedimiento o función**, incluidos los métodos del mismo tipo de objeto o de otro, se puede utilizar el tipo de dato objeto definido para indicar que debe **pasarse como parámetro un objeto** de dicho tipo en la llamada. Por ejemplo pensemos en un procedimiento al que se le debe pasar como parámentro un objeto del tipo Usuario:

```
PROCEDURE setUsuario(u IN Usuario)
```

La llamada a este método se realizaría utilizando como parámetro un objeto, como el que podemos tener en la variable declarada anteriormente:

```
setUsuario(u1);
```

De manera semejante una función puede **retornar objetos**:

```
FUNCTION getUsuario(codigo INTEGER) RETURN Usuario
```

Los objetos se crean durante la ejecución del código como instancias del tipo de objeto, y cada uno de ellos puede contener valores diferentes en sus atributos.

El **ámbito de los objetos** sigue las mismas reglas habituales en PL/SQL, es decir, en un bloque o subprograma los objetos son creados (instanciados) cuando se entra en dicho bloque o subprograma y se destruyen automáticamente cuando se sale de ellos. En un paquete, los objetos son instanciados en el momento de hacer referencia al paquete y dejan de existir cuando se finaliza la sesión en la base de datos.

Suponiendo que tienes declarado el tipo de objeto Factura. ¿Cuál de las siguientes declaraciones de variable para guardar un objeto de ese tipo es correcta?

- Factura factura1;
- factura1 := Factura;
- **factura1 Factura;**

4.2.- Inicialización de objetos.

Para **crear o instanciar un objeto** de un determinado tipo de objeto, debes hacer una **llamada a su método constructor**. Esto lo puedes realizar empleando la **instrucción** `NEW` seguido del nombre del tipo de objeto como una llamada a una función en la que se indican como parámetros los valores que se desean asignar a los atributos inicialmente. En una asignación también puedes optar por hacer eso mismo omitiendo la palabra `NEW`.

El **orden de los parámetros** debe coincidir con el orden en el que están declarados los atributos, así como los tipos de datos. El formato sería como el siguiente:

```
variable_objeto := NEW Nombre_Tipo_Objeto (valor_atributo1, valor_atributo2, ...);
```

Por ejemplo, en el caso del tipo de objeto Usuario:

```
u1 := NEW Usuario('luitom64', 'LUIS ', 'TOMAS BRUÑA', '24/10/07', 100);
```

En ese momento se **inicializa el objeto**. Hasta que no se inicializa el objeto llamando a su constructor, el objeto tiene el valor `NULL`.

Es habitual inicializar los objetos en su declaración.

```
u1 Usuario := NEW Usuario('luitom64', 'LUIS ', 'TOMAS BRUÑA', '24/10/07', 100);
```

La llamada al método constructor se puede realizar en cualquier lugar en el que se puede hacer una llamada a una función de la forma habitual. Por ejemplo, la llamada al método constructor puede ser utilizada **como parte de una expresión**.

Los **valores de los parámetros** que se pasan al constructor cuando se hace la llamada, son asignados a los atributos del objeto que está siendo creado. Si la llamada es al método constructor que incorpora Oracle por defecto, debes indicar un parámetro para cada atributo, en el mismo orden en que están declarados los atributos. Ten en cuenta que los atributos, en contra de lo que ocurre con variables y constantes, no pueden tener valores por defecto asignados en su declaración. Por tanto, los valores que se desee que tengan inicialmente los atributos de un objeto instanciado deben indicarse como parámetros en la llamada al método constructor.

Existe la posibilidad de **utilizar los nombres de los parámetros formales** en la llamada al método constructor, en lugar de utilizar el modelo posicional de los parámetros. De esta manera no es obligatorio respetar el orden en el que se encuentran los parámetros reales respecto a los parámetros formales, que como se ha comentado antes coincide con el orden de los atributos.

```
DECLARE
   u1 Usuario;
BEGIN
   u1 := NEW Usuario('user1', -10);
   /* Se mostrará el crédito como cero, al intentar asignar un crédito negativo */
   dbms_output.put_line(u1.credito);
END;
/
```

¿Cuál de las siguientes inicializaciones de objetos es correcta para el tipo de objeto Factura, suponiendo que dispone de los atributos número (INTEGER), nombre (VARCHAR2) e importe (NUMBER)?

- **factura1 := NEW Factura(3, 'Juan Álvarez', 30.50);**
- factura1 = Factura(3, 'Juan Álvarez', 30.50);
- factura1 := NEW Factura('Juan Álvarez', 3, 30.50);
- factura1 := NEW (3, 'Juan Álvarez', 30.50);

4.3.- Acceso a los atributos de objetos.

Para hacer referencia a un atributo de un objeto debes utilizar el **nombre de dicho atributo**, **utilizando el punto** para acceder al valor que contiene o bien para modificarlo. Antes debe ir **precedido del objeto** cuyo atributo deseas conocer o modificar.

```
nombre_objeto.nombre_atributo
```

Por ejemplo, la consulta del valor de un atributo puede utilizarse como parte de una asignación o como parámetro en la llamada a una función:

```
unNombre := usuario1.nombre;
dbms_output.put_line(usuario1.nombre);
```

La **modificación del valor** contenido en el atributo puede ser similar a la siguiente:

```
usuario1.nombre:= 'Nuevo Nombre';
```

Los nombres de los atributos pueden ser encadenados, lo que permite el acceso a atributos de tipos de objetos anidados. Por ejemplo, si el objeto sitio1 tiene un atributo del tipo de objeto Usuario, se accedería al atributo del nombre del usuario con:

```
sitio1.usuario1.nombre
```

Si se utiliza en una expresión el acceso a un atributo de un objeto que **no ha sido inicializado**, se evalúa como `NULL`. Por otro lado, si se intenta asignar valores a un objeto no inicializado, éste lanza una excepción `ACCESS INTO NULL`.

Para comprobar si un objeto es `NULL` se puede utilizar el operador de comparación `IS NULL` con el que se obtiene el valor `TRUE` si es así.

De manera similar, al intentar hacer una llamada a un método de un objeto que no ha sido inicializado, se lanza una excepción `NULL_SELF_DISPATCH`. Si se pasa como parámetro de tipo IN, los atributos del objeto `NULL` se evalúan como `NULL`, y si el parámetro es de tipo `OUT` o `IN OUT` lanza una excepción al intentar modificar el valor de sus atributos.

¿Cuál de las siguientes expresiones es correcta para asignar el valor 50 al atributo importe del objeto factura1?

- **factura1.importe := 50;**
- importe.factura1 := 50;
- 50 := factura1.importe;

4.4.- Llamada a los métodos de los objetos.

Al igual que las llamadas a subprogramas, puedes invocar a los métodos de un tipo de objetos **utilizando un punto entre el nombre del objeto y el del método**. Los parámetros reales que se pasen al método se indicarán separados por comas, entre paréntesis, después del nombre del método.

```
usuario1.setNombreCompleto('Juan', 'García Fernández');
```

Si el método **no tiene parámetros**, se indicará la lista de parámetros reales vacía (sólo con los paréntesis), aunque se pueden omitir los paréntesis.

```
credito := usuario1.getCredito();
```

Las llamadas a métodos pueden **encadenarse**, en cuyo caso, el orden de ejecución de los métodos es de derecha a izquierda. Se debe tener en cuenta que el método de la izquierda **debe retornar un objeto** del tipo correspondiente al método de la derecha.

Por ejemplo, si dispones de un objeto sitio1 que tiene declarado un método getUsuario el cual retorna un objeto del tipo Usuario, puedes realizar con ese valor retornado una llamada a un método del tipo de objeto Usuario:

```
sitio1.getUsuario.setNombreCompleto('Juan', 'García Fernández');
```

Los **métodos** `MEMBER` son invocados utilizando una instancia del tipo de objeto:

```
nombre_objeto.metodo()
```

En cambio, los **métodos** `STATIC` se invocan usando el tipo de objeto, en lugar de una de sus instancias:

```
nombre_tipo_objeto.metodo()
```

¿Cuál de las siguientes llamadas al método getImporte es correcto para el objeto factura1?

- valor := getImporte.factura1();
- **valor := factura1.getImporte();**
- valor := getImporte().factura1;

4.5.- Herencia.

El lenguaje PL/SQL admite la herencia simple de tipos de objetos, mediante la cual, puedes **definir subtipos** de los tipos de objeto. Estos subtipos, o tipos heredados, **contienen todos los atributos y métodos del tipo padre**, pero además pueden contener **atributos y métodos adicionales**, o incluso sobreescribir métodos del tipo padre.

Para indicar que un tipo de objeto es **heredado** de otro hay que usar la **palabra reservada** `UNDER`, y además hay que tener en cuenta que el tipo de objeto del que **hereda** debe tener la **propiedad** `NOT FINAL`. Por defecto, los tipos de objeto se declaran como `FINAL`, es decir, que no se puede crear un tipo de objeto que herede de él.

Si no se indica lo contrario, siempre se pueden crear objetos (instancias) de los tipos de objeto declarados. Indicando la opción `NOT INSTANTIABLE` puedes declarar tipos de objeto de los que **no se pueden crear objetos**. Estos tipos tendrán la función de ser padres de otros tipos de objeto.

En el siguiente ejemplo puedes ver cómo se crea el tipo de objeto Persona, que se utilizará heredado en el tipo de objeto UsuarioPersona. De esta manera, este último tendrá los atributos de Persona

más los atributos declarados en UsuarioPersona. En la creación del objeto puedes observar que se deben asignar los valores para todos los atributos, incluyendo los heredados.

```
CREATE TYPE Persona AS OBJECT (
   nombre VARCHAR2(20),
   apellidos VARCHAR2(30)
) NOT FINAL;
/

CREATE TYPE UsuarioPersona UNDER Persona (
   login VARCHAR(30),
   f_ingreso DATE,
   credito NUMBER
);
/

DECLARE
   u1 UsuarioPersona;
BEGIN
   u1 := NEW UsuarioPersona('nombre1', 'apellidos1', 'user1', '01/01/2001', 100);
   dbms output.put line(u1.nombre);
END;
/
```

Un tipo de objeto que se ha declarado como hijo de otro, ¿hereda los atributos y métodos del tipo de objeto padre?

(•) Verdadero ( ) Falso

5.- Métodos MAP y ORDER.

Caso práctico

Juan *se ha dado cuenta de que va a tener un problema cuando necesite realizar comparaciones y órdenes entre objetos del mismo tipo.*

Cuando tenga una serie de objetos de tipo Usuario, y necesite realizar una operación de ordenación entre ellos, ¿cómo debe hacerlo? Podría indicar que se realizaran las consultas de forma ordenada en función de alguno de los atributos del tipo de objeto. Ha estudiado el funcionamiento de los métodos MAP y ORDER, y ha comprobado que son los que le van a ofrecer un mecanismo sencillo para ello.

Las instancias de un tipo de objeto no tienen un orden predefinido. Si deseas establecer un orden en ellos, con el fin de **hacer una ordenación o una comparación, debes crear un método** MAP.

Por ejemplo, si haces una comparación entre dos objetos Usuario, y deseas saber si uno es mayor que otro, ¿en base a qué criterio se hace esa comparación? ¿Por el orden alfabético de apellidos y nombre? ¿Por la fecha de alta? ¿Por el crédito? Hay que establecer con un método `MAP` cuál va a ser el valor que se va a utilizar en las comparaciones.

Para crear un método MAP debes declarar un método que **retorne el valor que se va a utilizar para hacer las comparaciones**. El método que declares para ello debe empezar su declaración con la palabra MAP:

```
CREATE OR REPLACE TYPE Usuario AS OBJECT (
   login VARCHAR2(30),
   nombre VARCHAR2(30),
   apellidos VARCHAR2(40),
   f_ingreso DATE,
   credito NUMBER,
   MAP MEMBER FUNCTION ordenarUsuario RETURN VARCHAR2
);
/
```

En el cuerpo del método se debe retornar el valor que se utilizará para realizar las comparaciones entre las instancias del tipo de objeto. Por ejemplo, si se quiere establecer que las comparaciones entre objetos del tipo Usuario se realice considerando el orden alfabético habitual de apellidos y nombre:

```
CREATE OR REPLACE TYPE BODY Usuario AS
   MAP MEMBER FUNCTION ordenarUsuario RETURN VARCHAR2 IS
   BEGIN
      RETURN (apellidos || ' ' || nombre);
   END ordenarUsuario;
END;
/
```

El lenguaje PL/SQL utiliza los métodos `MAP` para evaluar expresiones lógicas que resultan valores booleanos como objeto1 > objeto2, y para realizar las comparaciones implícitas en las cláusulas `DISTINCT`, `GROUP BY` y `ORDER BY`.

Cada tipo de objeto **sólo puede tener un método** `MAP` **declarado**, y sólo puede retornar alguno de los siguientes tipos: `DATE`, `NUMBER`, `VARCHAR2`, `CHARACTER` o `REAL`.

5.1.- Métodos ORDER.

De forma **similar al método** MAP, puedes declarar en cualquier tipo de objeto un método `ORDER` que te permitirá **establecer un orden** entre los objetos instanciados de dicho tipo.

Cada tipo de objeto **sólo puede contener un método** `ORDER`, el cual debe **retornar un valor numérico que permita establecer el orden entre los objetos**. Si deseas que un objeto sea menor que otro puedes retornar, por ejemplo, el valor -1. Si vas a determinar que sean iguales, devuelve 0, y si va a

ser mayor, retorna 1. De esta manera, considerando el valor retornado por el método `ORDER`, se puede establecer si un objeto es menor, igual o mayor que otro en el momento de hacer una ordenación entre una serie de objetos del mismo tipo.

Para declarar qué método va a realizar esta operación, debes indicar la palabra `ORDER` delante de su declaración. Debes tener en cuenta que va a **retornar un valor numérico (`INTEGER`)**, y que necesita que se indique un **parámetro que será del mismo tipo de objeto**. El objeto que se indique en ese parámetro será el que se compare con el objeto que utilice este método.

En el siguiente ejemplo, el sistema que se va a establecer para ordenar a los usuarios se realiza en función de los dígitos que se encuentran a partir de la posición 7 del atributo login.

```
CREATE OR REPLACE TYPE BODY Usuario AS
   ORDER MEMBER FUNCTION ordenUsuario(u Usuario) RETURN INTEGER IS
   BEGIN
      /* La función substr obtiene una subcadena desde la posición indicada hasta el final*/
      IF substr(SELF.login, 7) < substr(u.login, 7) THEN
         RETURN -1;
      ELSIF substr(SELF.login, 7) > substr(u.login, 7) THEN
         RETURN 1;
      ELSE
         RETURN 0;
      END IF;
   END;
END;
```

Con ese ejemplo, el usuario con login 'luitom64' se considera mayor que el usuario 'caragu62', ya que '64' es mayor que '62'.

El método debe retornar un número negativo, cero, o un número positivo que significará que el objeto que utiliza el método (`SELF`) es menor, igual, o mayor que el objeto pasado por parámetro.
Debes tener en cuenta que puedes declarar **un método `MAP` o un método `ORDER`, pero no los dos**.
Cuando se vaya a ordenar o mezclar un alto número de objetos, es preferible usar un método `MAP`, ya que en esos casos un método `ORDER` es menos eficiente.

¿Los métodos MAP sólo sirven para evaluar expresiones lógicas que resultan valores booleanos?

○ Verdadero ◉ **Falso**

También se pueden utilizar en consultas con las opciones DISTINCT, GROUP BY y ORDER BY

6.- Tipos de datos colección.

Caso práctico

__Ana__ ha estudiado los vectores y matrices que se usan en varios lenguajes de programación. Le pregunta a __Juan__ si con las bases de datos objeto-relacionales se puede utilizar algo parecido para almacenar los objetos instanciados. Él le dice que para eso existen las colecciones.

Para que puedas almacenar en memoria un conjunto de datos de un determinado tipo, la base de datos de Oracle te ofrece los tipos de datos __colección__.

Una **colección** es un conjunto de elementos del mismo tipo. Puede comparase con los **vectores y matrices** que se utilizan en muchos lenguajes de programación. En este caso, las colecciones sólo pueden tener **una dimensión** y los elementos se indexan mediante un valor de tipo numérico o cadena de caracteres.

La base de datos de Oracle proporciona los tipos `VARRAY` y `NESTED TABLE` (tabla anidada) como tipos de datos colección.

- ✓ Un `VARRAY` es una colección de elementos a la que se le establece una dimensión máxima que debe indicarse al declararla. Al tener una longitud fija, la eliminación de elementos no ahorra espacio en la memoria del ordenador.
- ✓ Una `NESTED TABLE` (**tabla anidada**) puede almacenar cualquier número de elementos. Tienen, por tanto, un tamaño dinámico, y no tienen que existir forzosamente valores para todas las posiciones de la colección.
- ✓ Una variación de las tablas anidadas son los **arrays asociativos**, que utilizan valores arbitrarios para sus índices. En este caso, los índices no tienen que ser necesariamente consecutivos.

Cuando necesites almacenar un número fijo de elementos, o hacer un recorrido entre los elementos de forma ordenada, o si necesitas obtener y manipular toda la colección como un valor, deberías utilizar el tipo `VARRAY`.

En cambio, si necesitas ejecutar consultas sobre una colección de manera eficiente, o manipular un número arbitrario de elementos, o bien realizar operaciones de inserción, actualización o borrado de forma masiva, deberías usar una`NESTED TABLE`.

Las colecciones pueden ser declaradas como una instrucción SQL o en el bloque de declaraciones de un programa PL/SQL. El tipo de dato de los elementos que puede contener una colección declarada en PL/SQL es cualquier tipo de dato PL/SQL, excepto REF CURSOR. Los elementos de las colecciones declaradas en SQL, además **no pueden ser de los tipos**: `BINARY_INTEGER`, `PLS_INTEGER`, `BOOLEAN`, `LONG`, `LONG RAW`, `NATURAL`, `NATURALN`, `POSITIVE`,`POSITIVEN`, `REF CURSOR`, `SIGNTYPE`, `STRING`.

Cualquier tipo de objetos declarado previamente puede ser utilizado como tipo de elemento para una colección.

Una tabla de la base de datos puede contener **columnas que sean colecciones**. Sobre una tabla que contiene colecciones se podrán realizar operaciones de consulta y manipulación de datos de la misma manera que se realiza con tablas con los tipos de datos habituales.

En este documento puedes encontrar más información sobre los tipos de datos colección entre otros conceptos de las bases de datos objetos-relacionales:
Anexo I - Soporte objeto-relacional en Oracle 8.

¿Qué tipo de colección tiene un tamaño fijo?

(•) VARRAY ( ) NESTED TABLE ( ) Array Asociativo

6.1.- Declaración y uso de colecciones.

La **declaración** de estos tipos de colecciones sigue el formato siguiente:

```
TYPE nombre_tipo IS VARRAY (tamaño max) OF tipo elemento;
TYPE nombre_tipo IS TABLE OF tipo_elemento;
TYPE nombre_tipo IS TABLE OF tipo_elemento INDEX BY tipo_índice;
```

Donde nombre_tipo representa el nombre de la colección, tamaño_max es el número máximo de elementos que podrá contener la colección, y tipo_elemento es el tipo de dato de los elementos que forman la colección. El tipo de elemento utilizado puede ser también cualquier tipo de objetos declarado previamente.

En caso de que la declaración se haga en SQL, fuera de un subprograma PL/SQL, se debe declarar con el formato `CREATE [OR REPLACE] TYPE`:

```
CREATE TYPE nombre_tipo IS ...
```

El tipo_índice representa el tipo de dato que se va a utilizar para el índice. Puede ser `PLS_INTEGER`, `BINARY INTEGER` o `VARCHAR2`. En este último tipo se debe indicar entre paréntesis el tamaño que se va a utilizar para el índice, por ejemplo, `VARCHAR2(5)`.

Hasta que no sea inicializada una colección, ésta es `NULL`. Para **inicializar** una colección debes utilizar el constructor, que es una función con el mismo nombre que la colección. A esta función se le pasa como parámetros los valores iniciales de la colección. Por ejemplo:

```
DECLARE
   TYPE Colores IS TABLE OF VARCHAR(10);
   misColores Colores;
BEGIN
   misColores := Colores('Rojo', 'Naranja', 'Amarillo', 'Verde', 'Azul');
END;
```

La inicialización se puede realizar en el bloque de código del programa, o bien, directamente en el bloque de declaraciones como puedes ver en este ejemplo:

```
DECLARE
   TYPE Colores IS TABLE OF VARCHAR(10);
   misColores Colores := Colores('Rojo', 'Naranja', 'Amarillo', 'Verde', 'Azul');
```

Para **obtener uno de los elementos** de la colección o modificar su contenido debes indicar el nombre de la colección seguido, entre paréntesis, del índice que ocupa el elemento deseado. Tanto en los `VARRAY` como en `NESTED TABLE`, el primer elemento tiene el índice 1.

Por ejemplo, para mostrar en pantalla el segundo elemento ('Naranja') de la colección Colores:

```
dbms_output.put_line(misColores(2));
```

En el siguiente ejemplo se modifica el contenido de la posición 3:

```
misColores(3) := 'Gris';
```

En el siguiente ejemplo puedes comprobar cómo pueden utilizarse las colecciones para almacenar sus datos en una tabla de la base de datos, así como la utilización con sentencias de consulta y manipulación de los datos de la colección que se encuentra en la tabla.

```
CREATE TYPE ListaColores AS TABLE OF VARCHAR2(20);
/
CREATE TABLE flores (nombre VARCHAR2(20), coloresFlor ListaColores)
   NESTED TABLE coloresFlor STORE AS colores_tab;

DECLARE
   colores ListaColores;
BEGIN
   INSERT INTO flores VALUES('Rosa', ListaColores('Rojo','Amarillo','Blanco'));
   colores := ListaColores('Rojo','Amarillo','Blanco','Rosa Claro');
   UPDATE flores SET coloresFlor = colores WHERE nombre = 'Rosa';
```

```
   SELECT coloresFlor INTO colores FROM flores WHERE nombre = 'Rosa';
END;/
```

7.- Tablas de objetos.

Caso práctico

*Aunque **Ana** ya ha aprendido a almacenar objetos en memoria gracias a las colecciones de objetos, necesita almacenarlos de manera persistente en una base de datos. **Juan** va a enseñarle que puede realizarlo de manera muy parecida a la gestión de tablas que ya conoce, en la que se usan los tipos de datos habituales de las bases de datos.*

Después de haber visto que un grupo de objetos se puede almacenar en memoria mediante colecciones, vas a ver en este apartado que también se pueden **almacenar los objetos en tablas** de igual manera que los tipos de datos habituales de las bases de datos.

Los tipos de datos objetos se pueden usar para formar una tabla exclusivamente formado por elementos de ese tipo, o bien, para usarse como un tipo de columna más entre otras columnas de otros tipos de datos.

En caso de que desees crear una **tabla formada exclusivamente por un determinado tipo de dato objeto**, (tabla de objetos) debes utilizar la sentencia `CREATE TABLE` junto con el tipo de objeto de la siguiente manera:

```
CREATE TABLE NombreTabla OF TipoObjeto;
```

Siendo `NombreTabla` el nombre que deseas dar a la tabla que va a almacenar los objetos del tipo TipoObjeto. Por ejemplo, para crear la tabla UsuariosObj, que almacene objetos del tipo Usuario:

```
CREATE TABLE UsuariosObj OF Usuario;
```

Debes tener en cuenta que si una tabla hace uso de un tipo de objeto, **no podrás eliminar ni modificar la estructura de dicho tipo de objeto**. Por tanto, desde el momento en que el tipo de objeto sea utilizado en una tabla, no podrás volver a definirlo.

Para poder crear este ejemplo previamente debes tener declarado el tipo de objeto Usuario, que se ha utilizado en apartados anteriores.

Al crear una tabla de esta manera, estamos consiguiendo que podamos almacenar objetos del tipo Usuario en una tabla de la base de datos, quedando así sus datos **persistentes** mientras no sean eliminados de la tabla. Anteriormente hemos instanciado objetos que se han guardado en variables, por lo que al terminar la ejecución, los objetos, y la información que contienen, desaparecen. Si esos objetos se almacenan en tablas no desaparecen hasta que se eliminen de la tabla en la que se encuentren.

Cuando se instancia un objeto con el fin de almacenarlo en una tabla, dicho objeto no tiene identidad fuera de la tabla de la base de datos. Sin embargo, el tipo de objeto existe independientemente de cualquier tabla, y puede ser usado para crear objetos en cualquier modo.

Las tablas que sólo contienen filas con objetos, reciben el nombre de **tablas de objetos**.

En la siguiente imagen se muestra el contenido de la tabla que incluye dos filas de objetos de tipo Usuario. Observa que los atributos del tipo de objeto se muestran como si fueran las columnas de la tabla:

En este documento puedes encontrar información general sobre las bases de datos objeto-relacionales, con algunos ejemplos de creación de tablas de objetos:
Bases de datos orientadas a objetos. (0.19 MB)

7.1.- Tablas con columnas tipo objeto.

Puedes usar cualquier tipo de objeto, que hayas declarado previamente, para utilizarlo como un **tipo de dato de una columna más en una tabla** de la base de datos. Así, una vez creada la tabla, puedes utilizar cualquiera de las sentencias SQL para insertar un objeto, seleccionar sus atributos y actualizar sus datos.

Para crear una tabla en el que alguno de sus columnas sea un tipo de objeto, simplemente debes hacerlo como si fuera una columna como las que has utilizado hasta ahora, pero en el tipo de dato debes especificar el tipo de objeto.

Por ejemplo, podemos crear una tabla que contenga, entre otras columnas, una columna de objetos del tipo Usuario que hemos utilizado anteriormente.

```
CREATE TABLE Gente (
   dni VARCHAR2(10),
   unUsuario Usuario,
   partidasJugadas SMALLINT
);
```

Como puedes comprobar en la siguiente imagen, los datos del campo unUsuario se muestran como integrantes de cada objeto Usuario, a diferencia de la tabla de objetos que has visto en el apartado anterior. Ahora todos los atributos del tipo de objeto Usuario no se muestran como si fueran varias columnas de la tabla, sino que forman parte de una única columna.

En las tablas con columnas de tipo objeto, ¿los atributos se muestran como columnas de la tabla?

(•) Falso ( ) Verdadero

En las tablas con columnas de tipo objeto, se muestran los atributos como parte del objeto

7.2.- Uso de la sentencia Select.

De manera similar a las consultas que has realizado sobre tablas sin tipos de objetos, **puedes utilizar la sentencia** `SELECT` para obtener datos de las filas almacenadas en tablas de objetos o tablas con columnas de tipos de objetos.

El uso más sencillo sería para mostrar todas las filas contenidas en la tabla:

```
SELECT * FROM NombreTabla;
```

Como puedes apreciar en la imagen, la tabla que forme parte de la consulta puede ser una tabla de objetos (como la tabla UsuariosObj), o una tabla que contiene columnas de tipos de objetos (como la tabla Gente).

En las sentencias `SELECT` que utilices con objetos, puedes incluir cualquiera de las **cláusulas y funciones de agrupamiento** que has aprendido para la sentencia `SELECT` que has usado anteriormente con las tablas que contienen columnas de tipos básicos. Por ejemplo, puedes utilizar: `SUM`, `MAX`, `WHERE`, `ORDER`, `JOIN`, etc.

Es habitual utilizar **alias** para hacer referencia al nombre de la tabla. Observa, por ejemplo, la siguiente consulta, en la que se desea obtener el nombre y los apellidos de los usuarios que tienen algo de crédito:

```
SELECT u.nombre, u.apellidos FROM UsuariosObj u WHERE u.credito > 0
```

Si se trata de una **tabla con columnas de tipo objeto**, el acceso a los atributos del objeto se debe realizar indicando previamente el nombre asignado a la columna que contiene los objetos:

```
SELECT g.unUsuario.nombre, g.unUsuario.apellidos FROM Gente g;
```

En este documento puedes encontrar información general sobre las bases de datos objeto-relacionales, con algunos ejemplos de uso de la sentencia `SELECT` con tablas de objetos:
Anexo II - Tipos de Objeto en PL/SQL.

7.3.- Inserción de objetos.

Evidentemente, no te servirá de nada una tabla que pueda contener objetos sino conocemos la manera de insertar objetos en la tabla.

La manera que tienes para ello **es la misma** que has utilizado para introducir datos de cualquier tipo habitual en las tablas de la base de datos: usando la sentencia `INSERT` de SQL.

En las tablas habituales, cuando querías añadir una fila a una tabla que tenía un campo `VARCHAR` le suministrabas a la sentencia `INSERT` un dato de ese tipo. Pues si la tabla es de un determinado tipo de objetos, o si posee un campo de un determinado tipo de objeto, tendrás que suministrar a la sentencia `INSERT` un objeto instanciado de su tipo de objeto correspondiente.

Por tanto, si queremos insertar un Usuario en la tabla Gente que hemos creado en el apartado anterior, previamente debemos crear el objeto o los objetos que se deseen insertar. A continuación podremos utilizarlos dentro de la sentencia INSERT como si fueran valores simplemente.

```
DECLARE
   u1 Usuario;
   u2 Usuario;
BEGIN
   u1 := NEW Usuario('luitom64', 'LUIS', 'TOMAS BRUNA', '24/10/2007', 50);
   u2 := NEW Usuario('caragu72', 'CARLOS', 'AGUDO SEGURA', '06/07/2007', 100);
   INSERT INTO UsuariosObj VALUES (u1);
   INSERT INTO UsuariosObj VALUES (u2);
END;
```

De una manera más directa, puedes crear el objeto dentro de la sentencia `INSERT` directamente, sin necesidad de guardar el objeto previamente en una variable:

```
INSERT INTO UsuariosObj VALUES (Usuario('luitom64', 'LUIS', 'TOMAS BRUNA', '24/10/2007', 50));
```

Podrás comprobar los resultados haciendo una consulta `SELECT` sobre la tabla de la manera habitual:

```
SELECT * FROM UsuariosObj;
```

De manera similar puedes realizar la **inserción de filas en tablas con columnas de tipo objeto**. Por ejemplo, para la tabla Gente que posee entre sus columnas, una de tipo objeto Usuario, podríamos usar el formato de instanciar el objeto directamente en la sentencia `INSERT`, o bien, indicar una variable que almacena un objeto que se ha instanciado anteriormente:

```
INSERT INTO Gente VALUES ('22900970P', Usuario('luitom64', 'LUIS', 'TOMAS BRUNA',
'24/10/2007', 50), 54);

INSERT INTO Gente VALUES ('62603088D', u2, 21);
```

7.4.- Modificación de objetos.

Si deseas modificar un objeto almacenado en una tabla tan sólo tienes que utilizar las **mismas sentencias SQL** que disponías para modificar registros de una tabla. ¿Recuerdas la sentencia `UPDATE`? Ahora puedes volver a utilizarla para modificar también los objetos de la tabla, de igual manera que cualquier otro tipo de dato.

Hay una pequeña diferencia en la forma de especificar los nombre de los campos afectados, en función del tipo de tabla: según sea una tabla de objetos, o bien una tabla con alguna columna de tipo objeto.

Si se trata de una tabla de objetos, se hará referencia a los atributos de los objetos justo detrás del nombre asignado a la tabla. Sería algo similar al formato siguiente:

```
UPDATE NombreTabla
   SET NombreTabla.atributoModificado = nuevoValor
   WHERE NombreTabla.atributoBusqueda = valorBusqueda;
```

Continuando con el ejemplo empleado anteriormente, vamos a suponer que deseas modificar los datos de un determinado usuario. Por ejemplo, modifiquemos el crédito del usuario identificado por el login 'luitom64', asignándole el valor 0.

```
UPDATE UsuariosObj
   SET UsuariosObj.credito = 0
   WHERE UsuariosObj.login = 'luitom64';
```

Es muy habitual abreviar el nombre de la tabla con un **alias**:

```
UPDATE UsuariosObj u
   SET u.credito = 0
   WHERE u.login = 'luitom64';
```

Pero no sólo puedes cambiar el valor de un determinado atributo del objeto. Puedes **cambiar un objeto por otro** como puedes ver en el siguiente ejemplo, en el que se sustituye el usuario con login 'caragu72' por otro usuario nuevo.

```
UPDATE UsuariosObj u SET u = Usuario('juaesc82', 'JUAN', 'ESCUDERO LARRASA', '10/04/2011', 0)
WHERE u.login = 'caragu72';
```

Si se trata de una tabla con columnas de tipo objeto, se debe hacer referencia al nombre de la columna que contiene los objetos:

```
UPDATE NombreTabla
   SET NombreTabla.colObjeto.atributoModificado = nuevoValor
   WHERE NombreTabla.colObjeto.atributoBusqueda = valorBusqueda;
```

A continuación puedes ver un ejemplo de actualización de datos de la tabla que se había creado con una columna del tipo de objeto Usuario. Recuerda que a la columna en la que se almacenaban los objetos de tipo Usuario se le había asignado el nombre unUsuario:

```
UPDATE Gente g
   SET g.unUsuario.credito = 0
   WHERE g.unUsuario.login = 'luitom64';
```

O bien, puedes cambiar todo un objeto por otro, manteniendo el resto de los datos de la fila sin modificar, como en el siguiente ejemplo, donde los datos de DNI y partidasJugadas no se cambia, sólo se cambia un usuario por otro.

```
UPDATE Gente g
   SET g.unUsuario = Usuario('juaesc82', 'JUAN', 'ESCUDERO LARRASA', '10/04/2011', 0)
   WHERE g.unUsuario.login = 'caragu72';
```

7.5.- Borrado de objetos.

Por supuesto, no nos puede faltar una sentencia que nos permita eliminar determinados objetos almacenados en tablas. Al igual que has podido comprobar en las operaciones anteriores, tienes a tu disposición la misma sentencia que has podido utilizar en las operaciones habituales sobre tablas. En este caso de borrado de objetos deberás utilizar la sentencia `DELETE`.

El modo de uso de `DELETE` sobre objetos almacenados en tablas es muy similar al utilizado hasta ahora:

```
DELETE FROM NombreTablaObjetos;
```

Recuerda que si no se indica ninguna condición, se **eliminarán todos** los objetos de la tabla, por lo que suele ser habitual utilizar la sentencia `DELETE` con una condición detrás de la cláusula `WHERE`. Los objetos o filas de la tabla que**cumplan con la condición** indicada serán los que se eliminen.

```
DELETE FROM NombreTablaObjetos WHERE condición;
```

Observa el siguiente ejemplo en el que se borrarán de la tabla UsuariosObj, que es una tabla de objetos, los usuarios cuyo crédito sea 0. Observa que se utiliza un alias para el nombre de la tabla:

```
DELETE FROM UsuariosObj u WHERE u.credito = 0;
```

De manera similar se puede realizar el borrado de filas en tablas en las que alguna de sus columnas son objetos. Puedes comprobarlo con el siguiente ejemplo, donde se utiliza la tabla Gente, en la que una de sus columnas (unUsuario) es del tipo de objeto Usuario que hemos utilizado en otros apartados anteriores.

```
DELETE FROM Gente g WHERE g.unUsuario.credito = 0;
```

Esta sentencia, al igual que las anteriores, se puede **combinar con otras consultas** `SELECT`, de manera que en vez de realizar el borrado sobre una determinada tabla, se haga sobre el resultado de una consulta, o bien que la condición que determina las filas que deben ser eliminadas sea también el resultado de una consulta. Es decir, todo lo aprendido sobre las operaciones de manipulación de datos sobre las tablas habituales, se puede aplicar sobre tablas de tipos de objetos, o tablas con columnas de tipos de objetos.

7.6.- Consultas con la función VALUE.

Cuando tengas la necesidad de **hacer referencia a un objeto en lugar de alguno de sus atributos**, puedes utilizar la función `VALUE` junto con el nombre de la tabla de objetos o su alias, **dentro de una sentencia** `SELECT`. Puedes ver a continuación un ejemplo de uso de dicha función para hacer inserciones en otra tabla (Favoritos) del mismo tipo de objetos:

```
INSERT INTO Favoritos SELECT VALUE(u) FROM UsuariosObj u WHERE u.credito >= 100;
```

Esa misma función `VALUE` puedes utilizarla para hacer comparaciones de igualdad entre objetos, por ejemplo, si deseamos obtener datos de los usuarios que se encuentren en las tablas Favoritos y UsuariosObj.

```
SELECT u.login FROM UsuariosObj u JOIN Favoritos f ON VALUE(u)=VALUE(f);
```

Observa la diferencia en el uso cuando se hace la comparación con una columna de tipo de objetos. En ese caso la referencia que se hace a la columna (g.unUsuario) permite obtener directamente un objeto, sin necesidad de utilizar la función `VALUE`.

```
SELECT g.dni FROM Gente g JOIN Favoritos f ON g.unUsuario=VALUE(f);
```

Usando la **cláusula** `INTO` **podrás guardar en variables el objeto obtenido** en las consultas usando la función VALUE. Una vez que tengas asignado el objeto a la variable podrás hacer uso de ella de cualquiera de las formas que has visto anteriormente en la manipulación de objetos. Por ejemplo, puedes acceder a sus atributos, formar parte de asignaciones, etc.

En el siguiente ejemplo se realiza una consulta de la tabla UsuariosObj para obtener un determinado objeto de tipo Usuario. El objeto resultante de la consulta se guarda en la variable u1. Esa variable se utiliza para mostrar en pantalla el nombre del usuario, y para ser asignada a una segunda variable, que contendrá los mismos datos que la primera.

```
DECLARE
   u1 Usuario;
   u2 Usuario;
BEGIN
   SELECT VALUE(u) INTO u1 FROM UsuariosObj u WHERE u.login = 'luitom64';
```

```
   dbms_output.put_line(u1.nombre);
   u2 := u1;
   dbms output.put line(u2.nombre);
END;
```

En este documento puedes encontrar información general sobre las bases de datos objeto-relacionales, incluyendo información y ejemplos de la función VALUE:
Anexo III - Bases de datos objeto-relacionales.

7.7.- Referencias a objetos.

El paso de objetos a un método resulta ineficiente cuando se trata de objeto de gran tamaño, por lo que es más conveniente **pasar un puntero a dicho objeto**, lo que permite que el método que lo recibe pueda hacer referencia a dicho objeto sin que sea necesario que se pase por completo. Ese puntero es lo que se conoce en Oracle como una **referencia (`REF`)**.

Al compartir un objeto mediante su referencia, **los datos no son duplicados**, por lo que cuando se hace cualquier cambio en los atributos del objeto, se producen en un único lugar.

Cada objeto almacenado en una tabla tiene un **identificador de objeto** que identifica de forma única al objeto guardado en una determinada fila y sirve como una referencia a dicho objeto.

Las referencias se crean utilizando el modificador `REF` delante del tipo de objeto, y se puede usar con variables, parámetros, campos, atributos, e incluso como variables de entrada o salida para sentencias de manipulación de datos en SQL.

```
CREATE OR REPLACE TYPE Partida AS OBJECT (
   codigo INTEGER,
   nombre VARCHAR2(20),
   usuarioCreador REF Usuario
);
/

DECLARE
   u ref REF Usuario;
   p1 Partida;
BEGIN
   SELECT REF(u) INTO u_ref FROM UsuariosObj u WHERE u.login = 'luitom64';
   p1 := NEW Partida(1, 'partida1', u_ref);
END;
/
```

Hay que tener en cuenta que sólo se pueden usar **referencias a tipos de objetos que han sido declarados previamente**. Siguiendo el ejemplo anterior, no se podría declarar el tipo Partida antes que el tipo Usuario, ya que dentro del tipo Partida se utiliza una referencia al tipo Usuario. Por tanto, primero debe estar declarado el tipo Usuario y luego el tipo Partida.

El problema surge cuando tengamos dos tipos que utilizan referencias mutuas. Es decir, un atributo del primer tipo hace referencia a un objeto del segundo tipo, y viceversa. Esto se puede solucionar haciendo una **declaración de tipo anticipada**. Se realiza indicando únicamente el nombre del tipo de objeto que se detallará más adelante:

```
CREATE OR REPLACE TYPE tipo2;
/
CREATE OR REPLACE TYPE tipo1 AS OBJECT (
   tipo2 ref REF tipo2
   /*Declaración del resto de atributos del tipo1*/
);
/
CREATE OR REPLACE TYPE tipo2 AS OBJECT (
   tipo1 ref REF tipo1
   /*Declaración del resto de atributos del tipo2*/
);
```

```
/
```

7.8.- Navegación a través de referencias.

Debes tener en cuenta que **no se puede acceder directamente a los atributos de un objeto referenciado que se encuentre almacenado en una tabla**. Para ello, puedes utilizar la función `DEREF`.

Esta función **toma una referencia a un objeto y retorna el valor** de ese objeto.

Vamos a verlo en un ejemplo suponiendo que disponemos de las siguientes variables declaradas

```
u_ref REF Usuario;
u1 Usuario;
```

Si `u ref` hace referencia a un objeto de tipo Usuario que se encuentra en la tabla `UsuariosObj`, para obtener información sobre alguno de los atributos de dicho objeto referenciado, hay que utilizar la función `DEREF`.

Esta función se utiliza **como parte de una consulta** `SELECT`, por lo que hay que utilizar una tabla tras la cláusula `FROM`. Esto puede resultar algo confuso, ya que las referencias a objetos apuntan directamente a un objeto concreto que se encuentra almacenado en una determinada tabla. Por tanto, no debería ser necesario indicar de nuevo en qué tabla se encuentra. Realmente es así. Podemos hacer referencia a cualquier tabla en la consulta, y la función `DEREF` nos devolverá el objeto referenciado que se encuentra en su tabla correspondiente.

La base de datos de Oracle ofrece la **tabla** DUAL para este tipo de operaciones. Esta tabla es creada de forma automática por la base de datos, es accesible por todos los usuarios, y **tiene un solo campo y un solo registro**. Por tanto, es como una tabla comodín.

```
SELECT DEREF(u_ref) INTO u1 FROM Dual;
dbms_output.put_line(u1.nombre);
```

Por tanto, para obtener el objeto referenciado por una variable `REF`, debes **utilizar una consulta sobre cualquier tabla**, independientemente de la tabla en la que se encuentre el objeto referenciado. Sólo existe la condición de que siempre se obtenga una solo fila como resultado. Lo más **cómodo es utilizar esa tabla** DUAL. Aunque se use esa tabla comodín, **el resultado será un objeto** almacenado en la tabla UsuariosObj.

En este documento puedes encontrar información general sobre todo lo tratado en esta unidad, incluyendo ejemplos de tablas de objetos con uso de referencias:
Anexo IV - Bases de Datos Objeto-Relacionales en Oracle 8.

Anexo I - Soporte objeto-relacional en Oracle 8

Original en: http://alarcos.inf-cr.uclm.es/doc/bbddavanzadas/08-09/Documentacion-PracticaII.pdf

1. Registros PL/SQL

Un registro es un grupo de datos relacionados, almacenados en campos, cada uno de los cuales tiene su propio nombre y tipo y que se tratan como una sola unidad lógica.

Los campos de un registro pueden ser inicializados y pueden ser definidos como `NOT NULL`. Aquellos campos que no sean inicializados explícitamente, se inicializarán a `NULL`.
Los registros pueden estar anidados.

1.1. Declaración de tipos y de variables registro

Declaración del tipo registro:
Sintaxis:

```
TYPE nombre_tipo_reg IS RECORD
(declaración_campo [,declaración_campo]...);
```

Donde `declaración campo` tiene la forma:

```
nombre_campo
{tipo campo | variable%TYPE | table.columns%TYPE | table%ROWTYPE }
 [ [NOT NULL] {:=  | DEFAULT} expresión]
```

Declaración de una variable del tipo registro:

```
id_variable nombre_tipo_reg;
```

1.2. Acceso a un campo de una variable de tipo registro

```
 id_variable.nombre_campo
```

1.3. Asignación de valores

- ✓ Puede asignarse un valor directamente a un campo:
  `Id.campo := 3;`
- ✓ Puede asignarse valor a todos los campos de un registro utilizando una sentencia `SELECT`. En este caso hay que tener cuidado en especificar las columnas en el orden conveniente según la declaración de los campos del registro.
- ✓ Puede asignarse un registro a otro siempre y cuando sean del mismo tipo

1.4. Declaración de registros con el atributo %ROWTYPE

Se puede declarar un registro basándose en una colección de columnas de una tabla o una vista de la base de datos mediante el atributo `%ROWTYPE`.

Por ejemplo, si tengo una tabla empleado declarada como:

```
CREATE TABLE empleado(
id number,
nombre char(10),
apellido char(20),
dirección char(30));
```

Puedo declarar una variable de tipo registro como:

```
 DECLARE
 reg_emp empleado%ROWTYPE;
```

Lo cual significa que el registro `reg emp` tendrá la siguiente estructura:

```
id number,
nombre char(10),
apellido char(20),
dirección char(30)
```

De esta forma se podrán asignar valores a los campos de un registro a través de un select sobre la tabla a partir de la cual se creo el registro.

2. Tablas PL/SQL

Una tabla PL/SQL :

- ✓ Es similar a un array
- ✓ Tiene dos componentes: Una clave primaria de tipo `BINARY_INTEGER` que permite indexar en la tabla PL/SQL y una columna de escalares o registros que contiene los elementos de la tabla PL/SQL
- ✓ Puede incrementar su tamaño dinámicamente.

2.1. Creación de una tabla

Declaración de un tipo tabla PL/SQL.
Sintaxis:

```
TYPE nombre_tipo_tabla IS TABLE OF
{tipo columna | variable%TYPE | tabla.columna%TYPE}[NOT NULL]
INDEX BY BINARY_INTEGER ;
```

Declaración de una variable del tipo

```
nombre_var nombre_tipo_tabla;
```

No es posible inicializar las tablas en la inicialización.

2.2. Referenciar un elemento de la tabla

Sintaxis:

```
Pl/sql_nombre_tabla(valor_primary_key);
```

El rango de binary integer es –2147483647.. 2147483647, por lo tanto el índice puede ser negativo, lo cual indica que el índice del primer valor no tiene que ser necesariamente el uno.

3. Tablas PL/SQL de registros

Es posible declarar elementos de una tabla PL/SQL como de tipo registro.
Para referenciar un elemento se hará de la siguiente forma:

```
nombre_tabla(índice).nombre_campo
```

4. Funciones para el manejo de tablas PL/SQL

Función `EXISTS(i)`. Utilizada para saber si en un cierto índice hay almacenado un valor. Devolverá `TRUE` si en el índice i hay un valor.
Función `COUNT`. Devuelve el número de elementos de la tabla PL/SQL.
Función `FIRST`. Devuelve el menor índice de la tabla. `NULL` si está vacía.
Función `LAST`. Devuelve el mayor índice de la tabla. NULL si está vacía.
Función `PRIOR (n)`. Devuelve el número del índice anterior a n en la tabla.
Función `NEXT (n)`. Devuelve el número del índice posterior a n en la tabla.
Función `TRIM`. Borra un elemento del final de la tabla PL/SQL. `TRIM(n)` borra **n** elementos del final de la tabla PL/SQL.
Función `DELETE`. Borra todos los elementos de la tabla PL/SQL. `DELETE(n)` borra el correspondiente al índice **n**. `DELETE(m,n)` borra los elementos entre **m** y **n**.

5. Tablas anidadas (Nested tables)

Un tipo colección es aquel que maneja varias variables como una unidad. La versión 2 de PL/SQL sólo tenía un tipo de dato colección, las tablas PL/SQL vistas anteriormente.

La versión 8 de Oracle PL/SQL añade dos tipos colección nuevos las tablas anidadas y los varrays. Cada uno de estos tipos de colecciones puede interpretarse como un tipo de objeto con atributos y métodos.

Las tablas anidadas son muy parecidas a las tablas PL/SQL (también llamadas indexadas). Las tablas anidadas añaden funcionalidad a las indexadas al añadir métodos de colección adicionales y la capacidad de almacenar tablas anidadas en una tabla de la base de datos. Estas tablas también pueden manejarse directamente utilizando órdenes SQL.

Aparte de esto, la funcionalidad básica de una tabla anidada es la misma que la de una tabla PL/SQL

5.1. Sintaxis

```
TYPE nombre_tabla IS TABLE OF tipo_tabla [NOT NULL];
```

La única diferencia con la declaración de una tabla indexada es que ahora no aparece la cláusula `INDEX BY BINARY_INTEGER`.

5.2. Inicialización de una tabla

Para inicializar una tabla anidada hay que utilizar el constructor (igual que se hace con los objetos). Este constructor tendrá el mismo nombre que la tabla anidada. A continuación podrán ponerse los valores con los que queramos inicializar o nada (en cuyo caso se inicializa a una tabla sin elementos, que es diferente que una tabla a `NULL`).

Ejemplo.

```
set serveroutput on;
DECLARE
TYPE tabla numero IS TABLE OF NUMBER;
var tabla 1  tabla numero := tabla numero(0);
var tabla 2 tabla numero := tabla numero(1, 2, 3, 4);
var_tabla_3 tabla_numero := tabla_numero();
var_tabla_4 tabla_numero;
BEGIN
   var tabla 1(1):= 123;
   DBMS_OUTPUT.PUT_LINE('Valor 1 de var_1: ' || var_tabla_1(1));
   DBMS_OUTPUT.PUT_LINE('Valor 1 de var_2: ' || var_tabla_2(1));
   IF var_tabla_3 IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('var_tabla_3 SI es NULL');
   ELSE
      DBMS OUTPUT.PUT LINE('var tabla 3 NO es NULL');
   END IF;
   IF var_tabla_4 IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('var_tabla_4 SI es NULL');
   ELSE
      DBMS_OUTPUT.PUT_LINE('var_tabla_4 NO es NULL');
   END IF;
END;
/
```

La salida será:

```
Valor 1 de var_1: 123
Valor 1 de var_2: 1
var_tabla_3 NO es NULL
var_tabla_4 SI es NULL
```

5.3. Claves en la inicialización

Cuando se inicializa una tabla utilizando el constructor, los elementos de la tabla se numeran secuencialmente desde 1 hasta el número de elementos especificado en la llamada al constructor. Es posible eliminar un elemento mediante el uso de `DELETE` (que se verá más adelante en esta misma práctica). En caso de borrado en una tabla anidada de la base de datos, las claves se renumeran para seguir siendo secuenciales.

Ejemplo

```
set serveroutput on;
DECLARE
TYPE tabla_numero IS TABLE OF NUMBER;
var_tabla_2 tabla_numero := tabla_numero(10, 20, 30, 40);
BEGIN
   DBMS OUTPUT.PUT LINE('Nro elem de var tabla 2: ' ||
   var tabla 2.count);
   DBMS OUTPUT.PUT LINE('Primer elem: ' || var tabla 2.first);
   DBMS OUTPUT.PUT LINE('Último elem: ' || var tabla 2.last);
END;
/
```

La salida será:

```
NRO ELEM DE VAR_TABLA_2: 4
PRIMER ELEM: 1
ÚLTIMO ELEM: 4
```

5.4. Adición de elementos a una tabla existente

Aunque las tablas no tienen un tamaño fijo, no se puede asignar un valor a un elemento que todavía no existe. Para poder añadir elementos hay que utilizar el método `EXTEND` (que se verá más adelante en esta misma práctica).

Ejemplo.

```
set serveroutput on;
DECLARE
TYPE tabla numero IS TABLE OF NUMBER;
var tabla 2 tabla numero := tabla numero(10, 20, 30, 40);
BEGIN
   var_tabla_2 (1) := 50;
   DBMS_OUTPUT.PUT_LINE('Primer elemento de var_tabla_2: ' ||
   var_tabla_2(1) );
END;
/
```

La salida es:

```
PRIMER ELEMENTO DE VAR_TABLA_2: 50
PROCEDIMIENTO PL/SQL TERMINADO CON ÉXITO.
```

Pero si pongo:

```
set serveroutput on;
DECLARE
TYPE tabla numero IS TABLE OF NUMBER;
var_tabla_2 tabla_numero := tabla_numero(10, 20, 30, 40);
BEGIN
   var_tabla_2 (1) := 50;
   DBMS_OUTPUT.PUT_LINE('Primer elemento de var_tabla_2: ' ||
   var_tabla_2(1) );
   Var tabla 2 (5) := 60;
   DBMS_OUTPUT.PUT_LINE('Quinto elemento de var_tabla_2: ' ||
   var_tabla_2(5));
END;
/
```

Salida:

```
PRIMER ELEMENTO DE VAR_TABLA_2: 50
DECLARE
```

```
*
ERROR EN LÍNEA 1:
ORA-06533: SUBÍNDICE MAYOR QUE EL RECUENTO
ORA-06512: EN  LÍNEA 7
```

5.5. Tablas anidadas en la base de datos

Una tabla anidada se puede almacenar como una columna de una tabla. Para ello hay que definir la tabla anidada con la orden `CREATE TYPE` para crear el tipo de la tabla anidada en lugar de `TYPE`.

Ejemplo

```
CREATE TYPE inf_libro AS OBJECT(
   titulo varchar2(40),
   nombre_autor varchar2(40),
   isbn number
);
CREATE TYPE  isbn libros AS TABLE OF inf libro;
CREATE TABLE prestamo (
   fecha_entrega date,
   nro_socio number(10),
   libros_prestados isbn_libros)
   NESTED TABLE  libros_prestados STORE AS prestados_tabla;
```

5.6. Manipulación de tablas completas

Una tabla anidada almacenada en una tabla de la base de datos se puede manipular en su integridad o pueden manipularse sus filas individuales. Cualquiera de los dos casos, se pueden utilizar órdenes SQL. El índice de la tabla anidada sólo puede utilizarse cuando la tabla está en PL/SQL.

Inserción

Para insertar una fila en una tabla anidada se utiliza la orden `INSERT`. Hay que tener en cuenta que la tabla se crea e inicializa en PL/SQL y después se inserta en la base de datos.

Ejemplo inserción.

```
DECLARE
libros isbn libros := isbn libros(inf libro('La ruta no natural', 'Macario Polo',
1234567));
BEGIN
   INSERT INTO prestamo VALUES (sysdate, 12,
   isbn libros( inf libro('Métricas para bases de datos', 'Coral Calero', 234567),
      inf libro('La amigdalitis de Tarzán', 'Alfredo Bryce', 3456)));

   INSERT INTO prestamo VALUES (sysdate, 24, libros);
END;
/
```

Modificación

De forma similar, se utiliza `UPDATE` para modificar la tabla almacenada

Ejemplo modificación

```
DECLARE
libros isbn_libros := isbn_libros(inf_libro('La ruta no natural', 'Macario Polo',
1234567), inf_libro('La amigdalitisTarzá', 'Alfredo Bryce', 3456));
BEGIN
   UPDATE prestamo
   SET libros_prestados = libros
   WHERE nro_socio = 24;
END;
/
```

Eliminación

`DELETE` puede eliminar una fila que contenga una tabla anidada.

Ejemplo eliminación

```
BEGIN
   DELETE FROM prestamo WHERE nro_socio = 24;
END;
```

Selección

Cuando se recupera una tabla anidada en una variable PL/SQL, se asignan claves comenzando por 1 hasta llegar al número de elementos que contiene la tabla. Este valor puede determinarse mediante el método `COUNT`, el cual se describe más adelante. Se trata de las mismas claves establecidas por el constructor.

Ejemplo selección

```
Set serveroutput on;
DECLARE
libros isbn_libros;
i NUMBER;

BEGIN
   SELECT libros_prestados INTO libros FROM prestamo WHERE nro_socio=12;
FOR i IN 1 .. libros.count LOOP
   DBMS_OUTPUT.PUT_LINE('Título: ' || libros(i).titulo|| ' del elemento: '|| i);
END LOOP;
END;
/
```

Salida:

```
Título: Métricas para bases de datos del elemento: 1
Título: La amigdalitis de Tarzán del elemento: 2
Procedimiento PL/SQL terminado con éxito.
```

6. VARRAYS

Un varray se manipula de forma muy similar a las tablas indexadas o anidadas pero se implementa de forma diferente. Los elementos en el varray se almacenan comenzando en el índice 1 hasta la longitud máxima declarada en el tipo varray.

6.1. Declaración de un varray

El tipo_elementos puede especificarse utilizando `%TYPE`. Sin embargo, no puede ser `BOOLEAN`, `NCHAR`, `NCLOB`, `NVARCHAR(2)`, `REF CURSOR`, `TABLE` u otro `VARRAY`.

```
 TYPE nombre_tipo IS VARRAY (tamaño_maximo) OF tipo_elementos;
```

6.2. Inicialización de un varray

De forma similar a las tablas, los `VARRAY` se inicializan utilizando un constructor.

Ejemplo

```
DECLARE
   TYPE tipo_numeros IS VARRAY(20) OF NUMBER(3);
 nros uno tipo numeros;
 nros dos tipo numeros := tipo numeros(1,2);
 nros tres tipo numeros := tipo numeros(NULL);
 BEGIN
   IF nros_uno IS NULL THEN
      DBMS_OUTPUT.PUT_LINE(' nros_uno es NULL');
   END IF;
   IF nros tres IS NULL THEN
      DBMS_OUTPUT.PUT_LINE(' nros_tres es NULL');
   END IF;
   IF nros_tres(1) IS NULL THEN
      DBMS_OUTPUT.PUT_LINE(' nros_tres(1) es NULL');
   END IF;
END;
/
```

Salida:

```
NROS_UNO ES NULL
NROS_TRES(1) ES NULL
PROCEDIMIENTO PL/SQL TERMINADO CON ÉXITO.
```

6.3. Manipulación de los elementos de un varray

Como las tablas anidadas, el tamaño inicial de un `VARRAY` se establece mediante el número de elementos utilizados en el constructor usado para declararlo. Si se hacen asignaciones a elementos que queden fuera del rango se producirá un error.

Ejemplo.

```
DECLARE
   TYPE t cadena IS VARRAY(5) OF VARCHAR2(10);
   v_lista t_cadena:= ('Paco', 'Pepe', 'Luis');
BEGIN
   v lista(2) := 'Lola';
   v lista(4) := 'Está mal';
END;
/
```

El tamaño de un `VARRAY` podrá aumentarse utilizando la función `EXTEND` que se verá más adelante, pero nunca con mayor dimensión que la definida en la declaración del tipo. Por ejemplo, la variable `v_lista` que sólo tiene 3 valores definidos por lo que se podría ampliar pero nunca más allá de cinco.

6.4. Varrays en la base de datos

Los VARRAYS pueden almacenarse en las columnas de la base de datos. Sin embargo, un varray sólo puede manipularse en su integridad, no pudiendo modificarse sus elementos individuales de un varray.

Ejemplo

```
CREATE OR REPLACE TYPE lista_libros AS VARRAY(10) OF inf_libro;
CREATE TABLE ejemplo(
   id number,
   libros inf_libro);
```

6.5. Manipulación de varrays almacenados

Para modificar un varray almacenado, primero hay que seleccionarlo en una variable PL/SQL. Luego se modifica la variable y se vuelve a almacenar en la tabla.

7. VARRAYS y Tablas Anidadas

Los varrays y las tablas anidadas son colecciones y, como tales, tienen algunos aspectos similares:

- ✓ Ambos tipos permiten el acceso a elementos individuales utilizando la notación con subíndices
- ✓ Ambos tipos pueden almacenarse en la base de datos

Pero también existen diferencias:

- ✓ Los varrays tienen un tamaño máximo y las tablas anidadas no
- ✓ Los varrays se almacenan junto con la tabla que los contiene mientras que las tablas anidadas se almacenan en una tabla separada, que puede tener diferentes características de almacenamiento
- ✓ Cuando están almacenados en la base de datos, los varrays mantienen el orden y los valores de los subíndices para los elementos, mientras que las tablas anidadas no.
- ✓ Los elementos individuales se pueden borrar de una tabla anidada usando el método TRIM por lo que el tamaño de la tabla disminuye. Un varray siempre tiene un tamaño constante.

8. Métodos de colecciones

Los métodos de colecciones sólo se pueden llamar desde órdenes procedimentales y no desde órdenes SQL.

Todos se verán con un ejemplo basado en la siguiente declaración:

```
CREATE OR REPLACE TYPE NumTab AS TABLES OF NUMBER;
CREATE OR REPLACE TYPE NumVar AS VARRAY(25) OF NUMBER;
```

EXISTS

Se usa para averiguar si en realidad existe el elemento referenciado.

Sintaxis:

```
EXISTS(n)
```

Donde n es una expresión entera. Si el elemento existe (incluso aunque sea NULL) devuelve TRUE.

```
LOOP
IF tabla.EXISTS(v_count) THEN
   EXISTE
   v count:=v count+1;
ELSE
   EXIT;
END IF;
END LOOP;
```

COUNT

Devuelve un número entero correspondiente al número de elementos que tiene actualmente una colección.

```
 VarTab NumTab:=(1,2,3);
 VarVec NumVar := (-1, -2, -3, -4);
 BEGIN
   DBMS...(VarTab.COUNT, VarVec.COUNT);
 END;
```

LIMIT

Devuelve el número máximo actual de elementos de una colección. Siempre devuelve `NULL` cuando se aplica a una tabla anidada (ya que no tienen tamaño máximo). Para los `VARRAY` devolverá el máximo valor definido en la declaración.

FIRST y LAST

`FIRST` devuelve el índice del primer elemento de la colección y `LAST` el último. En un `VARRAY`, `FIRST` siempre devuelve 1 y `LAST` siempre devuelve el valor de `COUNT`.

NEXT y PRIOR

Se utilizan para incrementar o decrementar la clave de una colección.

`NEXT(n)` Clave del elemento inmediatamente posterior a **n**.

`PRIOR(n)` Clave del elemento inmediatamente anterior a **n**.

Si no existe (el posterior o anterior) devuelve NULL.

`EXTEND`

Se usa para añadir elementos al final de una tabla anidada. Tiene tres formatos:

`EXTEND` Añade un elemento `NULL` con índice `LAST +1` al final de la tabla

`EXTEND(n)` Añade **n** elementos con valor `NULL` al final de la tabla.

`EXTEND(n,i)` Añade **n** copias del elemento i al final de la tabla.

Si la tabla se ha creado con restricciones NOT NULL sólo se podrá utilizar el último formato.

DELETE

Elimina uno o más elementos de una tabla anidada. Tiene tres formatos:
`DELETE` Elimina la tabla completa
`DELETE(n)` Elimina el elemento con el índice **n**
`DELETE(m,n)` Elimina todos los elementos entre **m** y **n**.

Si un elemento que se va a borrar no existe, `DELETE` no da error y lo salta

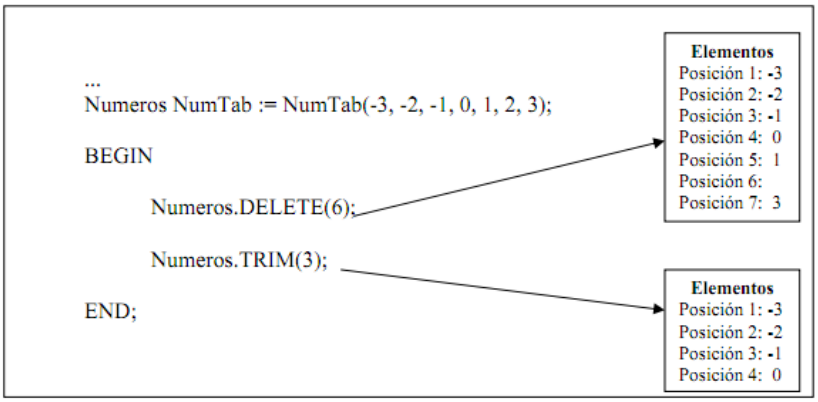
TRIM

Elimina elementos del final de una tabla anidada. `TRIM` no tiene efecto cuando se usa sobre un `varray`.

Tiene dos formatos:
`TRIM` Elimina el elemento del final de la colección
`TRIM (n)` Elimina **n** elementos.

Si n es mayor que `COUNT` se genera una excepción. `TRIM` opera sobre el tamaño interno de la colección, incluyendo cualquier elemento eliminado con un `DELETE`.

Ejemplo.



9. Paquetes y resolución del problema de las tablas mutantes

Finalmente, vamos a ver como solucionar el problema de las tablas mutantes utilizando disparadores y tablas PL/SQL.

Una tabla es mutante sólo para los disparadores a nivel de fila. No se puede usar, sin más, un disparador a nivel de orden, porque, por lo general, necesitamos acceder a valores que han sido modificados (de ahí el problema de las tablas mutantes).

La solución consiste en crear dos disparadores, uno a nivel de fila y otro a nivel de orden:

- ✓ En el disparador con nivel de fila almacenamos (en una estructura de datos apropiada) los datos que queremos consultar (los que provocan el error de tabla mutante)
- ✓ En el disparador con nivel de orden realizamos la consulta (pero sobre los datos almacenados en lugar de sobre la tabla)

La mejor forma de almacenar los valores es en una tabla PL/SQL y aunar todas las operaciones descritas dentro de un paquete.

Por ejemplo, dada la siguiente tabla:

```
CREATE TABLE estudiantes(
id   INTEGER PRIMARY KEY,
```

```
especialidad VARCHAR2(20));
```

Con los siguientes valores:

```
insert into estudiantes values(1001, 'Historia');
insert into estudiantes values(1002, 'Historia');
insert into estudiantes values(1003, 'Informática');
insert into estudiantes values(1004, 'Matemáticas');
insert into estudiantes values(1005, 'Informática');
insert into estudiantes values(1006, 'Historia');
insert into estudiantes values(1007, 'Informática');
insert into estudiantes values(1008, 'Matemáticas');
insert into estudiantes values(1009, 'Historia');
insert into estudiantes values(10010, 'Informática');
insert into estudiantes values(10011, 'Historia');
```

y creamos el siguiente disparador:

```
CREATE OR REPLACE TRIGGER limite_especialidad
BEFORE INSERT OR UPDATE OF especialidad ON estudiantes
FOR EACH ROW
DECLARE
maxEstudiantes CONSTANT NUMBER:=5;
EstudiantesActuales NUMBER;
BEGIN
SELECT COUNT(*) INTO EstudiantesActuales
FROM estudiantes
WHERE especialidad = :new.especialidad;
IF EstudiantesActuales+1>maxEstudiantes THEN
   RAISE APPLICATION ERROR(-20000, 'Demasiados estudiantes
en la especialidad: '|| :new.especialidad);
END IF;
END limite_especialidad;
```

Si ejecutamos

```
UPDATE estudiantes
SET especialidad = 'Historia'
WHERE id=1003;
```

Nos da un error de tabla mutante.

Para arreglarlo definimos el siguiente paquete:

```
CREATE OR REPLACE PACKAGE DatosEstudiantes AS
TYPE TipoEspecialidad IS TABLE OF estudiantes.especialidad%TYPE
INDEX BY BINARY_INTEGER;
TYPE TipoIdentificador IS TABLE OF estudiantes.id%TYPE INDEX BY
BINARY_INTEGER;
EspecEst TipoEspecialidad;
IdEst TipoIdentificador;
NumEntradas BINARY INTEGER:=0;
END DatosEstudiantes;
```

Creamos el disparador a nivel de fila para almacenar los nuevos datos:

```
CREATE OR REPLACE TRIGGER FilaLimiteEspecialidad
BEFORE INSERT OR UPDATE OF especialidad ON estudiantes
FOR EACH ROW
BEGIN
DatosEstudiantes.NumEntradas := DatosEstudiantes.NumEntradas + 1;
DatosEstudiantes.EspecEst(DatosEstudiantes.NumEntradas) :=
:new.especialidad;
DatosEstudiantes.IdEst(DatosEstudiantes.NumEntradas) := :new.id;
END FilaLimiteEspecialidad;
```

Creamos el disparador a nivel de orden para dar la funcionalidad que queríamos:

```
CREATE OR REPLACE TRIGGER OrdenLimiteEspecialidad
AFTER INSERT OR UPDATE OF especialidad ON estudiantes
DECLARE
maxEstudiantes CONSTANT NUMBER:=5;
EstudiantesActuales NUMBER;
EstudianteId estudiantes.id%TYPE;
```

```
LaEspecialidad estudiantes.especialidad%TYPE;
BEGIN
FOR indice IN 1..DatosEstudiantes.NumEntradas LOOP
EstudianteId := DatosEstudiantes.IdEst(indice);
LaEspecialidad := DatosEstudiantes.EspecEst(indice);
SELECT COUNT(*) INTO EstudiantesActuales
   FROM estudiantes
   WHERE especialidad = LaEspecialidad;
IF EstudiantesActuales+1>maxEstudiantes THEN
   RAISE APPLICATION ERROR(-20000, 'Demasiados estudiantes
en la especialidad: '|| LaEspecialidad);
END IF;
END LOOP;
DatosEstudiantes.NumEntradas := 0;
END OrdenLimiteEspecialidad;
```

Si de nuevo ejecutamos

```
UPDATE estudiantes
SET especialidad = 'Historia'
WHERE id=1003;
```

Ya no nos da el error de tabla mutante sino que nos devuelve:

DEMASIADOS ESTUDIANTES EN LA ESPECIALIDAD: HISTORIA

Pero si ejecutamos:

```
UPDATE estudiantes
SET especialidad = 'Filología'
WHERE id=1003;
```

Sintaxis del bucle FOR:

```
FOR i IN valor_ini.. valor_fin LOOP
   Cuerpo For
END LOOP;
```

Para escribir valores por pantalla:

```
dbms_output.put_line ('text' ||var);
```

Pero para que funcione, al comenzar la sesión hay que ejecutar

```
set serveroutput on;
```

Anexo II - Tipos de Objeto en PL/SQL

Original en: http://www.kybele.etsii.urjc.es/docencia/BD/2011-2012/Material/%5BBD-2010-2011%5DPLSQL.ObjectTypes.pdf

Introducción

Una BDOR soporta las dos tecnologías: relacional y objeto-relacional

Aportaciones OR

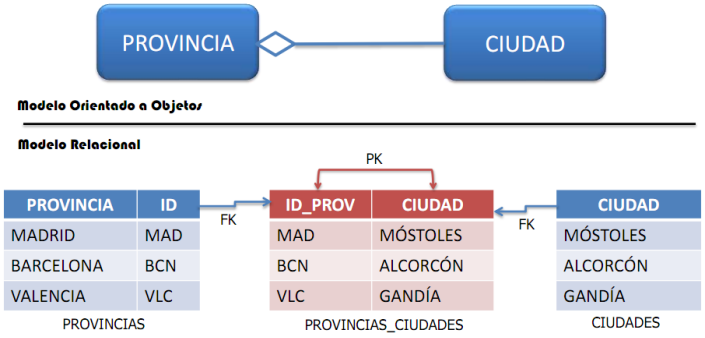
- ✓ UDTs
  - ➔ Atributos / Métodos …
- ✓ Tipos Complejos en una columna
- ✓ Referencias

Oracle incorpora estas mejoras desde la versión 8i

Las estructuras de almacenamiento siguen siendo tablas, pero se pueden usar mecanismos de orientación al objeto para la gestión de datos

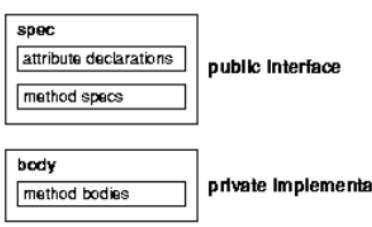
Cada objeto tiene un tipo, se almacena en una fila de una tabla y tiene un identificador que permite referenciarlo desde otros objetos (otras filas de otras tablas)

Los objetos permiten modelar mejor las relaciones "parte-todo".



Estructura de un tipo de objeto

Los parámetros no deben restringirse en tamaño. Se invocan igual que se recupera un campo (Especificando los parámetros si fuera el caso). Definir un tipo no implica reservar espacio para objetos del tipo




```
CREATE [OR REPLACE] TYPE Complejo AS OBJECT (
parte_r   REAL, -- attributo
parte_e   REAL,
MEMBER FUNCTION suma (x Complejo) RETURN Complejo, -- metodo
MEMBER FUNCTION resta (x Complejo) RETURN Complejo,
```

```
CREATE [OR REPLACE] TYPE BODY Complejo AS
   MEMBER FUNCTION suma (x Complejo) RETURN Complejo IS
   BEGIN
      RETURN Complejo(parte_r + x.parte_r, parte_e + x.parte_e);
   END plus;

   MEMBER FUNCTION resta (x Complejo) RETURN Complejo IS
   BEGIN
      RETURN Complejo(parte_r - x.parte_r, parte_e - x.parte_e);
   END less;
END;
```

Tablas de objetos

Las tablas tipadas son tablas que almacenan objetos del tipo sobre el que han sido definidas. Cada fila almacena un objeto

También podemos verlas como una tabla con una única columna del tipo objeto y una tabla con tantas columnas como el tipo objeto

Almacenan un objeto en cada fila y permiten acceder a los campos del objeto como si fueran columnas de la tabla. Las restricciones se definen sobre la tabla y se aplicarán SÓLO sobre aquellos objetos almacenados en dicha tabla

```
CREATE OR REPLACE TYPE Tipo_Coche AS OBJECT (
    Marca          VARCHAR2(25),
    Modelo         VARCHAR2(25),
    Matricula      VARCHAR2(9))
/

CREATE TABLE COCHES OF Tipo_Coche;
```

```
CREATE OR REPLACE TYPE Tipo_Coche AS OBJECT (
    Marca          VARCHAR2(25),
    Modelo         VARCHAR2(25),
    Matricula      VARCHAR2(9))
/

CREATE TABLE COCHES OF Tipo_Coche;

ALTER TABLE COCHES ADD
                CONSTRAINT PK_COCHE PRIMARY KEY (Marca, Modelo);
```

Métodos

Implementan el comportamiento de los objetos del tipo y pueden ser `MEMBER`, `STATIC` o `CONSTRUCTOR`

```
CREATE TYPE Tipo_Cubo AS OBJECT (
        largo           INTEGER,
        ancho           INTEGER,
        alto            INTEGER,
        MEMBER FUNCTION superficie RETURN INTEGER,
        MEMBER FUNCTION volumen RETURN INTEGER,
        MEMBER PROCEDURE mostrar ());
/
CREATE TYPE BODY Tipo_Cubo AS
        MEMBER FUNCTION volume RETURN INTEGER IS
        BEGIN
                RETURN largo * ancho * alto;
                -- RETURN SELF.largo * SELF.ancho * SELF.alto;
        END;
        MEMBER FUNCTION superficie RETURN INTEGER IS
        BEGIN
                RETURN 2 * (largo * ancho + largo * alto + ancho * alto);
        END;
        MEMBER PROCEDURE mostrar ()
        IS
        BEGIN
                DBMS_OUTPUT.PUT_LINE('Largo: ' || largo || ' - ' || 'Ancho: ' || ancho || ' - ' || 'Alto: ' || alto);
                DBMS_OUTPUT.PUT_LINE('Volumen: ' || volumen || ' - ' || 'Superficie: ' || superficie);
        END;
END;
/
```

La variable SELF permite referirse al objeto sobre el que se invocó la función/procedimiento

Método MEMBER

```
CREATE TYPE Tipo_Cubo AS OBJECT (…);
/

CREATE TYPE BODY Tipo_Cubo AS …
END;
/

CREATE TABLE Cubos of Tipo_Cubo;
INSERT INTO Cubos VALUES(Tipo_Cubo (10, 10, 10));
INSERT INTO Cubos VALUES(Tipo_Cubo (3, 4, 5));
SELECT * FROM Cubos;
SELECT c.volumen(), c.superficie () FROM cubos c
WHERE c.largo = 10;

DECLARE
        mi_cubo      Tipo_Cubo;
BEGIN

  SELECT VALUE(c) INTO mi_cubo FROM Cubos c
  WHERE c.largo = 10;
  mi_cubo.mostrar ();
END;
/
DROP TABLE Cubos;
DROP TYPE tipo_Cubo FORCE;
```

Métodos STATIC:
Operaciones globales, que no son de los objetos, sino del tipo

```
CREATE TYPE Tipo_Cubo AS OBJECT(
 ….,
STATIC PROCEDURE nuevoCubo (
  v_largo      INTEGER,
  v_ancho      INTEGER,
  v_alto       INTEGER));
/
CREATE TYPE BODY Tipo_Cubo AS
STATIC PROCEDURE nuevoCubo (
                v_largo         INTEGER,
                v_ancho         INTEGER,
                v_alto          INTEGER)
IS
  sqlstmt VARCHAR2(100);
BEGIN
  sqlstmt := 'INSERT INTO Cubos '||schname||'.'||tabname|| '
             VALUES (Tipo_Cubo(largo, ancho, alto))';
  EXECUTE IMMEDIATE sqlstmt;
END;
/

BEGIN
   Tipo_Cubo.nuevoCubo(1, 1, 1);
END;
/
```

Constructores

Cada vez que se crea un tipo de objeto, Oracle crea automáticamente un método constructor identificado por el mismo nombre del tipo. Es una función que devuelve una nueva instancia del tipo definido por el usuario y establece los valores de sus atributos.

Recibe como parámetros los atributos del tipo. Debe ser siempre explícitamente invocado cada vez que se desee crear un objeto del tipo

Métodos constructor

```
CREATE OR REPLACE TYPE Tipo_Coche AS OBJECT (
    Marca          VARCHAR2(25),
    Modelo         VARCHAR2(25),
    Matricula      VARCHAR2(9))
/

CREATE OR REPLACE TYPE Tipo_Persona AS OBJECT (
    Nombre         VARCHAR2(25),
    Coche          Tipo_Coche)
/

CREATE TABLE PERSONAS OF Tipo_Persona;

INSERT INTO PERSONAS VALUES ('Ramón Ramirez',
                             Tipo_Coche('CITROEN', '2-CV', 'M-9999999'));
```

Llamada al constructor del tipo para crear el objeto embebido

Declarar e inicializar objetos

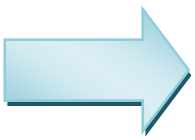
Podemos usar un tipo de objeto igual que cualquier otro tipo de dato. Una buena práctica es inicializar los objetos al declararlos

```
DECLARE
  r Tipo_Persona; -- r toma valor NULL
BEGIN
  IF r IS NULL THEN ... – esta comparación devuelve TRUE
  IF r > (2/3) … -- pero esta comparación TAMBIÉN devuelve NULL
  r := Tipo_Persona('Ramón Ramírez', NULL);
```

Acceso a los atributos de un objeto

Para acceder a las propiedades de un objeto se utiliza la notación punto (**.**)

```
CREATE TYPE Tipo_Coche AS OBJECT (
    Marca          VARCHAR2(25),
    Modelo         VARCHAR2(25),
    Matricula      VARCHAR2(9));
/

CREATE TYPE Tipo_Persona AS OBJECT (
    Nombre         VARCHAR2(25),
    Coche          Tipo_Coche);
/
```



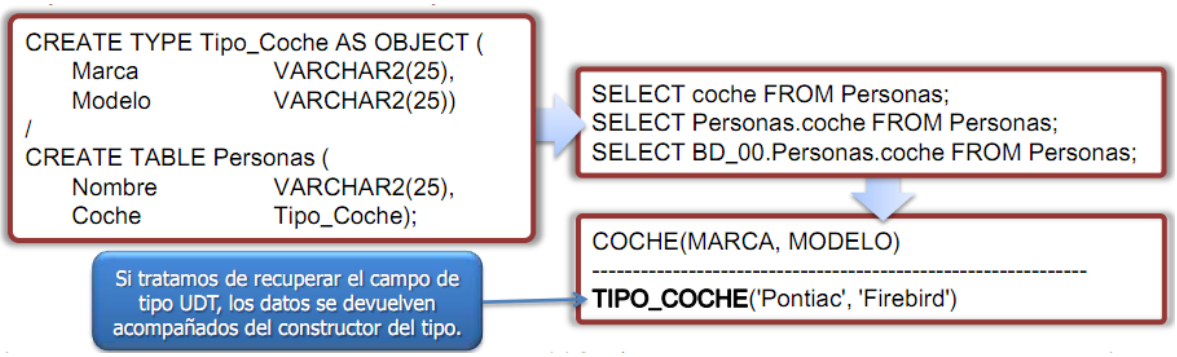
```
DECLARE
  v_persona Tipo_Persona;
BEGIN
  v_persona.coche.marca := 'CITROEN';
  -- error ACCESS_INTO_NULL
  v_persona := Tipo_Persona(NULL,…);
  v_persona.coche.marca := 'CITROEN';
  …
END;
```

Variables de correlación

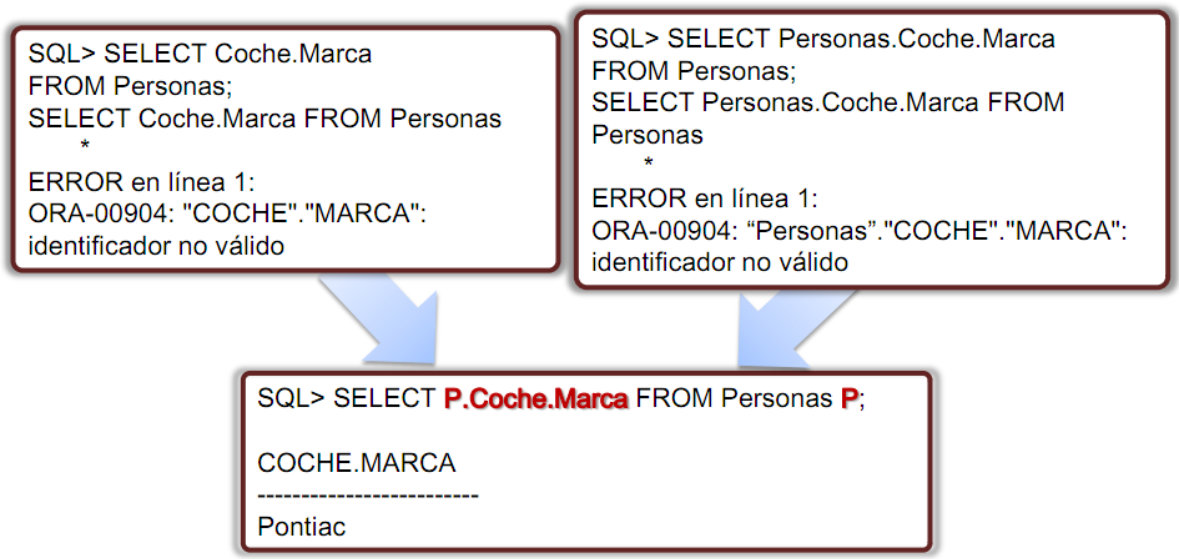
Son un tipo especial de variable PL/SQL. En general, las correlation variables se identifican con los alias de tabla

```
SELECT E.Nombre FROM Empleado E;
SELECT Empleado.Nombre FROM Empleado;
```

Son opcionales y podemos omitirlas, pero cuando trabajamos con UDTs es OBLIGATORIO utilizarlas para acceder a los campos del UDT



Igualmente, para acceder a los campos del UDT por separado hay que utilizar las correlation variables



Las correlation variables permiten resolver ambigüedades

```
SQL> SELECT Cliente.Coche.Marca;
```

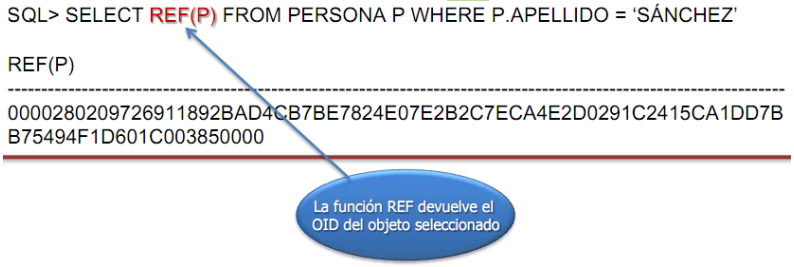
`Cliente` es un nombre de usuario, `Coche` es una tabla y `Marca` una columna
`Cliente` es un nombre de tabla, `Coche` de columna y `Marca` un campo de esa columna

El uso de la variable resuelve el problema

```
SQL> SELECT C.Coche.Marca;
```

OID - Object IDentifier

Cada fila de una tabla tipada (`OR`) tendrá un identificador del objeto fila → OID
Para guardar esos identificadores Oracle utiliza un tipo `REF`



Tipos referencia

Cada fila (objeto fila) podrá ser referenciada como un objeto a través de su OID

```
CREATE OR REPLACE TYPE Tipo_Persona AS OBJECT (
    Nombre    VARCHAR2(25),
    Coche     Tipo_Coche)
/

CREATE OR REPLACE TYPE Tipo_Empresa AS OBJECT (
    Nombre    VARCHAR2(25),
    NIF       VARCHAR2(25),
    Director  REF Tipo_Persona)
/
CREATE TABLE PERSONAS OF Tipo_Persona;
CREATE TABLE EMPRESAS OF Tipo_Empresa;

SQL> DESC EMPRESAS
 Nombre                                    ¿Nulo?   Tipo
 ----------------------------------------- -------- ------------------------------------
 NOMBRE                                             VARCHAR2(25)
 NIF                                                VARCHAR2(25)
 DIRECTOR                                           TIPO_PERSONA()


SQL> SELECT * FROM EMPRESAS;
Nombre                    NIF          DIRECTOR
------------------------- ------------ ----------------
'Juan Nadie'              '000001'     0000280209726911892BAD4CB7BE7824E07E ....
```

Para obtener una referencia a un objeto utilizamos el operador `REF`

```
DECLARE
  persona_ref REF Persona;
BEGIN
  SELECT REF(p) INTO persona_ref FROM Personas p
  WHERE p.nombre = 'Pepe Pérez';
END;
/
```

Una columna de tipo `REF` guarda un puntero a una fila de la otra tabla. Contiene el OID de dicha fila

```
INSERT INTO EMPRESAS VALUES ('ACME', '666',
        (SELECT REF(P) FROM PERSONAS P WHERE P.NOMBRE LIKE '%Ramirez'));

SELECT * FROM EMPRESAS;
NOMBRE   NIF  DIRECTOR
---------- ----  --------------------------------------------------------------------------------
ACME       666  0000220208EB2D8E93BE39430EB8D325E255E81F5E2F8521EDFD3F4
```

Las referencias no se pueden navegar en PL/SQL

```
DECLARE
  p_ref REF Persona;
  telefono VARCHAR2(15);
BEGIN
  telefono := p_ref.telefono; -- no permitido
  ..............
```

En su lugar hay que usar la función `DEREF` (o el paquete `UTL_REF`), que permite obtener el objeto referenciado

```
DECLARE
  p1 Persona;
  p_ref REF Persona;
  nombre VARCHAR2(15);
BEGIN ...
  -- Si p_ref tiene una referencia válida a una fila de una tabla
  SELECT DEREF(p_ref) INTO p1 FROM dual;
  nombre :=   p1.nombre;
  .....
SQL> SELECT DEREF(E.Director) FROM Empresas E;

DEREF(E.DIRECTOR)(NOMBRE, COCHE(MARCA, MODELO, MATRICULA))
--------------------------------------------------------------------------------
TIPO_PERSONA('Ramón Ramirez', TIPO_COCHE('CITROEN', '2-CV', 'M-9999999'))
```

En cambio si se pueden navegar en SQL

```
CREATE OR REPLACE TYPE Tipo_Persona AS OBJECT (
    Nombre        VARCHAR2(25),
    Coche         Tipo_Coche)
/
CREATE OR REPLACE TYPE Tipo_Empresa AS OBJECT (
    Nombre        VARCHAR2(25),
    NIF           VARCHAR2(25),
    Director      REF Tipo_Persona)
/
CREATE TABLE PERSONAS OF Tipo_Persona;
CREATE TABLE EMPRESAS OF Tipo_Empresa (
    SCOPE FOR (Director) IS Personas);

SELECT E.Director.Nombre FROM Empresas E
WHERE E.Nombre = 'ACME';
```

DEREF implícito

Podemos restringir el conjunto de objetos a los que apuntará la `REF` a los contenidos en una única tabla. De esta forma el almacenamiento ocupará menos espacio y el acceso será más eficiente

```
CREATE OR REPLACE TYPE Tipo_Persona AS OBJECT (
    Nombre          VARCHAR2(25),
    Coche           Tipo_Coche)
/
CREATE OR REPLACE TYPE Tipo_Empresa AS OBJECT (
    Nombre          VARCHAR2(25),
    NIF             VARCHAR2(25),
    Director        REF Tipo_Persona)
/
CREATE TABLE PERSONAS OF Tipo_Persona;
CREATE TABLE EMPRESAS OF Tipo_Empresa (
    SCOPE FOR (Director) IS Personas);
```

Operador VALUE

Para obtener el objeto almacenado en una fila (y no sólo el valor de los campos de dicho objeto) se necesita la función `VALUE`

```
SELECT * FROM Personas;

NOMBRE
--------------------
COCHE(MARCA, MODELO, MATRICULA)
--------------------------------------------------------------------------------
Ramón Ramirez
TIPO_COCHE('CITROEN', '2-CV', 'M-9999999')
```

La consulta sólo devuelve el valor de los campos del objeto de la clase Tipo_Persona

```
SQL> SELECT VALUE(P) FROM PERSONAS P;

VALUE(P)(NOMBRE, COCHE(MARCA, MODELO, MATRICULA))
--------------------------------------------------------------------------------
TIPO_PERSONA('Ramón Ramirez', TIPO_COCHE('CITROEN', '2-CV', 'M-9999999'))
```

La consulta devuelve el objeto de la clase Tipo_Persona

Gestión de objetos

Al recuperar el objeto completo, de pueden realizar operaciones con él: modificarlo, insertarlo …

```
SQL> SET SERVEROUTPUT ON
DECLARE
  v_persona     Tipo_Persona;
BEGIN
  SELECT VALUE(P) INTO v_persona FROM PERSONAS P WHERE NOMBRE LIKE '%Ram%';
  DBMS_OUTPUT.PUT_LINE(v_persona.nombre);
  DBMS_OUTPUT.PUT_LINE(v_persona.coche.marca);
  DBMS_OUTPUT.PUT_LINE(v_persona.coche.modelo);
END;
/
Ramón Ramirez
CITROEN
2-CV

DECLARE
     v_persona     Tipo_Persona;
BEGIN
    v_persona := Tipo_Persona('Mamen Tido', Tipo_Coche('SEAT', '600', 'B-8888888'));
    INSERT INTO Personas VALUES (v_persona);
END;
/
```

Como hemos creado el objeto antes, no necesitamos invocar el constructor del tipo

Forward Type Definitions

A la hora de crear un tipo sólo podemos referirnos a otros tipos que ya estén creados. Esto representa un problema en caso de referencias circulares

```
CREATE TYPE Empleado AS OBJECT (
  nombre VARCHAR2(20),
  dept REF Departmento,
  ... );

CREATE TYPE Departamento AS OBJECT (
  number INTEGER,
  jefe REF Empleado,
  ... );
```

Para resolverlo se utilizan las Forward Type Definitions:

- ✓ Declarar A, crear B y crear A
- ✓ Recompilar A

```
CREATE TYPE Empleado; -- tipo incompleto

CREATE TYPE Departamento AS OBJECT (
  number INTEGER,
  jefe REF Empleado,
  ... );

CREATE TYPE Empleado AS OBJECT (
  nombre VARCHAR2(20),
  dept REF Departmento,
  ... );
```

Tipos Colección

Oracle soporta dos tipos `VARRAYS`:

- ✓ colección ordenada de elementos de tamaño fijo
- ✓ `NESTED TABLES`: colección no ordenada y de tamaño variable

Como el tipo objeto, incorpora constructores por defecto que hay que utilizar para crear objetos colección

Sus parámetros serán los elementos de la colección

Creación

```
CREATE OR REPLACE TYPE Tipo_Persona AS OBJECT (
    Nombre          VARCHAR2(25),
    DNI             VARCHAR2(10))
/

CREATE OR REPLACE TYPE Tipo_Personas AS
        VARRAY(15) OF Tipo_Persona
/

CREATE OR REPLACE TYPE Tipo_Empresa AS OBJECT (
    Nombre          VARCHAR2(25),
    NIF             VARCHAR2(25),
    Empleados       Tipo_Personas)
/
CREATE TABLE EMPRESAS OF Tipo_Empresa ();
```

```
CREATE OR REPLACE TYPE Tipo_Personas AS
    TABLE OF Tipo_Persona
/
```

```
CREATE TABLE Empresas OF Tipo_Empresa
  NESTED TABLE Empleados STORE AS Lista_Emp;
/
```

Constructor del tipo colección

```
INSERT INTO EMPRESA VALUES ('ACME', '00000000X',
        Tipo_Personas(Tipo_Persona('Pepe Pèrez', '999999999M')));
```

Constructor del tipo objecto

Consulta

La consulta estándar recupera los datos anidados

```
SELECT E.Empleados FROM  Empresa(E) WHERE E.Nombre = 'ACME'

Empleados(Nombre, DNI)
--------------------------------------------------------------------------------
Tipo_Personas(Tipo_Persona('Pepe Pèrez', '999999999M')
```

Devuelve los datos anidados

Mientras que la expresión `TABLE` permite descomponerlos

```
SELECT Emp.* FROM Empresa E, TABLE(E.Empleados) Emp
WHERE E.Nombre = 'ACME'

Nombre DNI
---------- --------------------------
'Pepe Pèrez', '999999999M'
```

Esta sería la forma *intutiitva* de realizar la consulta, pero al ejecutarla obtendríamos un error

```
SELECT E.Emp.* FROM Empresa E
WHERE E.Nombre = 'ACME'
```

Operaciones DML

Operaciones que actúan sobre toda la colección o sobre elementos aislados

Las segundas utilizan el operador `TABLE`

No se soportan actualizaciones individuales en `VARRAYs`

```
-- INSERCIÓN
INSERT INTO TABLE(SELECT E.Empleados FROM  Empresa(E)
                  WHERE E.Nombre = 'ACME')
VALUES ('Mamen Tido', '12345678A')

--ACTUALIZACIÓN
UPDATE TABLE (SELECT E.Empleados FROM  Empresa(E)
                  WHERE E.Nombre = 'ACME')  E
SET VALUE(E) = ('Mamen Tido Mucho', '12345678A')
WHERE E.NOMBRE = 'Mamen Tido';

-- BORRADO
DELETE FROM TABLE(SELECT E.Empleados FROM  Empresa(E)
                  WHERE E.Nombre = 'ACME')  E
WHERE E.NOMBRE = 'Mamen Tido';
```

Operadores

Obtener información sobre la colección

- ✓ `COUNT` devuelve el número de filas.
- ✓ `EXISTS` devuelve `TRUE` si la fila existe.
- ✓ `FIRST/LAST` devuelve el índice de la primera y última fila.
- ✓ `NEXT/PRIOR` devuelve la fila anterior o posterior a la actual.
- ✓ `LIMIT` informa del número máximo de elementos que puede contener .

Modificar los elementos de la colección

- ✓ `DELETE` borra uno o más elementos usando su índice.
- ✓ `EXTEND` añade nuevas filas.
- ✓ `TRIM` elimina filas.

PL/SQL

```
CREATE TYPE Tipo_Nombres_Dep IS VARRAY(7) OF VARCHAR2(30);
/
CREATE TABLE Departamentos (
                region                  VARCHAR2(25),
                nombres_dep             Tipo_Nombres_Dep);

BEGIN
  INSERT INTO Departamentos VALUES ('Europe', Tipo_Nombres_Dep('Shipping','Sales','Finance'));
  INSERT INTO Departamentos VALUES('Americas', Tipo_Nombres_Dep('Sales','Finance','Shipping'));
  INSERT INTO Departamentos VALUES('Asia', Tipo_Nombres_Dep('Finance','Payroll','Shipping','Sales'));
COMMIT;
END;
/

DECLARE
  v_nombres   Tipo_Nombres_Dep := Tipo_Nombres_Dep('Benefits', 'Advertising', '
              Contracting', 'Executive', 'Marketing');
  v_nombres2 Tipo_Nombres_Dep;
BEGIN
  UPDATE Departamentos SET nombres_dep = v_nombres WHERE region = 'Europe';
  COMMIT;
  SELECT nombres_dep INTO v_nombres2 FROM Departamentos WHERE region = 'Europe';
  FOR i IN v_nombres2.FIRST .. V_nombres2.LAST
  LOOP
     DBMS_OUTPUT.PUT_LINE('Departamentos = ' || v_nombres2(i));
  END LOOP;
END;
/
```

Recorre la colección de nombres de departamento

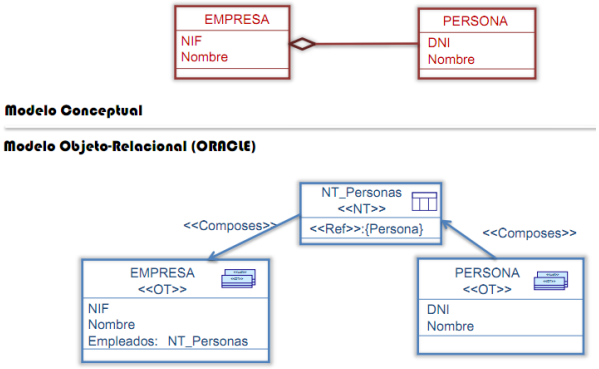
Cursores y colecciones

```
CREATE TYPE Tipo_Nombres_Dep IS VARRAY(7) OF VARCHAR2(30);
/
CREATE TABLE Departamentos (
                region                  VARCHAR2(25),
                nombres_dep             Tipo_Nombres_Dep);

BEGIN
  INSERT INTO Departamentos VALUES ('Europe', Tipo_Nombres_Dep('Shipping','Sales','Finance'));
COMMIT;
END;
/

DECLARE
  CURSOR c_depts IS SELECT * FROM Departamentos;
  v_region      VARCHAR2(25);
  v_nombres   Tipo_Nombres_Dep;
BEGIN
  OPEN c_depts;
  LOOP
    FETCH c_depts INTO v_region, v_nombres;
    EXIT WHEN c_depts%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('REGION: ' || v_region);
    FOR i IN v_nombres2.FIRST .. V_nombres2.LAST
    LOOP
      DBMS_OUTPUT.PUT_LINE(' - Departamento = ' || '(' || i || ') → ' || v_nombres2(i));
    END LOOP;
  END LOOP;
END;
/
```

Carga una fila (región + lista de nombres de departamento) en las dos variables utilizadas

Colecciones de tipo REF

La consulta a colecciones de tipos `REF` crea algunos problemas



Tipos colección

```
CREATE TYPE Tipo_Persona AS OBJECT (
  DNI           VARCHAR2(30),
  Nombre        VARCHAR2(30))
/

CREATE TYPE Tipo_Personas AS TABLE OF REF Tipo_Persona
/

CREATE TYPE Tipo_Persona AS OBJECT (
  NIF           VARCHAR2(30),
  Nombre        VARCHAR2(30),
  Empleados     Tipo_Personas)
/

CREATE TABLE Personas OF Tipo_Persona;

CREATE TABLE Empresas OF Tipo_Empresa
  NESTED TABLE Empleados STORE AS Lista_Emp;
```

```
INSERT INTO Personas VALUES ('0000001', 'Mamen Tido');
INSERT INTO Personas VALUES ('0000002', 'Francisco Rupto');

DECLARE
  v_p1          REF Tipo_Persona;
  v_p2          REF Tipo_Persona;
BEGIN
  SELECT REF(P) INTO v_p1 FROM Personas P
    WHERE P.DNI = '0000001';
  SELECT REF(P) INTO v_p2 FROM Personas P
    WHERE P.DNI = '0000002';
  INSERT INTO Empresas E VALUES ('12345678A', 'ACME',
    Tipo_Personas(vp1, vp2));
END;
/
```

Una alternativa pasa por crear un tipo intermedio que proporcione el nombre columna ...

```
SELECT E.Nombre, EMP.Nombre FROM Empresas E,
TABLE(E.Empleados) EMP;
```

Esta consulta eleva un error, el problema es que el resultado es un OID sin más. Necesitamos un nombre de columna ...

```
SELECT E.Nombre, EMP.COLUMN_VALUE.Nombre
FROM Empresas E, TABLE(E.Empleados) EMP;
```

```
CREATE TYPE Tipo_REF_Persona AS OBJECT (
  Persona       REF Tipo_Persona)
/

CREATE TYPE Tipo_Personas AS TABLE OF Tipo_REF_Persona
/

SELECT E.Nombre, EMP.Persona.Nombre FROM Empresas E,
TABLE(E.Empleados) EMP;
```

Tipos de Objeto en PL/SQL

SQL permite

Capture

Cuando una declaración de tipo de un ámbito diferente impide que el compilador resuelva correctamente un nombre o referencia

Para evitar estos errores, Oracle define una serie de reglas, entre otras:

- ✓ Especificar una alias para cualquier tabla involucrada en una sentencia DML
- ✓ Preceder cualquier nombre de columna con el alias dado a la tabla correspondiente
- ✓ Evitar el uso de alias que coincidan con el nombre del esquema

Resolución de nombres: uso de alias

```
CREATE TYPE Tipo1 AS OBJECT (
        a NUMBER);
CREATE TABLE Tab1 (
        tab2 Tipo1);
CREATE TABLE Tab2 (
        x NUMBER);
SELECT * FROM Tab1 T1  -- alias con el mismo nombre que el esquema
  WHERE EXISTS (SELECT * FROM T1.Tab2 WHERE x = T1.tab2.a);
-- T1.tab2.a se resuelve apuntando al atributo a de la columna tab2 de la tabla Tab1
```

```
CREATE TYPE Tipo1 AS OBJECT (
        a NUMBER);
CREATE TABLE Tab1 (
        tab2 Tipo1);
CREATE TABLE Tab2 (
        x NUMBER,
        a NUMBER);
SELECT * FROM Tabla1 T1  -- alias con el mismo nombre que el esquema
  WHERE EXISTS (SELECT * FROM T1.Tabla2 WHERE x = T1.tab2.a);
-- Ahora T1.tab2.a se resuelve apuntando al atributo a de la la tabla Tab2
```

```
SELECT * FROM Tabla1 tb1
WHERE EXISTS (SELECT * FROM T1.Tabla2 tb2
                    WHERE tb2.x = tb1.tab2.a);
-- Evitamos ambigüedades siguiendo las reglas de nombrado
```

Anexo III - Bases de datos objeto-relacionales

Original en: http://informatica.uv.es/iiguia/DBD/Teoria/capitulo_4.pdf

El término base de datos objeto-relacional se usa para describir una base de datos que ha evolucionado desde el modelo relacional hasta una base de datos híbrida, que contiene ambas tecnologías:

relacional y de objetos.

Durante muchos años ha habido debates sobre cómo sería la siguiente generación de la tecnología de bases de datos de uso común:

- ✓ Las bases de datos orientadas a objetos.
- ✓ Una base de datos basada en SQL con extensiones orientadas a objetos.

Los partidarios de la segunda opción esgrimen varias razones para demostrar que el modelo objeto relacional dominará:

- ✓ Las bases de datos objeto-relacionales tales como Oracle8i son compatibles en sentido ascendente con las bases de datos relacionales actuales y que además son familiares a los usuarios.
  Los usuarios pueden pasar sus aplicaciones actuales sobre bases de datos relaciones al nuevo modelo sin tener que reescribirlas.
  Posteriormente se pueden ir adaptando las aplicaciones y bases de datos para que utilicen las funciones orientadas a objetos.
- ✓ Las primeras bases de datos orientadas a objetos puras no admitían las capacidades estándar de consulta ad hoc de las bases de datos SQL. Esto también hace que resulte problemático realizar la interfaz entre las herramientas SQL estándar y las bases de datos orientadas a objetos puras.

Una de las principales razones por las que las bases de datos relacionales tuvieron un éxito tan rápido fue por su capacidad para crear consultas ad hoc.

Tecnología objeto-relacional

Para ilustrar la tecnología objeto-relacional utilizaremos como ejemplo el modelo que implementa la base de datos Oracle8.

Tipos de objetos

El modelo relacional está diseñado para representar los datos como una serie de tablas con columnas y atributos.

Oracle8 es una base de datos objeto-relacional: incorpora tecnologías orientadas a objetos. En este sentido, permite construir tipos de objetos complejos, entendidos como:

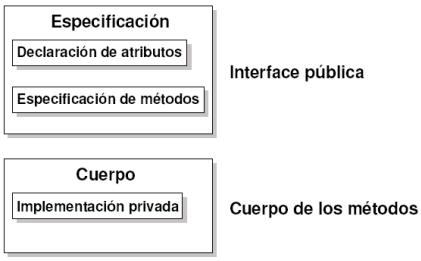
- ✓ Capacidad para definir objetos dentro de objetos.
- ✓ Cierta capacidad para encapsular o asociar métodos con dichos objetos.

Estructura de un tipo de objeto

Un tipo de objeto consta de dos partes: especificación y cuerpo:

- ✓ La **especificación** constituye la interface a las aplicaciones; aquí se declaran las estructuras de datos (conjunto de atributos) y las operaciones (métodos) necesarios para manipular los datos.
- ✓ El **cuerpo** define los métodos, es decir, implementa la especificación.

En la figura se representa la estructura de un tipo de objeto:



Toda la información que un cliente necesita para utilizar los métodos se encuentra en la especificación.
Es conveniente pensar en la especificiación como en la interface operacional y en el cuerpo como en una caja negra. Esto permite depurar, mejorar o reemplazar el cuerpo sin necesidad de modificar la especificación y sin afectar, por tanto, a las aplicaciones cliente.

Características:

En una especificación de tipo de objeto los atributos deben declararse antes que cualquiera de los métodos.

Si una especificación de tipo sólo declara atributos, el cuerpo es innecesario.

Todas las declaraciones en la especificación del tipo son públicas. Sin embargo, el cuerpo puede contener declaraciones privadas, que definen métodos internos del tipo de objeto.

El ámbito de las declaraciones privadas es local al cuerpo del objeto.

Ejemplo:

Un tipo de objeto para manipular números complejos. Un número complejo se representa mediante dos números reales (parte real y parte imaginaria respectivamente) y una serie de operaciones asociadas:

```
CREATE TYPE Complex AS OBJECT (
   rpart REAL,
   ipart REAL,
   MEMBER FUNCTION plus(x Complex) RETURN Complex ,
   MEMBER FUNCTION less(x Complex) RETURN Complex ,
   MEMBER FUNCTION times(x Complex) RETURN Complex ,
   MEMBER FUNCTION divby(x Complex) RETURN Complex
) ;
/
CREATE TYPE BODY Complex AS
   MEMBER FUNCTION plus (x Complex) RETURN Complex IS
   BEGIN
      RETURN Complex(rpart + x.rpat, ipart + x.ipart) ;
   END plus ;
   MEMBER FUNCTION less(x Complex) RETURN Complex IS
   BEGIN
      RETURN Complex(rpart - x.rpart, ipart - x.ipart) ;
   END less;
   MEMBER FUNCTION times(x Complex ) RETURN Complex IS
   BEGIN
      RETURN Complex(rpart * x.rpart - ipart * x.ipart,
      rpart * x.ipart + ipart * x.rpart ) ;
   END times;
   MEMBER FUNCTION divby(x Complex) RETURN Complex IS
      z REAL := x.rpart **2 + x.ipart **2;
   BEGIN
      RETURN Complex ((rpart * x.rpart + ipart * x.ipart) / z ,
         (ipart * x.rpart - rpart * x.ipart) / z) ;
   END divby ;
END;
```

Componentes de un tipo de objeto

Un tipo de objeto encapsula datos y operaciones, por lo que en la especificación sólo se pueden declarar atributos y métodos, pero no constantes, excepciones, cursores o tipos.
Se requiere al menos un atributo y los métodos son opcionales.

Atributos

1. Como las variables, un atributo se declara mediante un nombre y un tipo.
2. El nombre debe ser único dento del tipo de objeto (aunque puede reutilizarse en otros objetos)
3. El tipo puede ser cualquier tipo de Oracle excepto:
   - ✓ `LONG` y `LONG RAW`.

- ✓ `NCHAR`, `NCLOB` y `NVARCHAR2`.
- ✓ `MLSLABEL` y `ROWID`.
- ✓ Los tipos específicos de PL/SQL: `BINARY INTEGER` (y cualquiera de sus subtipos), `BOOLEAN`, `PLS_INTEGER`, `RECORD`, `REF CURSOR`, `%TYPE` y `%ROWTYPE`.
- ✓ Los tipos definidos en los paquetes PL/SQL.

4. Tampoco se puede inicializar un atributo en la declaración empleando el operador de asignación o la cláusula `DEFAULT`.
5. Del mismo modo, no se puede imponer la restricción `NOT NULL`.
6. Sin embargo, los objetos se pueden almancenar en tablas de la base de datos en las que sí es posible imponer restricciones.

Las estructuras de datos pueden llegar a ser muy complejas:
- ✓ El tipo de un atributo puede ser otro tipo de objeto (denominado entonces tipo de objeto anidado).
- ✓ Esto permite construir tipos de objeto complejos a partir de objetos simples.
- ✓ Algunos objetos, tales como colas, listas y árboles son dinámicos (pueden crecer a medida que se utilizan).
- ✓ También es posible definir modelos de datos sofisticados utilizando tipos de objeto recursivos, que contienen referencias directas o indirectas a ellos mismos.

Métodos

Un método es un subprograma declarado en una especificación de tipo mediante la palabra clave `MEMBER`.

El método no puede tener el mismo nombre que el tipo de objeto ni el de ninguno de sus atributos.

Muchos métodos constan de dos partes: especificación y cuerpo.
- ✓ La especificación consiste en el nombre del método, una lista opcional de parámetros y en el caso de funciones un tipo de retorno.
- ✓ El cuerpo es el código que se ejecuta para llevar a cabo una operación específica.

Para cada especificación de método debe existir el cuerpo del método.

El PL/SQL compara la especificación del método y el cuerpo token a token, por lo que las cabeceras deben coincidir.

Los métodos pueden hacer referencia a los atributos y a los otros métodos sin cualificador:

```
CREATE TYPE Stack AS OBJECT (
   top INTEGER,
   MEMBER FUNCTION full RETURN BOOLEAN,
   MEMBER FUNCTION push(n IN INTEGER) ,
   . . .
) ;
/
CREATE TYPE BODY Stack AS
   . . .
   MEMBER FUNCTION push (n IN INTEGER) IS
   BEGIN
      IF NOT full THEN
         top := top + 1 ;
. . .
      END push;
   END;
/
```

El parámetro SELF

Todos los métodos de un tipo de objeto aceptan como primer parámetro una instancia predefinida del mismo tipo denominada `SELF`.

Independientemente de que se declare implicita o explícitamente, `SELF` es siempre el primer parámetro pasado a un método.

Por ejemplo, el método transform declara `SELF` como un parámetro `IN OUT`:

```
CREATE TYPE Complex AS OBJECT (
   MEMBER FUNCTION t rans form (SELF IN OUT Complex ) . . .
```

El modo de acceso de `SELF` cuando no se declara explícitamente es:

- ✓ En funciones miembro el acceso de `SELF` es `IN`.
- ✓ En procedimientos, si `SELF` no se declara, su modo por omisión es `IN OUT`.

En el cuerpo de un método, `SELF` denota al objeto a partir del cual se invocó el método.

Los métodos pueden hacer referencia a los atributos de `SELF` sin necesidad de utilizar un cualificador:

```
CREATE FUNCTION gcd ( x INTEGER, y INTEGER) RETURN INTEGER AS
-- Encuentra e l máximo común d i v i s o r de x e y
   ans INTEGER;
BEGIN
   IF x < y THEN ans := gcd(x , y ) ;
   ELSE ans := gcd (y , x MOD y ) ;
   ENDIF;
   RETURN ans ;
END;
/
CREATE TYPE Rational AS
   num INTEGER,
   den INTEGER,
   MEMBER PROCEDURE normalize ,
   . . .
) ;
/
CREATE TYPE BODY Rational AS
   MEMBER PROCEDURE normalize IS
   g INTEGER;
   BEGIN
      -- Estas dos sentencias son equivalentes
      g := gcd(SELF.num, SELF.den ) ;
      g := gcd(num, den ) ;
      num := num / g ;
      den := den / g ;
   END normalize;
   . . .
END;
/
```

Sobrecarga

Los métodos del mismo tipo (funciones y procedimientos) se pueden sobrecargar: es posible utilizar el mismo nombre para métodos distintos si sus parámetros formales difieren en número, orden o tipo de datos.

Cuando se invoca uno de los métodos, el PL/SQL encuentra el cuerpo adecuado comparando la lista de parámetros actuales con cada una de las listas de parámetros formales.

La operación de sobrecarga no es posible en las siguientes circunstancias:

- ✓ Si los parámetros formales difieren sólo en el modo.
- ✓ Si las funciones sólo difieren en el tipo de retorno.

Métodos MAP y ORDER

Los valores de un tipo escalar, como `CHAR` o `REAL`, tienen un orden predefinido que permite compararlos.

Las instancias de un objeto carecen de un orden predefinido. Para ordenarlas, el PL/SQL invoca a un método de `MAP` definido por el usuario.

En el siguiente ejemplo, la palabra clave `MAP` indica que el método `convert` ordena los objetos `Rational` proyectándolos como números reales:

```
CREATE TYPE Rational AS OBJECT (
   num INTEGER,
   den INTEGER,
   MAP MEMBER FUNCTION convert RETURN REAL,
   . . .
) ;
/
CREATE TYPE BODY Rat ional AS
   MAP MEMBER FUNCTION convert RETURN REAL IS
   -- Convierte un numero racional en un real
   BEGIN
      RETURN num / den ;
   END convert;
   . . .
END;
/
```

El PL/SQL usa esta función para evaluar expresiones booleanas como x >y y para las comparaciones implícitas que requieren las cláusulas `DISTINCT`, `GROUP BY` y `ORDER BY`.

Un tipo de objeto puede contener sólo una función de `MAP`, que debe carecer de parámetros y debe devolver uno de los siguientes tipos escalares: `DATE`, `NUMBER`, `VARCHAR2` y cualquiera de los tipos ANSI SQL (como `CHARACTER` o `REAL`).

Alternativamente, es posible definir un método de ordenación (`ORDER`). Un método `ORDER` utiliza dos parámetros: el parámetro predefinido `SELF` y otro objeto del mismo tipo. En el siguiente ejemplo, la palabra clave `ORDER` indica que el método match compara dos objetos.

Ejemplo:

`c1` y `c2` son objetos del tipo `Customer`. Una comparación del tipo `c1>c2` invoca al método `match` automáticamente. El método devuelve un número negativo, cero o positivo (`SELF` es menor, igual o mayor que el otro parámetro):

```
CREATE TYPE Customer AS OBJECT (
   id NUMBER,
   name VARCHAR2(20) ,
   addr VARCHAR2(30) ,
   ORDER MEMBER FUNCTION match(c Customer ) RETURN INTEGER
) ;
/
CREATE TYPE BODY Customer AS
   ORDER MEMBER FUNCTION match(c Customer ) RETURN INTEGER IS
   BEGIN
      IF id < c.id THEN
         RETURN -1; -- Cualquier número negativo vale.
      ELSEIF id > c.id THEN
         RETURN 1; -- Cualquier número positivo vale.
      ELSE
         RETURN 0;
      END IF;
   END;
END;
/
```

Un tipo de objeto puede contener un único método ORDER, que es una función que devuelve un resultado numérico.

Es importante tener en cuenta los siguientes puntos:
- ✓ Un método `MAP` proyecta el valor de los objetos en valores escalares. Un método `ORDER` compara el valor de un objeto con otro.
- ✓ Se puede declarar un método `MAP` o un método `ORDER`, pero no ambos.
- ✓ Si se declara uno de los dos métodos, es posible comparar objetos en SQL o en un procedimiento. Si no se declara ninguno, sólo es posible comparar la igualdad o desigualdad de dos objetos y sólo en SQL. Dos objeto sólo son iguales si los valores de sus atributos son iguales.
- ✓ Cuando es necesario ordenar un número grande de objetos es mejor utilizar un método `MAP` (ya que una llamada por objeto proporciona una proyección escalar que es más fácil de ordenar). `ORDER` es menos eficiente: debe invocarse repetidamente ya que compara sólo dos objetos cada vez.

Constructores

Cada tipo de objeto tiene un constructor: función definida por el sistema con el mismo nombre que el objeto. Se utiliza para inicializar y devolver una instancia de ese tipo de objeto.

Oracle genera un constructor por omisión para cada tipo de objeto. Los parámetros del constructor coinciden con los atributos del tipo de objeto: los parámetros y los atributos se declaran en el mismo orden y tienen el mismo nombre y tipo.

PL/SQL nunca invoca al constructor implícitamente: el usuario debe invocarlo explícitamente.

Pragma RESTRIC_REFERENCES

Para ejecutar una sentencia SQL que invoca a una función miembro, Oracle debe conocer el nivel de **pureza** de la función: la medida en que la función está libre de **efectos colaterales**.

Los **efectos colaterales** pueden impedir:
- ✓ La paralelización de una consulta.
- ✓ Dar lugar a resultados dependientes del orden (y por tanto indeterminados).
- ✓ Requerir que un módulo mantenga un cierto estado entre diferentes sesiones de usuario.

Una función miembro debe cumplir las siguientes características:
- ✓ No puede insertar, actualizar o borrar las tablas de la base de datos.
- ✓ No se puede ejecutar en paralelo o remotamente si lee o escribe los valores de una variable en un módulo.
- ✓ No puede escribir una variable de un módulo excepto si se invoca desde una cláusula `SELECT`, `VALUES` o `SET`.
- ✓ No puede invocar a otro método o subprograma que rompa alguna de las reglas anteriores. Tampoco puede hacer referencias a una vista que imcumpla estas reglas.
- ✓ La directiva de compilación `PRAGMA_REFERENCES` se utiliza para forzar las reglas anteriores.
- ✓ La sentencia `PRAGMA` indica al compilador PL/SQL que debe denegar a la función miembro el acceso a las tablas de la base de datos, variables de un paquete o ambos.
- ✓ La sentencia pragma se codifica después del método sobre el que actúa en la especificación del tipo de objeto.

Sintáxis:

```
PRAGMA RESTRICT_REFERENCES ({DEFAULT | nombre_método } ,
{RNDS | WNDS | RNPS | WNPS} [,{RNDS | WNDS | RNPS | WNPS} ]...) ;
```

Ejemplo:

La siguiente sentencia pragma impide al método de `MAP convert`:
- ✓ Leer el estado de la base de datos (Read No Database State).
- ✓ Modificar el estado de la base de datos (Write No Database State)
- ✓ Leer el estado de un paquete o módulo (Read No Package State).

✓ Modificar el estado de un paquete (Write No Package State):

```
CREATE TYPE Rational AS OBJECT (
   num INTEGER,
   den INTEGER,
   MAP MEMBER FUNCTION convert RETURN REAL,
      . . .
   PRAGMA RESTRICT REFERENCES ( convert,RNDS,WNDS,RPNS,WNPS)
) ;
/
```

Un método sólo se puede invocar en consultas paralelas si se indican las cuatro limitaciones anteriores.

Si se utiliza la palabra clave `DEFAULT` en lugar del nombre del método, la sentencia pragma se aplica a todas las funciones miembro incluído el constructor definido por el sistema.
Ejemplo:

La siguiente sentencia `pragma` limita a todas las funciones miembro la modificación del estado de la base de datos o de los paquetes:

```
PRAGMA RESTRICT_REFERENCES (DEFAULT, WNDS, WNPS)
```

Es posible declarar un pragma para cada función miembro, que predomina sobre cualquier pragma por omisión definido para el objeto.

Declaración e inicialización de objetos

Una vez que se ha definido un tipo de objeto y se ha instalado en el esquema de la base de datos, es posible usarlo en cualquier bloque PL/SQL.

Las instancias de los objetos se crean en tiempo de ejecución.
Estos objetos siguen las reglas normales de ámbito y de instanciación.

- ✓ En un bloque o subprograma, los objetos locales son instanciados cuando se entra en el bloque o subproprograma y dejan de existir cuando se sale.
- ✓ En un paquete, los objetos se instancian cuando se referencia por primera vez al paquete y dejan de existir cuando finaliza la sesión.

Declaración de objetos

Los tipos de objetos se declaran del mismo modo que cualquier tipo interno.

Ejemplo:

En el bloque que sigue se declara un objeto r de tipo Racional y se invoca al constructor para asignar su valor. La llamada asigna los valores 6 y 8 a los atributos `num` y `den` respectivamente:

```
DECLARE
   r Racional;
   BEGIN
      r := Racional(6,8);
      DBMS_OUTPUT.PUT_LINE(r.num); -- muestra 6
```

También es posible declarar objetos como parámetros formales de funciones y procedimientos, de modo que es posible pasar objetos a los subprogramas almacenados y de un subprograma a otro.

Ejemplos:

1. Aquí se emplea un objeto de tipo `Account` para especificar el tipo de dato de un parámetro formal:

```
DECLARE
```

```
   . . .
   PROCEDURE open_acct(new_acct IN OUT Account) IS ...
```

2. En el siguiente ejemplo se declara una función que devuelve un objeto de tipo `Account`:

```
DECLARE
   . . .
   FUNCTION get_acct(acct_id IN INTEGER)
      RETURN Account IS ...
```

Inicialización de objetos

Hasta que se inicializa un objeto, invocando al constructor para ese tipo de objeto, el objeto se dice que es Atómicamente nulo:
El propio objeto es nulo, no sólo sus atributos.
Un objeto nulo siempre es diferente a cualquier otro objeto. De hecho, la comparación de un objeto nulo con otro objeto siempre resulta NULL.
Si se asigna un objeto con otro objeto atómicamente nulo, el primero se convierte a su vez en un objeto atómicamente nulo (y para poder utilizarlo debe ser reinicializado).
En general, si asignamos el no-valor NULL a un objeto, éste se convierte en atómicamente nulo
Ejemplo:

```
DECLARE
   r Racional ;
BEGIN
   r Racional := Racional (1,2); -- r = 1/2
   r := NULL; -- r atómicamente nulo
   IF r IS NULL THEN ... -- la condición resulta TRUE
```

Una buena práctica de programación consiste en inicializar los objetos en su declaración, como se muestra en el siguiente ejemplo:

```
DECLARE
r Racional := Racional (2,3); -- r = 2/3
```

Objetos sin inicializar en PL/SQL

PL/SQL se comporta del siguiente modo cuando accede a objetosvsin inicializar:

- ✓ Los atributos de un objeto no inicializado se evalúan envcualquier expresión como `NULL`.
- ✓ Intentar asignar valores a los atributos de un objeto sin inicializar provoca la excepción predefinida `ACCESS_INTO_NULL`.
- ✓ La operación de comparación `IS NULL` siempre produce `TRUE` cuando se aplica a un objeto no inicializado o a cualquiera de sus atributos.

Existe por tanto, una sutil diferencia entre objetos nulos y objetos con atributos nulos. El siguiente ejemplo intenta ilustrar esa diferencia:

```
DECLARE
   r Relacional ; -- r es atómicamente nulo
   BEGIN
      IF r IS NULL THEN ... -- TRUE
      IF r.num IS NULL THEN ... -- TRUE
      r := Racional (NULL,NULL) ; -- Inicializar
      r.num = 4 ;    -- Exito: r ya no es atómicamente nulo aunque
                     -- sus atributos son nulos
      r := NULL;     -- r es de nuevo atómicamente nulo
      r.num := 4 ;   -- Provoca la excepción ACCESS INTO NULL
   EXCEPTION
      WHEN ACCESS INTO NULL THEN
         ...
   END;
/
```

La invocación de los métodos de un objeto no inicializado está permitida, pero en este caso:

- ✓ `SELF` toma el valor `NULL`.
- ✓ Cuando los atributos de un objeto no inicializado se pasan como parámetros `IN`, se evalúan como `NULL`.

- ✓ Cuando los atributos de un objeto no inicializado se pasan como parámetros `OUT` o `IN OUT`, se produce una excepción si se intenta asignarles un valor.

Acceso a los atributos

Para acceder o cambiar los valores de un atributo se emplea la notación punto ('.'):

```
DECLARE
   r Racional := Racional (NULL, NULL) ;
   numerador INTEGER;
   denominador INTEGER;
BEGIN
   ...
   denominador := r.den ;
   r.num = numerador ;
```

Los nombres de los atributos pueden encadenarse, lo que permite acceder a los atributos de un tipo de objeto anidado. Por ejemplo, supongamos que definimos los tipos de objeto `Address` y `Student` como sigue:

```
CREATE TYPE Address AS OBJECT (
   street      VARCHAR2(30) ,
   city        VARCHAR2(20) ,
   state       CHAR(2) ,
   zip_code    VARCHAR2(5)
) ;
/
CREATE TYPE Student AS OBJECT (
   name           VARCHAR2(20) ,
   home_address   Address ,
   phone_number   VARCHAR2(10) ,
   status         VARCHAR2(10) ,
   advisor name   VARCHAR2(20) ,
   . . .
) ;
/
```

`zip code` es un atributo de tipo Address
`Address` es el tipo de dato del atributo `home address` del tipo de objeto `Student`.
Si s es un objeto `Student`, para acceder al valor de su `zip code` se emplea la siguiente notación:
`s.home_address.zip_code`

Invocación de constructores y métodos

La invocación de un constructor está permitida en cualquier punto en donde se puede invocar una función.
Como las funciones, un constructor se invoca como parte de una expresión.

```
DECLARE
   r1 Racional := Racional (2,3) ;
   FUNCTION average(x Racional, y Racional)
      RETURN Racional IS
   BEGIN
      ...
   END;
BEGIN
   r1 := average(Racional(3,4), Racional(7,11)) ;
   IF (Racional(5,8) > r1) THEN
      ...
   END IF ;
END;
/
```

Paso de parámetros a un constructor

Cuando se pasan parámetros a un constructor la invocación asigna valores iniciales a los atributos del objeto que se está instanciando.

Es necesario suministrar parámetros para cada uno de los atributos ya que, a diferencia de las constantes y variables, los atributos carecen de la cláusula `DEFAULT`.

También es posible invocar al constructor utilizando la notación con nombre en lugar de la notación posicional, como se muestra:

```
BEGIN
   r := Racional (den => 6, num => 5);
   -- asigna num = 5 y den = 6
```

Invocación de métodos

Los métodos se invocan usando la notación punto.

Ejemplo: invocación del método normaliza que divide los atributos `num` y `den` por el mayor común divisor:

```
DECLARE
   r Racional ;
BEGIN
   r := Racional(6,8);
   r.normaliza;
   DBMS_OUTPUT.PUT_LINE(r.num) ; -- muestra 3
```

Es posible encadenar las llamadas a los métodos:

```
DECLARE
   r Racional := Racional(6,8);
BEGIN
   r.reciproco().normaliza;
   DBMS_OUTPUT.PUT_LINE(r.num); -- muestra 4
```

La ejecución se realiza de izquierda a derecha: primero se invoca la función reciproco y después la función normaliza.

En las sentencias SQL la invocación de un métodos sin parámetros requiere la lista vacia de parámetros: '()'.

En sentencias de procedimiento la lista vacia de parámetros es opcional, excepto cuando se encadenan llamadas: es obligatoria para todas las llamadas excepto la última.

No es posible encadenar invocaciones a métodos adicionales a la derecha de la invocación de un procedimiento (los procedimientos no son parte de una expresión). La siguiente sentencia es ilegal:

```
r.normaliza().reciproco; -- i l e g a l
```

Cuando se encadenan dos llamadas a función, el resultado de la primera función debe ser un objeto que puede ser pasado a la segunda función.

Compartición de objetos

La mayoría de los objetos del mundo real son considerablemente más grandes y complejos que el tipo Relacional. Por ejemplo, consideremos los siguientes tipos de objeto:

```
CREATE TYPE Address AS OBJECT (
   street_address    VARCHAR2( 35 ) ,
   city              VARCHAR2( 15 ) ,
   state             CHAR( 2 ) ,
   zip code          INTEGER
) ;
/
CREATE TYPE Person AS OBJECT (
   first_name        VARCHAR2( 15 ) ,
   last_name         VARCHAR2( 15 ) ,
   birthday          DATE,
   home_address      Address , -- Objeto anidado
   pone_number       VARCHAR2( 1 5 ) ,
   ss_number         INTEGER
```

```
   ...
) ;
/
```

Los objetos de tipo `Address` tienen más del doble de atributos que los del tipo Relacional y los objetos de tipo `Person` todavía tienen más atributos, incluyendo uno de tipo `Address`.

Cuando se utilizan objetos grandes, resulta ineficiente pasar copias de él entre subprogramas. En estas circunstancias tiene más sentido compartir el objeto.

Esto se puede hacer si el objeto cuenta con un identificador de objeto.

Para compartir objetos se utilizan referencias (`refs` de forma abreviada). Una `ref` es un puntero al objeto.

La compartición de objetos proporciona dos ventajas importantes:
- ✓ La información no se duplican innecesariamente.
- ✓ Cuando se actualiza un objeto compartido, el cambio se produce sólo en un lugar y cualquier referencia al objeto puede recuperar los valores actualizados inmediatamente.

En el ejemplo siguiente, obtenemos las ventajas de la compartición definiendo el tipo de objeto Home y creando una tabla que almacena las instancias de ese tipo:

```
CREATE TYPE Home AS OBJECT (
   address     VARCHAR2(35),
   owner       VARCHAR2(25),
   age         INTEGER,
   style       VARCHAR(15),
   floor_plan  BLOB,
   price       REAL(9,2),
   . . .
) ;
/
. . .
CREATE TABLE homes OF Home;
```

Utilización de referencias

Revisando la definición anterior del objeto de tipo `Person`, observamos que podemos diseñar una comunidad que puede compartir la misma casa (`Home`).

```
CREATE TYPE Person AS OBJECT (
   first_name     VARCHAR2(15),
   last_name      VARCHAR2(15),
   birthday       DATE,
   home_address   REF Home , -- Compartido con la familia
   phone number   VARCHAR2( 15 ) ,
   ss_number      INTEGER
   mother         REF Person , -- Miembros de la familia
   father         REF Person ,
   ...
) ;
/
```

Note cómo las referencias entre `Person` y `Homes` y entre `Person` entre sí definen relaciones que se dan en el mundo real.

Es posible declarar referencias como variables, parámetros, campos o atributos.

Se pueden utilizar referencias como parámetros `IN` y `OUT` en funciones y procedimientos.

Pero no es posible navegar a través de referencias.

Ejemplo: un intento ilegal de navegar a través de una referencia a un objeto:

```
DECLARE
   p_ref    REF Person ;
   pone_no VARCHAR2( 1 5 ) ;
BEGIN
   ...
   pone_no = p_ref.phone_number ; -- Ilegal!
```

Para llevar a cabo esta operación es necesario utilizar el operador `DEREF`, a través del cual se puede acceder al objeto.

Limitaciones en la definición de tipos

En la creación de un tipo sólo se puede hacer referencia a objetos que ya existan en el esquema de objetos.

En el siguiente ejemplo la primera sentencia `CREATE TYPE` es ilegal porque hace referencia al objeto de tipo Departament que todavía no existe:

```
CREATE TYPE Employee AS OBJECT (
   name VARCHAR2(20),
   dept REF Departament, -- Ilegal!
   ...
) ;
/
CREATE TYPE Departament AS OBJECT (
   number   INTEGER,
   manager REF Employee,
   ...
) ;
/
```

Cambiar el orden de las sentencias `CREATE TYPE` no soluciona el problema, ya que ambos tipos son mutuamente dependientes.

Para resolver el problema se utiliza una sentencia `CREATE TYPE` especial denominada definición previa de tipo, que permite la creación de tipos de objetos mutuamente dependientes.

```
CREATE TYPE Departament;   -- Definición previa de tipo
                           -- En este punto, Departament es un tipo de objeto incompleto
```

El tipo creado mediante una definición previa de tipo se denomina tipo de objeto incompleto ya que carece de atributos y métodos hasta que se defina en su totalidad.

Un tipo incompleto impuro cuenta con atributos, pero compila con errores semántico (no sintácticos) al hacer referencia a un tipo indefinido. Por ejemplo, la siguiente sentencia `CREATE TYPE` compila con errores debido a que el tipo de objeto Address todavía no está definido:

```
CREATE TYPE Customer AS OBJECT (
   id       NUMBER,
   name     VARCHAR2(20) ,
   addr     Address , -- todavía indefinido
   phone    VARCHAR2(15)
) ;
/
```

Esto permite retrasar la definición del tipo de objeto `Address`.

Más aún, las referencias al tipo incompleto Customer están disponibles para otras aplicaciones.

Manipulación de objetos

Es posible utilizar un tipo de objeto en una sentencia `CREATE TABLE` para especificar el tipo de una columna.

Una vez que la tabla se ha creado, se pueden utilizar las sentencias SQL para insertar un objeto, seleccionar sus atributos, invocar los métodos definidos y actualizar su estado.

Ejemplos:

1. La sentencia `INSERT` invoca al constructor del tipo Racional para insertar su valor. La sentencia `SELECT` recupera el valor del atributo num y la sentencia `UPDATE` invoca al método reciproco, que devuelve un valor Relacional después de invertir los valores de num y den. Observe que se requiere un alias de la tabla cuando se hace referencia a un atributo o método.

```
CREATE TABLE numbers(rn Racional, ...) ;
INSERT INTO numbers(rn) VALUES(Racional(3,62));
SELECT n.rn.num INTO my num FROM numbers n WHERE ...
UPDATE numbers n SET n.rn = n.rn.reciproco WHERE ...
```

   Cuando se crea un objeto de este modo, carece de identidad fuera de la tabla de la base de datos.

2. En el siguiente ejemplo se crea una tabla que almacena en sus filas objetos del tipo Relacional. Este tipo de tablas, cuyas filas contienen un tipo de objetos, se denominan tablas de objetos. Cada columna en una fila se corresponde con un atributo del tipo de objeto:

```
CREATE TABLE racional_nums OF Racional ;
```

Cada fila en una tabla de objetos cuenta con un identificador de objeto, que identifica de forma unívoca al objeto almacenado en dicha fila y sirve como una refencia al objeto.

Selección de objetos

Supongamos que ejecutamos el siguiente script de SQL*Plus, que crea un tipo de objeto denominado `Person` y una tabla de objetos `persons` junto con algunos valores:

```
CREATE TYPE Person AS OBJECT (
   first name     VARCHAR2(15),
   last name      VARCHAR2(15),
   birthday       DATE,
   home_address   Address,
   pone_number    VARCHAR2(15) ,
) ;
/
CREATE TABLE persons OF Person ;
/
```

La siguiente subconsulta produce como resultado un conjunto de filas que contienen sólo atributos de los objetos `Person`:

```
BEGIN
   INSERT INTO employees
      -- emplyees es otra tabla de objetos de tipo Person
      SELECT * FROM persons p
         WHERE p.last_name LIKE '%smith';
```

El operador VALUE

El comando `VALUE` devuelve el valor de un objeto.

`VALUE` requiere como argumento una variable de correlación (en este contexto, variable de correlación es una fila o alias de tabla asociado a una fila en una tabla de objetos).

Ejemplos:

1. Para obtener un conjunto de objetos Person se puede utilizar el comando `VALUES` del siguiente modo:

```
BEGIN
  INSERT INTO employees
     SELECT VALUE(p) FROM persons p
        WHERE p.last_name LIKE '%smith'
```

2. En el siguiente ejemplo, se utiliza el operador `VALUE` para obtener un objeto Person específico:

```
DECLARE
  p1 PERSON;
  p2 PERSON;
BEGIN
  SELECT VALUE(p) INTO p1 FROM persons p
     WHERE p.last name = 'Kroll';
  p2 := p1 ;
  ...
END;
```

   Después de ejecutar la consulta SQL, la variable p1 contiene un objeto Person local, que es una copia del objeto almacenado en la tabla persons. Del mismo modo, p2 contiene otra copia local del objeto.
3. Es posible utilizar las variables anteriores para acceder y modificar el objeto que contienen:

BEGIN

```
p1.last_name := p1.last_name || 'Jr';
```

El operador REF

El operador `REF` se utiliza para obtener una referencia a un objeto.

Como el operador VALUE, toma como argumento una variable de correlación.

Ejemplos:

1. En el siguiente ejemplo, primero se recuperan una o más referencias a objetos de tipo `Person` y despu¶es se insertan en la tabla `person refs`:

```
BEGIN
  INSERT INTO person_refs
     SELECT REF(p) FROM persons p
        WHERE p.last_name LIKE '%smith';
```

2. En el siguiente ejemplo se obtienen simultáneamente una referencia y un atributo:

```
DECLARE
  p ref          REF Person ;
  taxpayer_id    VARCHAR2( 9 ) ;
BEGIN
  SELECT REF(p ), p.ss number INTO p ref, taxpayer id
     FROM persons p
     WHERE p.last name = 'Parker' ; -- sólo una fila
  ...
END;
```

3. En este último ejemplo, se actualizan los atributos de un objeto `Person`:

```
DECLARE
   p_ref          REF Person;
   my_last_name   VARCHAR2(15) ;
   . . .
BEGIN
   ...
   SELECT REF(p) INTO p_ref FROM persons p
      WHERE p.last_name = my_last_name;
   UPDATE persons p
      SET p = Person ('Jill', 'Anders', '11-NOV-67' , ...)
      WHERE REF(p) = p_ref ;
END;
```

Referencias colgadas (dangling refs)

Si el objeto al cual apunta una referencia es borrado, la referencia queda "*colgada*" (*dangling*, apuntando a un objeto inexistente).

Para comprobar si se produce esta condición se puede utilizar el predicado `SQL IS DANGLING`.

Ejemplo:

Supongamos que la columna `manager` en la tabla relacional `department` contiene referencias a objetos `Employee` almacenados en
una tabla de objetos. Para convertir todas las referencias colgadas en nulos, podemos utilizar la siguiente sentencia `UPDATE`:

```
BEGIN
   UPDATE department SET manager = NULL
      WHERE manager IS DANGLING;
```

El operador DEREF

No es posible navegar a través de referencias en procedimientos SQL. Para esto es necesario utilizar el operador `DEREF` (abreviatura del término inglés dereference: derreferenciar un puntero es obtener el valor al cual apunta).

`DEREF` toma como argumento una referencia a un objeto y devuelve el valor de dicho objeto. Si la referencia está colgada, `DEREF` devuelve el valor `NULL`.

Ejemplos:

1. En el ejemplo que sigue se derreferencia una referencia a un objeto `Person`. En estas circunstancias, no es necesario especificar una tabla de objetos ni un criterio de búsqueda ya que cada objeto almacenado en una tabla de objetos cuenta con un identificador de objeto único e inmutable que es parte de cada referencia a un objeto.

```
DECLARE
  p1 Person ;
  p_ref REF Person;
  name VARCHAR2(15);
BEGIN
  ...
  /* Supongamos que p_ref contiene una referencia válida
     a un objeto almacenado en una tabla de objetos */
SELECT DEREF(p_ref) INTO p1 FROM DUAL;
name := p1.last_name ;
```

2. Es posible utilizar el operador `DEREF` en sentencias SQL sucesivas para derreferencias referencias, como se muestra en el siguiente ejemplo:

```
CREATE TYPE PersonRef AS OBJECT (p_ref REF Person);
/
DECLARE
   name     VARCHAR2(15);
   pr_ref   REF PersonRef;
   pr       PersonRef;
   p        Person;
BEGIN
   ...
   /* Supongamos que prref contiene
      una referencia válida */
   SELECT DEREF(pr_ref) INTO pr FROM DUAL;
   SELECT DEREF(pr.p ref) INTO p FROM DUAL;
   name := p.last name ;
   . . .
END;
/
```

3. En procedimientos SQL la utilización del operador DEREF es ilegal. En sentencias SQL se puede utilizar la notación punto para navegar a través de referencias. Por ejemplo, la siguiente sentencia es legal:

```
table_alias.object_column.ref_attribute
table alias.object column.ref attribute.attribute
table_alias.ref_column.attribute
```

4. Supongamos ahora que ejecutamos el siguiente script SQL*Plus que crea los tipos de objeto `Address` y `Person` y la tabla de objetos `persons`:

```
CREATE TYPE Address AS OBJECT (
     street_address    VARCHAR2(35),
     city              VARCHAR2(15),
     state             CHAR(2),
     zip_code          INTEGER
) ;
```

```
/
CREATE TYPE Person AS OBJECT (
    first name     VARCHAR2(15),
    last name      VARCHAR2(15),
    birthday       DATE,
    home_address   Address ,
    phone_number   VARCHAR2(15),
) ;
/
CREATE TABLE persons OF Person;
/
```

El atributo `home address` es una referencia a una columna en la tabla de objetos `persons`, que a su vez contiene referencias a objetos `Address` almacenados en otra tabla indeterminada.

Tras introducir algunos elementos en la tabla, es posible obtener una dirección particular derreferenciando su referencia, como se muestra en el siguiente ejemplo:

```
DECLARE
    addr1 Address,
    addr2 Address,
    ...
BEGIN
    SELECT DEREF(home_address) INTO addr1 FROM persons p
       WHERE p.last_name = 'Derringer';
```

5. Por último, en este ejemplo se navega a través de la columna de referencias `home address` hasta el atributo `street`. En este caso se requiere un alias a la tabla:

```
DECLARE
    my_street VARCHAR2(25),
    ...
BEGIN
    SELECT p.home address.street INTO my street
       FROM persons p
          WHERE p.last_name = 'Lucas';
```

Inserción de objetos

Para añadir objetos a una tabla de objetos se utiliza el comando `UPDATE`.
Ejemplos:

1. Para insertar un objeto `Person` en la tabla de objetos `persons` utilizamos la siguiente línea:

```
BEGIN
  INSERT INTO persons
     VALUES ('Jenifer', 'Lapidus', ...);
```

2. Alternativamente, es posible utilizar el constructor para el objeto de tipo `Person`:

```
BEGIN
  INSERT INTO persons
     VALUES (Person('Albert', 'Brooker', ...));
```

3. En el siguiente ejemplo, se utiliza la cláusula `RETURNING` para almacenar una referencia a `Person` en una variable local. Es importante destacar como esta cláusula simula una sentencia `SELECT`. La cláusula `RETURNING` se puede utilizar también en sentencias `UPDATE` y `DELETE`.

```
DECLARE
  p1_ref REF Person,
  p2 ref REF Person,
  ...
BEGIN
  INSERT INTO persons p
     VALUES(Person('Paul', 'Chang', ...))
     RETURNING REF(p) INTO p1_ref ;
  INSERT INTO persons p
     VALUES (Person('Ana', 'Thorne' ,...))
     RETURNING REF(p) INTO p2_ref ;
```

4. Para insertar objetos en una tabla de objetos se puede utilizar una consulta que devuelva un objeto del mismo tipo, como se muestra en el siguiente ejemplo:

```
BEGIN
  INSERT INTO persons2
     SELECT VALUE(p) FROM persons p
        WHERE p.last_name LIKE '%Jones';
```

Las filas copiadas a la tabla de objetos `persons2` cuentan con identificadores de objeto nuevos, ya que los identificadores de objeto son únicos.

5. El siguiente script crea la tabla relacional `department` que cuenta con una columna de tipo `Person`; después inserta una fila en la tabla. Es importante destacar cómo el constructor `Person()` proporciona un valor para la columna manager:

```
CREATE TABLE department (
   dept_name   VARCHAR2(20),
   manager     Person,
   location    VARCHAR2(20));
/
INSERT INTO department
   VALUES('Payroll',
      Person('Alan', 'Tsai', ...),
      'Los Angeles');
/
```

El nuevo objeto `Persona` almacenado en la columna `manager` no es referenciable, ya que al estar almacenada en una columna (y no en una fila) carece de identificador de objeto.

Actualización de objetos

Para modificar los atributos de un objeto en una tabla de objetos se utiliza la sentencia `UPDATE`
Ejemplo:

```
BEGIN
   UPDATE persons p SET p.home_address = '341 Oakdene Ave'
      WHERE p.last name = 'Brody';
      ...
   UPDATE persons p SET p = Person('Beth', 'Steinberg', ...)
      WHERE p.last_name = 'Steinway';
      . . .
END;
```

Borrado de objetos

Para eliminar objetos (filas) de una tabla de objetos se utiliza la sentencia `DELETE`.
Para eliminar objetos selectivamente se utiliza la cláusula `WHERE`.
Ejemplo:

```
BEGIN
   DELETE FROM persons p
      WHERE p.home address = '108 Palm Dr';
      . . .
END;
```

Anexo IV - Bases de Datos Objeto-Relacionales en Oracle 8

Original en: http://www3.uji.es/~mmarques/e16/teoria/lib_cap9.pdf

1. Introducción

Debido a los requerimientos de las nuevas aplicaciones, en su octava versión, el sistema gestión de bases de datos relacionales Oracle ha sido significativamente extendido con conceptos del modelo de bases de datos orientadas a objetos. De esta manera, aunque las estructuras de datos que se utilizan para almacenar la información siguen siendo tablas, los usuarios pueden utilizar muchos de los mecanismos de orientación a objetos para definir y acceder a los datos. Por esta razón, se dice que se trata de un modelo de datos objetorelacional.

Oracle 8 proporciona mecanismos para que el usuario pueda definir sus propios tipos de datos, cuya estructura puede ser compleja, y que se pueden aplicar para asignar un tipo a una columna de una tabla. También reconoce el concepto de objetos, de tal manera que un objeto tiene un tipo, se almacena en cierta fila de cierta tabla y tiene un identificador único (OID). Estos identificadores se pueden utilizar para referenciar a otros objetos y así representar relaciones de asociación y de agregación. Oracle 8 también proporciona mecanismos para asociar métodos a tipos, y constructores para diseñar tipos de datos multivaluados (colecciones) y tablas anidadas. La mayor deficiencia de este sistema es la imposibilidad de definir jerarquías de especialización y herencia, lo cual es una importante desventaja con respecto a las bases de datos orientadas a objetos.

2. Tipos de Datos Definidos por el Usuario

Los usuarios de Oracle 8 pueden definir sus propios tipos de datos, pudiendo ser de dos categorías: tipos de objetos (`object types`) y tipos para colecciones (`collection types`). Para construir los tipos de usuario se utilizan los tipos básicos provistos por el sistema y otros tipos de usuario previamente definidos. Un tipo define una estructura y un comportamiento común para un conjunto de datos de las aplicaciones.

2.1 Tipos de objetos

Un tipo de objetos representa a una entidad del mundo real. Un tipo de objetos se compone de los siguientes elementos:

- ✓ Un nombre que sirve para identificar el tipo de los objetos.
- ✓ Unos atributos que modelan la estructura y los valores de los datos de ese tipo. Cada atributo puede ser de un tipo de datos básico o de un tipo de usuario.
- ✓ Unos métodos que son procedimientos o funciones escritos en el lenguaje PL/SQL (almacenados en la base de datos), o escritos en C (almacenados externamente).

Los tipos de objetos pueden interpretarse como plantillas a las que se adaptan los objetos de ese tipo. A continuación se da un ejemplo de cómo definir el tipo de datos `Direccion_T` en el lenguaje de definición de datos de Oracle 8, y como utilizar este tipo de datos para definir el tipo de datos de los objetos de la clase de `Clientes_T`.

DEFINICIÓN ORIENTADA A OBJETOS

```
define type Direccion_T:
   tuple [calle:string,
   ciudad:string,
   prov:string,
   codpos:string]

define class Clientes_T
   type tuple [clinum: integer,
      clinomb:string,
      direccion:Direccion_T,
      telefono: string,
      fecha-nac:date]
   operations edad():integer
```

DEFINICIÓN EN ORACLE

```
CREATE TYPE direccion_t AS OBJECT (
   calle VARCHAR2(200),
   ciudad VARCHAR2(200),
   prov CHAR(2),
   codpos VARCHAR2(20) ) ;

CREATE TYPE clientes_t AS OBJECT (
   clinum NUMBER,
   clinomb VARCHAR2(200),
   direccion direccion_t,
   telefono VARCHAR2(20),
   fecha_nac DATE,
MEMBER FUNCTION edad RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(edad,WNDS)
) ;
```

2.2 Métodos

La especificación de un método se hace junto con la creación de su tipo, y debe llevar siempre asociada una directiva de compilación (`PRAGMA RESTRICT REFERENCES`), para evitar que los métodos manipulen la base de datos o las variables del paquete PL/SQL. Tienen el siguiente significado:

- ✓ WNDS: no se permite al método modificar las tablas de la base de datos
- ✓ WNPS: no se permite al método modificar las variables del paquete PL/SQL
- ✓ RNDS: no se permite al método leer las tablas de la base de datos
- ✓ RNPS: no se permite al método leer las variables del paquete PL/SQL

Los métodos se pueden ejecutar sobre los objetos de su mismo tipo. Si `x` es una variable PL/SQL que almacena objetos del tipo `Clientes T`, entonces `x.edad()` calcula la edad del cliente almacenado en `x`. La definición del cuerpo de un método en PL/SQL se hace de la siguiente manera:

```
CREATE OR REPLACE TYPE BODY clientes_t AS
   MEMBER FUNCTION edad RETURN NUMBER IS
      a NUMBER;
      d DATE;
   BEGIN
      d:= today();
      a:= d.año - fecha nac.año;
      IF (d.mes < fecha_nac.mes) OR
         ((d.mes = fecha_nac.mes) AND (d.dia < fecha_nac.dia))
         THEN a:= a+1;
      END IF;
      RETURN a;
   END;
END;
```

2.2.1 Constructores de tipo

En Oracle, todos los tipos de objetos tienen asociado por defecto un método que construye nuevos objetos de ese tipo de acuerdo a la especificación del tipo. El nombre del método coincide con el

nombre del tipo, y sus parámetros son los atributos del tipo. Por ejemplo las siguientes expresiones construyen dos objetos con todos sus valores.

```
direccion_t('Avenida Sagunto', 'Puzol', 'Valencia', 'E-23523')
clientes_t( 2347,
   'José Pérez Ruíz',
   direccion_t('Calle Eo', 'Onda', 'Castellón', '34568'),
   '696-779789',
   12/12/1981
   )
```

2.2.2 Métodos de comparación

Para comparar los objetos de cierto tipo es necesario indicar a Oracle cuál es el criterio de comparación. Para hacer esto hay que escoger entre un método `MAP` o `ORDER`, debiéndose definir al menos uno de estos métodos por cada tipo de objetos que necesiten ser comparados. La diferencia entre ellos es la siguiente:

- ✓ Un método de `MAP` sirve para indicar cuál de los atributos del tipo se va a utilizar para ordenar los objetos del tipo, y por lo tanto se puede utilizar para comparar los objetos de ese tipo por medio de los operadores de comparación típicos (**<**, **>**). Por ejemplo la siguiente declaración permite decir que los objetos del tipo `clientes_t` se van a comparar por su atributo clinum.

```
CREATE TYPE clientes t AS OBJECT (
   clinum NUMBER,
   clinomb VARCHAR2(200),
   direccion direccion_t,
   telefono VARCHAR2(20),
   fecha nac DATE,
MAP MEMBER FUNCTION ret value RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(
   ret_value, WNDS, WNPS, RNPS, RNDS),/*instrucciones a PL/SQL*/
MEMBER FUNCTION edad RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(edad, WNDS)
   ) ;
CREATE OR REPLACE TYPE BODY clientes t AS
   MAP MEMBER FUNCTION ret_value RETURN NUMBER IS
      BEGIN
         RETURN clinum
      END;
   END;
```

- ✓ Un método `ORDER` utiliza los atributos del objeto sobre el que se ejecuta para realizar un cálculo y compararlo con otro objeto del mismo tipo que toma como argumento de entrada. Este método debe devolver un valor negativo si el primero es mayor que el segundo, un valor positivo si ocurre lo contrario y un cero si ambos son iguales. El siguiente ejemplo define un orden para el tipo `clientes t` diferente al anterior. Solo una de estas definiciones puede ser válida a un tiempo.

```
CREATE TYPE clientes_t AS OBJECT (
   clinum NUMBER,
   clinomb VARCHAR2(200),
   direccion direccion_t,
   telefono VARCHAR2(20),
   fecha_nac DATE,
ORDER MEMBER FUNCTION
   cli ordenados (x IN clientes t) RETURN INTEGER,
PRAGMA RESTRICT_REFERENCES(
   cli_ordenados, WNDS, WNPS, RNPS, RNDS),
MEMBER FUNCTION edad RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(edad, WNDS)
   ) ;
CREATE OR REPLACE TYPE BODY clientes t AS
   ORDER MEMBER FUNCTION cli_ordenados (x IN clientes_t)
      RETURN INTEGER IS
      BEGIN
         RETURN clinum - x.clinum; /*la resta de los dos números clinum*/
      END;
   END;
```

Para un tipo de objetos que no tenga definido ninguno de estos métodos, Oracle es incapaz de deducir cuándo un objeto es mayor o menor que otro. Sin embargo sí que puede determinar cuándo dos objetos del mismo tipo son iguales. Para hacer esto, el sistema compara el valor de los atributos de los objetos uno a uno:

- ✓ Si todos los atributos son no nulos e iguales, Oracle indica que ambos objetos son iguales.
- ✓ Si alguno de los atributos no nulos es distinto en los dos objetos, entonces Oracle dice que son diferentes.
- ✓ En otro caso, Oracle dice que no puede comparar ambos objetos.

2.3 Tablas de objetos

Después de definir los tipos, éstos pueden utilizarse para definir otros tipos, tablas que almacenen objetos de esos tipos, o para definir el tipo de los atributos de una tabla. Una tabla de objetos es una clase especial de tabla que almacena un objeto en cada fila y que facilita el acceso a los atributos de esos objetos como si fueran columnas de la tabla. Por ejemplo, se puede definir una tabla para almacenar los clientes de este año y otra para almacenar los de años anteriores de la siguiente manera:

```
CREATE TABLE clientes_año_tab OF clientes_t
   (clinum PRIMARY KEY);
CREATE TABLE clientes_antiguos_tab (
   año NUMBER,
   cliente clientes t
   ) ;
```

La diferencia entre la primera y la segunda tabla es que la primera almacena objetos con su propia identidad (OID) y la segunda no es una tabla de objetos, sino una tabla con una columna con un tipo de datos de objetos. Es decir, la segunda tabla tiene una columna con un tipo de datos complejo pero sin identidad de objeto. Además de esto, Oracle permite considerar una tabla de objetos desde dos puntos de vista:

- ✓ Como una tabla con una sola columna cuyo tipo es el de un tipo de objetos.
- ✓ Como una tabla que tiene tantas columnas como atributos los objetos que almacena.

Por ejemplo, se puede ejecutar una de las dos instrucciones siguientes. En la primera instrucción, la tabla `clientes año tab` se considera como una tabla con varias columnas cuyos valores son los especificados. En el segundo caso se la considera como con una tabla de objetos que en cada fila almacena un objeto. En esta instrucción la cláusula `VALUE` permite visualizar el valor de un objeto.

```
INSERT INTO clientes_año_tab VALUES(
2347,
'José Pérez Ruíz',
direccion_t('Calle Castalia', 'Onda', 'Castellón', '34568'),
'696-779789',
12/12/1981
);
SELECT VALUE(c) FROM clientes_año_tab c
WHERE c.clinomb = 'José Pérez Ruíz'
```

Las reglas de integridad, de clave primaria, y el resto de propiedades que se definan sobre una tabla, sólo afectan a los objetos de esa tabla, es decir no se refieren a todos los objetos del tipo asignado a la tabla.

2.4 Referencias entre objetos

Los identificadores únicos asignados por Oracle a los objetos que se almacenan en una tabla, permiten que éstos puedan ser referenciados desde los atributos de otros objetos o desde las columnas de tablas. El tipo de datos proporcionado por Oracle para soportar esta facilidad se denomina `REF`. Un atributo de tipo `REF` almacena una referencia a un objeto del tipo definido, e implementa una relación de asociación entre los dos tipos de objetos. Estas referencias se pueden utilizar para acceder a los objetos referenciados y para modificarlos, sin embargo no es posible

operar sobre ellas directamente. Para asignar o actualizar una referencia se debe utilizar siempre `REF` o `NULL`.

Cuando se define una columna de un tipo a `REF`, es posible restringir su dominio a los objetos que se almacenen en cierta tabla. Si la referencia no se asocia a una tabla sino que sólo se restringe a un tipo de objetos, se podrá actualizar a una referencia a un objeto del tipo adecuado independientemente de la tabla donde se almacene. En este caso su almacenamiento requerirá más espacio y su acceso será menos eficiente. El siguiente ejemplo define un atributo de tipo `REF` y restringe su dominio a los objetos de cierta tabla.

```
CREATE TABLE clientes_tab OF clientes_t;
CREATE TYPE ordenes t AS OBJECT (
   ordnum NUMBER,
   cliente REF clientes_t,
   fechpedido DATE,
   direcentrega direccion t
);
CREATE TABLE ordenes_tab OF ordenes_t (
   PRIMARY KEY (ordnum),
   SCOPE FOR (cliente) IS clientes_tab
);
```

Cuando se borran objetos de la base de datos, puede ocurrir que otros objetos que referencien a los borrados queden en un estado inconsistente. Estas referencias se denominan *dangling references*, y Oracle proporciona un predicado que permite comprobar cuando sucede esto. El predicado se denomina `IS DANGLING`.

2.5 Tipos para colecciones

Para poder implementar relaciones `1:N`, en Oracle 8 es posible definir tipos para colecciones. Un dato de tipo colección está formado por un número indefinido de elementos, todos del mismo tipo. De esta manera en un atributo es posible almacenar un conjunto de tuplas en forma de array (`VARRAY`), o en forma de tabla anidada.

Al igual que los tipos para objetos, los tipos para colecciones también tienen por defecto unas funciones constructoras de colecciones cuyo nombre coincide con el del tipo.

Los argumentos de entrada de estas funciones son el conjunto de elementos que forman la colección separados por comas y entre paréntesis, y el resultado es un valor del tipo colección. En Oracle es posible diferenciar entre un valor nulo y una colección vacía. Para construir una colección sin elementos se puede utilizar la función constructora del tipo seguida por dos paréntesis sin elementos dentro.

2.5.1 El tipo VARRAY

Un array es un conjunto ordenado de elementos del mismo tipo. Cada elemento tiene asociado un índice que indica su posición dentro del array. Oracle permite que los VARRAY sean de longitud variable, aunque es necesario especificar un tamaño máximo cuando se declara el tipo VARRAY. Las siguientes declaraciones crean un tipo para una lista ordenada de precios, y un valor para dicho tipo.

```
CREATE TYPE precios AS VARRAY(10) OF NUMBER(12);
precios('35', '342', '3970');
```

Un tipo `VARRAY` se puede utilizar para:

- ✓ Definir el tipo de datos de una columna de una tabla relacional.
- ✓ Definir el tipo de datos de un atributo de un tipo de objetos.
- ✓ Para definir una variable PL/SQL, un parámetro, o el tipo que devuelve una función.

Cuando se declara un tipo `VARRAY` no se produce ninguna reserva de espacio. Si el espacio que requiere lo permite, se almacena junto con el resto de columnas de su tabla, pero si es demasiado largo (más de 4000 bytes) se almacena aparte de la tabla como un `BLOB`.
En el siguiente ejemplo, se quiere definir un tipo de datos para almacenar una lista ordenada de teléfonos (tipo `list`, en el tipo `set` no existe orden). Este tipo se utiliza después para asignárselo a un atributo del tipo de objetos `clientes_t`.

DEFINICIÓN ORIENTADA A OBJETOS	DEFINICIÓN EN ORACLE

`define type Lista_Tel_T: list(string);`
 	
`define class Clientes_T:`
`  tuple [clinum: integer,`
`  clinomb:string,`
`  direccion:Direccion T,`
`  lista_tel: Lista_Tel_T];`
	`CREATE TYPE lista_tel_t AS`
	`  VARRAY(10) OF VARCHAR2(20) ;`
	`CREATE TYPE clientes_t AS OBJECT (`
	`  clinum NUMBER,`
	`  clinomb VARCHAR2(200),`
	`  direccion direccion t,`
	`  lista_tel lista_tel_t );`

La principal limitación que presenta los tipos `VARRAY` es que en las consultas es imposible poner condiciones sobre los elementos almacenados dentro. Desde una consulta SQL, los valores de un `VARRAY` sólamente pueden ser accedidos y recuperados en un bloque. Es decir no se puede acceder a los elementos de un `VARRAY` individualmente. Sin embargo desde un programa PL/SQL si que es posible definir un bucle que itere sobre los elementos de un `VARRAY` (ver sección 4.2.5).

2.5.2 Tablas anidadas

Una tabla anidada es un conjunto de elementos del mismo tipo sin ningún orden predefinido.
Estas tablas solamente pueden tener una columna que puede ser de un tipo de datos básico de Oracle, o de un tipo de objetos definido por el usuario. En este último caso, la tabla anidada también puede ser considerada como una tabla con tantas columnas como atributos tenga el tipo de objetos.
En el siguiente ejemplo, se declara una tabla que después es anidada en el tipo `ordenes_t`. Los pasos de todo el diseño son los siguientes.

1- Se define el tipo de objetos linea_t para las filas de la tabla anidada.

```
define type Linea_T:
  tuple [linum:integer,
  item:string,
  cantidad:integer,
  descuento:real];
```

```
CREATE TYPE linea t AS OBJECT (
  linum NUMBER,
  item VARCHAR2(30),
  cantidad NUMBER,
  descuento NUMBER(6,2)
  ) ;
```

2- Se define el tipo de colección tabla lineas_pedido_t para después anidarla.

```
CREATE TYPE lineas_pedido_t AS TABLE OF linea_t ;
```

Esta definición permite utilizar el tipo colección `lineas_pedido_t` para:

- ✓ Definir el tipo de datos de una columna de una tabla relacional.
- ✓ Definir el tipo de datos de un atributo de un tipo de objetos.
- ✓ Para definir una variable PL/SQL, un parámetro, o el tipo que devuelve una función.

3- Se define el tipo de objetos `ordenes t` que en el atributo pedido almacena una tabla anidada del tipo `lineas pedido t`.

```
define class Ordenes T:
   tuple [ordnum:integer,
   cliente:Clientes T,
   fechpedido:date,
   fechentrega:date,
   pedido:set(Linea_T),
   direcentrega:Direccion_T];
```

```
CREATE TYPE ordenes_t AS OBJECT (
   ordnum NUMBER,
   cliente REF clientes_t,
   fechpedido DATE,
   fechentrega DATE,
   pedido lineas pedido t,
   direcentrega direccion_t
) ;
```

4- Se define la tabla de objetos `ordenes_tab` y se especifica la tabla anidada del tipo `lineas_pedido_t`.

```
CREATE TABLE ordenes_tab OF ordenes_t
   (ordnum PRIMARY KEY,
   SCOPE FOR (cliente) IS clientes_tab)
   NESTED TABLE pedido STORE AS pedidos_tab ;
```

Este último paso es necesario hacerlo porque la declaración de una tabla anidada no reserva ningún espacio para su almacenamiento. Lo que se hace es indicar en qué tabla (`pedidos tab`) se deben almacenar todas las lineas de pedido que se representen en el atributo pedido de cualquier objeto de la tabla `ordenes_tab`. Es decir, todas las líneas de pedido de todas las órdenes se almacenan externamente a la tabla de órdenes, en otra tabla especial. Para relacionar las tuplas de una tabla anidada con la tupla a la que pertenecen se utiliza una columna oculta que aparece en la tabla anidada por defecto. Todas las tuplas de una tabla anidada que pertenecen a la misma tupla tienen el mismo valor en esta columna (`NESTED TABLE ID`).

Al contrario que los VARRAY, los elementos de las tablas anidadas si pueden ser accedidos individualmente, y es posible poner condiciones de recuperación sobre ellos.

Como veremos en la próxima sección, una forma conveniente de acceder individualmente a los elementos de una tabla anidada es por medio de un cursor anidado. Además, las tablas anidadas pueden ser indexadas.

3. Inserción y Acceso a los Datos

3.1 Alias

En una base de datos con tipos y objetos, lo más recomendable es utilizar siempre alias para los nombres de las tablas. El alias de una tabla debe ser único en el contexto de una consulta. Los alias se utilizan para acceder al contenido de las tablas, pero hay que tener cuidado de utilizarlos adecuadamente en las tablas que almacenan objetos. El siguiente ejemplo ilustra cómo se deben utilizar.

```
CREATE TYPE persona AS OBJECT (nombre VARCHAR(20));
CREATE TABLE ptab1 OF persona;
CREATE TABLE ptab2 (c1 persona);
CREATE TABLE ptab3 (c1 REF persona);
```

La diferencia entre las dos tablas está en que la primera almacena objetos del tipo persona, mientras que la segunda tabla tiene una columna donde se almacenan valores del tipo persona. Considerando ahora las siguientes consultas, se ve cómo se puede acceder a estas tablas.

1. `SELECT nombre FROM ptab1;` **BIEN**
2. `SELECT c1.nombre FROM ptab2;` **MAL**
3. `SELECT p.c1.nombre FROM ptab2 p;` **BIEN**
4. `SELECT p.c1.nombre FROM ptab3 p;` **BIEN**
5. `SELECT p.nombre FROM ptab3 p;` **MAL**

En la primera consulta nombre es considerado como una de las columnas de la tabla ptab1, ya que los atributos de los objetos se consideran columnas de la tabla de objetos. Sin embargo, en la segunda consulta se requiere la utilización de un alias para indicar que nombre es el nombre de un atributo del objeto de tipo persona que se almacena en la columna c1. Para resolver este problema no es posible utilizar los nombres de las tablas directamente: ptab2.c1.nombre es incorrecto. Las consultas 4 y 5 muestran cómo acceder a los atributos de los objetps referenciados desde un atributo de la tabla ptab3.

En conclusión, para facilitar la formulación de consultas y evitar errores se recomienda utilizar alias para acceder a todas las tablas que contengan objetos con o sin identidad, y para acceder a las columnas de las tablas en general.

3.2 Inserción de referencias

La inserción de objetos con referencias implica la utilización del operador REF para poder insertar la referencia en el atributo adecuado. La siguiente instrucción inserta una orden en la tabla definida en la sección 2.4.

```
INSERT INTO ordenes_tab
   SELECT 3001, REF(C),'30-MAY-1999',NULL
      --se seleccionan los valores de los 4 atributos de la tabla
   FROM clientes_tab C WHERE C.clinum= 3 ;
```

El acceso a un objeto desde una referencia REF requiere derefenciar al objeto primero. Para realizar esta operación, Oracle proporciona el operador DEREF. No obstante, utilizando la notación de punto también se consigue dereferenciar a un objeto de forma implícita.

Observemos el siguiente ejemplo.

```
CREATE TYPE persona_t AS OBJECT (
   nombre VARCHAR2(30),
   jefe REF persona_t ) ;
```

Si x es una variable que representa a un objeto de tipo persona_t, entonces las dos expresiones siguientes son equivalentes:

```
1. x.jefe.nombre
2. y.nombre, y=DEREF(x.jefe)
```

Para obtener una referencia a un objeto de una tabla de objetos se puede aplicar el operador REF de la manera que se muestra en el siguiente ejemplo:

```
CREATE TABLE persona_tab OF persona_t;
DECLARE ref_persona REF persona_t;
SELECT REF(pe) INTO ref_persona
   FROM persona_tab pe WHERE pe.nombre= 'José Pérez Ruíz';
```

Simétricamente, para recuperar un objeto desde una referencia es necesario usar DEREF, como se muestra en este ejemplo que visualiza los datos del jefe de la persona indicada:

```
SELECT DEREF(pe.jefe)
FROM persona_tab pe WHERE pe.nombre= 'José Pérez Ruíz';
```

3.3 Llamadas a métodos

Para invocar un método hay que utilizar su nombre y unos paréntesis que encierren sus argumentos de entrada. Si el método no tienen argumentos hay que especificar los paréntesis aunque estén

vacíos. Por ejemplo, si tb es una tabla con la columna c de tipo de objetos t, y t tiene un método m sin argumentos de entrada, la siguiente consulta es correcta:

```
SELECT p.c.m( ) FROM tb p;
```

3.4 Inserción en tablas anidadas

Además del constructor del tipo de colección disponible por defecto, la inserción de elementos dentro de una tabla anidada puede hacerse siguiendo estas dos etapas:

1. Crear el objeto con la tabla anidada y dejar el campo que contiene las tuplas anidadas vacío.
2. Comenzar a insertar tuplas en la columna correspondiente de la tupla seleccionada por una subconsulta. Para ello se tiene que utilizar la palabra clave `THE` con la siguiente sintaxis.

```
INSERT INTO THE (subconsulta) (tuplas a insertar)
```

Esta técnica es especialmente útil cuando dentro de una tabla anidada se guardan referencias a otros objetos. En el siguiente ejemplo se ilustra la manera de realizar estas operaciones sobre la tabla de ordenes (`ordenes_tab`) definida en la sección 2.5.2.

```
INSERT INTO ordenes_tab --inserta una orden
   SELECT 3001, REF(C),
          SYSDATE,'30-MAY-1999',
          lineas_pedido_t(),
          NULL
      FROM clientes_tab C
      WHERE C.clinum= 3 ;
INSERT INTO THE ( --selecciona el atributo pedido de la orden
   SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 3001
   )
   SELECT 30, REF(S), 18, 30 --inserta una linea de pedido anidada
      FROM items_tab S
      WHERE S.itemnum = 3011;
```

Para poner condiciones sobre las tuplas de una tabla anidada, se pueden utilizar cursores dentro de un **SELECT** o desde un programa PL/SQL de la manera explicada en la sección 4.2.5. Veamos aquí un ejemplo de acceso con cursores. Utilizando el ejemplo de la sección 2.5.2, vamos a recuperar el número de las ordenes, sus fechas de pedido y las líneas de pedido que se refieran al item 'CH4P3'.

```
SELECT ord.ordnum, ord.fechpedido,
   CURSOR (SELECT * FROM TABLE(ord.pedido) lp WHERE lp.item= 'CH4P3')
FROM ordenes_tab ord;
```

La cláusula **THE** también sirve para seleccionar las tuplas de una tabla anidada. La sintaxis es como sigue:

```
SELECT ... FROM THE (subconsulta) WHERE ...
```

Por ejemplo, para seleccionar las primeras dos líneas de pedido de la orden 8778 se hace:

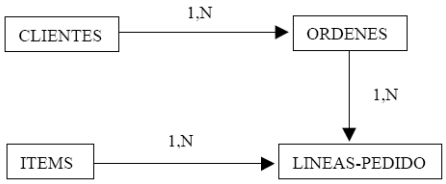
```
SELECT lp FROM THE
(SELECT ord.pedido FROM ordenes_tab ord WHERE ord.ordnum= 8778) lp
WHERE lp.linum<3;
```

4. Una Base de Datos Ejemplo

Partiendo de una base de datos para gestionar los pedidos de los clientes, veamos como se puede proporcionar una solución relacional y otra objeto_relacional en Oracle 8.

4.1 Modelo lógico para una base de datos relacional

```
CLIENTES(clinum, clinomb, calle, ciudad, prov, codpos, tel1, tel2, tel3)
ORDENES(ordnum, clinum, fechpedido, fechentrega, callent, ciuent, provent, codpent)
ITEMS(numitem, precio, tasas)
LINEAS(linum, ordnum, numitem, cantidad, descuento)
```



4.1.1 Implementación relacional con Oracle 8

Se crean tablas normalizadas y con claves ajenas para representar las relaciones.

```
CREATE TABLE clientes (
   clinum NUMBER,
   clinomb VARCHAR2(200),
   calle VARCHAR2(200),
   ciudad VARCHAR2(200),
   prov CHAR(2),
   codpos VARCHAR2(20),
   tel1 VARCHAR2(20),
   tel2 VARCHAR2(20),
   tel3 VARCHAR2(20),
   PRIMARY KEY (clinum)
) ;
CREATE TABLE ordenes (
   ordnum NUMBER,
   clinum NUMBER REFERENCES clientes,
   fechpedido DATE,
   fechaentrega DATE,
   callent VARCHAR2(200),
   ciuent VARCHAR2(200),
   provent CHAR(2),
   codpent VARCHAR2(20),
   PRIMARY KEY (ordnum)
) ;
CREATE TABLE items (
   numitem NUMBER PRIMARY KEY,
   precio NUMBER,
   tasas NUMBER
) ;
CREATE TABLE lineas (
   linum NUMBER,
   ordnum NUMBER REFERENCES ordenes,
   numitem NUMBER REFERENCES items,
   cantidad NUMBER,
   descuento NUMBER,
   PRIMARY KEY (ordnum, linum)
) ;
```

4.2 Modelo lógico para una base de datos orientada a objetos

Primero vamos utilizar el lenguaje de definición de bases de datos orientadas a objetos visto en el tema 5 para definir el esquema de la base de datos que después crearemos en Oracle 8.

```
define type Lista_Tel_T: type list(string);
define type Direccion_T: type tuple [ calle:string,
   ciudad:string,
   prov:string,
   codpos:string];
define class Clientes_T: type tuple [ clinum: integer,
   clinomb:string,
   direccion:Direccion_T,
   lista_tel: Lista_Tel_T];
define class Item_T: type tuple [ itemnum:integer,
   precio:real,
   tasas:real];
define type Linea_T: type tuple [linum:integer,
   item:Item_T,
   cantidad:integer,
   descuento:real];
define type Lineas_Pedido_T: type set(Linea_T);
define class Ordenes_T: type tuple [ ordnum:integer,
   cliente:Clientes T,
   fechpedido:date,
   fechentrega:date,
   pedido:Lineas Pedido T
   direcentrega:Direccion_T];
```

4.2.1 Implementación objeto-relacional con Oracle 8

Aquí se indica como definir todos los tipos anteriores en Oracle 8.

```
CREATE TYPE lista_tel_t AS VARRAY(10) OF VARCHAR2(20) ;
CREATE TYPE direccion_t AS OBJECT (
   calle VARCHAR2(200),
   ciudad VARCHAR2(200),
   prov CHAR(2),
   codpos VARCHAR2(20)
) ;
CREATE TYPE clientes_t AS OBJECT (
   clinum NUMBER,
   clinomb VARCHAR2(200),
   direccion direccion_t,
   lista_tel lista_tel_t,
) ;
CREATE TYPE item_t AS OBJECT (
   itemnum NUMBER,
   precio NUMBER,
   tasas NUMBER
) ;
CREATE TYPE linea_t AS OBJECT (
   linum NUMBER,
   item REF item_t,
   cantidad NUMBER,
   descuento NUMBER
) ;
CREATE TYPE lineas_pedido_t AS TABLE OF linea_t ;
CREATE TYPE ordenes_t AS OBJECT (
   ordnum NUMBER,
   cliente REF clientes_t,
   fechpedido DATE,
   fechentrega DATE,
   pedido lineas_pedido_t,
   direcentrega direccion_t,
) ;
```

4.2.2 Creación de tablas de objetos

Ahora vamos a crear las tablas donde almacenar los objetos de la aplicación.

```
CREATE TABLE clientes_tab OF clientes_t
   (clinum PRIMARY KEY);
CREATE TABLE items_tab OF item_t
   (itemnum PRIMARY KEY) ;
CREATE TABLE ordenes_tab OF ordenes_t (
   PRIMARY KEY (ordnum),
   SCOPE FOR (cliente) IS clientes_tab
)
NESTED TABLE pedido STORE AS pedidos_tab ;
ALTER TABLE pedidos_tab
   ADD (SCOPE FOR (item) IS items_tab) ;
```

Esta última declaración sirve para restringir el dominio de los objetos referenciados desde `item` a aquellos que se almacenan en la tabla `items_tab`.

4.2.3 Inserción de objetos en las tablas

```
REM inserción en la tabla ITEMS_TAB**********************************
INSERT INTO items_tab VALUES(1004, 6750.00, 2);
INSERT INTO items_tab VALUES(1011, 4500.23, 2);
INSERT INTO items_tab VALUES(1534, 2234.00, 2);
INSERT INTO items_tab VALUES(1535, 3456.23, 2);
INSERT INTO items_tab VALUES(2004, 33750.00, 3);
INSERT INTO items_tab VALUES(3011, 43500.23, 4);
INSERT INTO items_tab VALUES(4534, 5034.00, 6);
INSERT INTO items_tab VALUES(5535, 34456.23, 5);
REM inserción en la tabla CLIENTES_TAB*******************************
```

Nótese como en estas definiciones se utilizan los constructores del tipo de objeto `direccion_t` y el tipo de colección `lista_tel_t`.

```
INSERT INTO clientes_tab
```

```
   VALUES (
      1, 'Lola Caro',
      direccion t('12 Calle Lisboa', 'Nules', 'CS', '12678'),
      lista tel t('415-555-1212')
   ) ;
INSERT INTO clientes_tab
   VALUES (
      2, 'Jorge Luz',
      direccion_t('323 Calle Sol', 'Valencia', 'V', '08820'),
      lista tel t('609-555-1212','201-555-1212')
   ) ;
INSERT INTO clientes_tab
   VALUES (
      3, 'Jose Perez',
      direccion t('12 Calle Colon', 'Castellon', 'ES', '12001'),
      lista tel t('964-555-1212', '609-543-1212',
      '201-775-1212','964-445-1212')
   ) ;
INSERT INTO clientes_tab
   VALUES (
      4, 'Ana Gil',
      direccion t('5 Calle Sueca', 'Burriana', 'ES', '12345'),
      lista_tel_t()
   ) ;
REM inserción en la tabla ORDENES_TAB*********************************
```

Nótese como en estas definiciones se utiliza el operador `REF` para obtener una referencia a un objeto de `clientes tab` y almacenarlo en la columna de otro objeto de `ordenes tab`.

La palabra clave `THE` se utiliza para designar la columna de las tuplas que cumplen la condición del `WHERE`, donde se deben realizar la inserción. Las tuplas que se insertan son las designadas por el segundo `SELECT`, y el objeto de la orden debe existir antes de comenzar a insertar líneas de pedido.

```
REM Ordenes del cliente 1*********************************
   INSERT INTO ordenes_tab
      SELECT 1001, REF(C),
         SYSDATE,'10-MAY-1999',
         lineas_pedido_t(),
         NULL
      FROM clientes_tab C
      WHERE C.clinum= 1 ;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 1001
   )
      SELECT 01, REF(S), 12, 0
      FROM items_tab S
      WHERE S.itemnum = 1534;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 1001
   )
      SELECT 02, REF(S), 10, 10
      FROM items_tab S
      WHERE S.itemnum = 1535;
REM Ordenes del cliente 2*********************************
   INSERT INTO ordenes_tab
      SELECT 2001, REF(C),
         SYSDATE,'20-MAY-1999',
         lineas_pedido_t(),
         direccion_t('55 Madison Ave','Madison','WI','53715')
      FROM clientes_tab C
      WHERE C.clinum= 2;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 2001
   )
      SELECT 10, REF(S), 1, 0
      FROM items_tab S
      WHERE S.itemnum = 1004;
```

```
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum= 2001
   )
   VALUES( linea_t(11, NULL, 2, 1) ) ;
REM Ordenes del cliente 3**********************************
   INSERT INTO ordenes_tab
      SELECT 3001, REF(C),
         SYSDATE,'30-MAY-1999',
         lineas_pedido_t(),
         NULL
      FROM clientes_tab C
      WHERE C.clinum= 3 ;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 3001
   )
      SELECT 30, REF(S), 18, 30
      FROM items_tab S
      WHERE S.itemnum = 3011;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 3001
   )
      SELECT 32, REF(S), 10, 100
      FROM items_tab S
      WHERE S.itemnum = 1535;
**************************************
   INSERT INTO ordenes_tab
      SELECT 3002, REF(C),
         SYSDATE,'15-JUN-1999',
         lineas_pedido_t(),
         NULL
      FROM clientes_tab C
      WHERE C.clinum= 3 ;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 3002
   )
      SELECT 34, REF(S), 200, 10
      FROM items tab S
      WHERE S.itemnum = 4534;
REM Ordenes del cliente 4**********************************
   INSERT INTO ordenes_tab
      SELECT 4001, REF(C),
         SYSDATE,'12-MAY-1999',
         lineas_pedido_t(),
         direccion_t('34 Nave Oeste','Nules','CS','12876')
      FROM clientes_tab C
      WHERE C.clinum= 4;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 4001
   )
      SELECT 41, REF(S), 10, 10
      FROM items_tab S
      WHERE S.itemnum = 2004;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 4001
   )
      SELECT 42, REF(S), 32, 22
      FROM items_tab S
      WHERE S.itemnum = 5535;
```

4.2.4 Borrado de los objetos, las tablas y los tipos de usuario

```
DELETE FROM ordenes_tab;
DROP TABLE ordenes_tab;
DELETE FROM clientes_tab;
DROP TABLE clientes_tab;
```

```
DELETE FROM items_tab;
DROP TABLE items_tab;
DROP TYPE ordenes_t;
DROP TYPE lineas_pedido_t;
DROP TYPE linea_t;
DROP TYPE item_t;
DROP TYPE clientes_t;
DROP TYPE lista_tel_t;
DROP TYPE direccion_t;
```

4.2.5 Definición de métodos para los tipos

El siguiente método calcula la suma de los valores de las líneas de pedido de la orden de pedido sobre la que se ejecuta.

```
CREATE TYPE ordenes_t AS OBJECT (
   ordnum NUMBER,
   cliente REF clientes_t,
   fechpedido DATE,
   fechentrega DATE,
   pedido lineas_pedido_t,
   direcentrega direccion_t,
   MEMBER FUNCTION
      coste_total RETURN NUMBER,
      PRAGMA RESTRICT_REFERENCES(coste_total, WNDS, WNPS) );
   CREATE TYPE BODY ordenes_t AS
      MEMBER FUNCTION coste_total RETURN NUMBER IS
         i INTEGER;
         item item_t;
         linea linea_t;
         total NUMBER:=0;
   BEGIN
      FOR i IN 1..SELF.pedido.COUNT LOOP
         linea:=SELF.pedido(i);
         SELECT DEREF(linea.item) INTO item FROM DUAL;
         total:=total + linea.cantidad * item.precio;
      END LOOP;
      RETURN total;
   END;
END;
```

La palabra clave `SELF` permite referirse al objeto sobre el que se ejecuta el método.

La palabra clave `COUNT` sirve para contar el número de elementos de una tabla o de un array. Junto con la instrucción `LOOP` permite iterar sobre los elementos de una colección, en nuestro caso las líneas de pedido de una orden.

El `SELECT` es necesario porque Oracle no permite utilizar `DEREF` directamente en el código PL/SQL.

4.2.6 Consultas a la base de datos anterior

1- Consultar la definición de la tabla de clientes.

```
describe clientes_tab;
```

```
NAME                           NULL? TYPE
------------------------------ -------- ----
CLINUM                         NOT NULL NUMBER
CLINOMB                                 VARCHAR2(200)
DIRECCION                               DIRECCION_T
LISTA_TEL                               LISTA_TEL_T
```

2- Insertar en la tabla de clientes a un nuevo cliente con todos sus datos.

```
insert into clientes_tab
   values(5, 'John smith',
   direccion_t('67 rue de percebe', 'Gijon', 'AS', '73477'),
   lista_tel_t('7477921749', '83797597827'));
```

```
1 ROW CREATED.
```

3- Consultar y modificar el nombre del cliente número 2.

```
select clinomb from clientes_tab where clinum=2;
```

```
CLINOMB
--------------------------------------------------------------------
Jorge Luz
UNIVERSITAT JAUME I
158
```

```
update clientes_tab
   set clinomb='Pepe Puig' where clinum=5;
```

```
1 row updated.
```

4- Consultar y modificar la dirección del cliente número 2.

```
select direccion from clientes_tab where clinum=2;
```

```
DIRECCION(CALLE, CIUDAD, PROV, CODPOS)
--------------------------------------------------------------------
DIRECCION_T('Calle Sol', 'Valencia', 'VA', '08820')
```

```
update clientes_tab
   set direccion=direccion_t('Calle Luna','Castello','CS',68734')
   where clinum=2;
```

```
1 row updated.
```

5- Consultar todos los datos del cliente número 1 y añadir un nuevo teléfono a su lista de teléfonos.

```
select * from clientes_tab where clinum=1;
```

```
CLINUM
----------
CLINOMB
--------------------------------------------------------------------
DIRECCION(CALLE, CIUDAD, PROV, CODPOS)
--------------------------------------------------------------------
LISTA_TEL
--------------------------------------------------------------------
1
Lola Caro
DIRECCION_T('Calle Luna', 'Castellon', 'CS', '64827')
LISTA_TEL_T('415-555-1212')
```

También se puede consultar así:

```
select value(C) from clientes_tab C where C.clinum=1;
```

```
VALUE(C)(CLINUM, CLINOMB, DIRECCION(CALLE, CIUDAD, PROV, CODPOS),
LISTA_TEL)
--------------------------------------------------------------------
CLIENTES_T(1, 'Lola Caro', DIRECCION_T('Calle Luna', 'Castellon', 'CS',
'64827'), LISTA_TEL_T('415-555-1212'))
```

```
update clientes_tab
   set lista_tel=lista_tel_t('415-555-1212', '6348635872')
   where clinum=1;
```

```
1 row updated.
```

6- Visualizar el nombre del cliente que ha realizado la orden número 1001.

```
select o.cliente.clinomb from ordenes_tab o where o.ordnum=1001;
```

```
CLIENTE.CLINOMB
--------------------------------------------------------------------
Lola Caro
DISEÑO DE SISTEMAS DE BASES DE DATOS
159
```

7- Visualizar todos los detalles del cliente que ha realizado la orden número 1001.

```
select DEREF(o.cliente) from ordenes_tab o where o.ordnum=1001;
```

```
DEREF(O.CLIENTE)(CLINUM, CLINOMB, DIRECCION(CALLE, CIUDAD, PROV, CODPOS),
LISTA_TEL
```

```
--------------------------------------------------------------------
CLIENTES_T(1, 'Lola Caro', DIRECCION_T('Calle Luna', 'Castellon', 'CS',
'64827'), LISTA_TEL_T('415-555-1212', '6348635872'))
```

De la siguiente manera se obtiene la referencia al objeto, la cuál es ininteligible.

```
select o.cliente from ordenes_tab o where o.ordnum=1001;
CLIENTE
--------------------------------------------------------------------
00002EA5F6693E4A73F8E003960286473F83EA5F6693E3E73F8E0039680286473F8
```

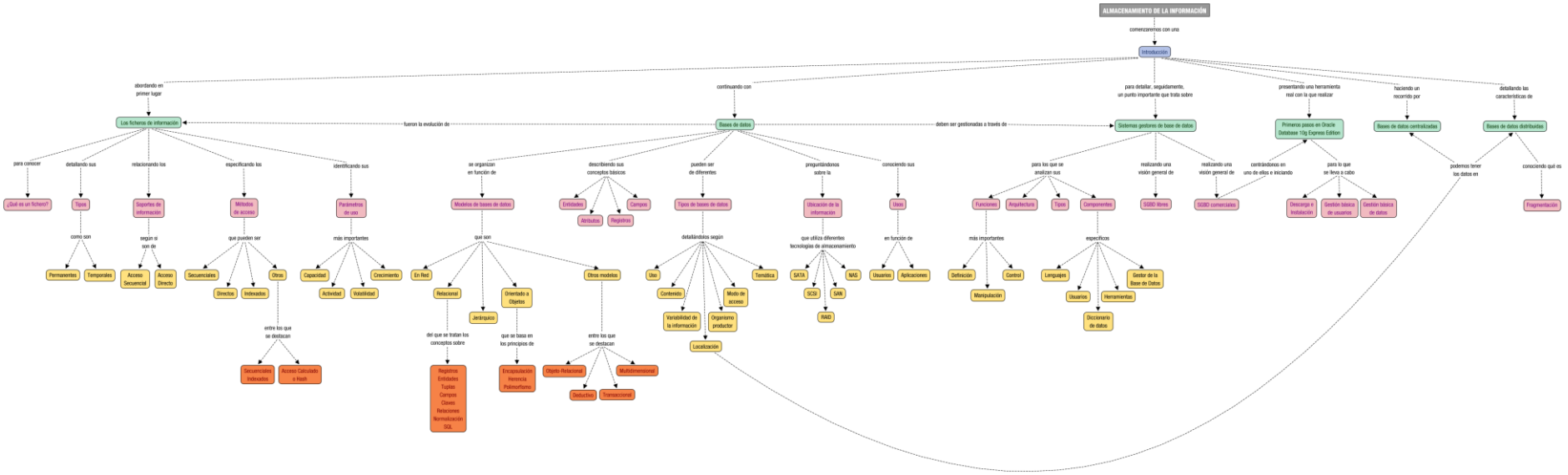
8- Visualizar el número de todos los items que se han pedido en la orden número 3001.

```
select cursor(select p.item.itemnum from table(o.pedido) p)
   from ordenes_tab o where o.ordnum=3001;
CURSOR(SELECTP.ITEM.
--------------------
CURSOR STATEMENT : 1
CURSOR STATEMENT : 1
ITEM.ITEMNUM
------------
3011
1535
```

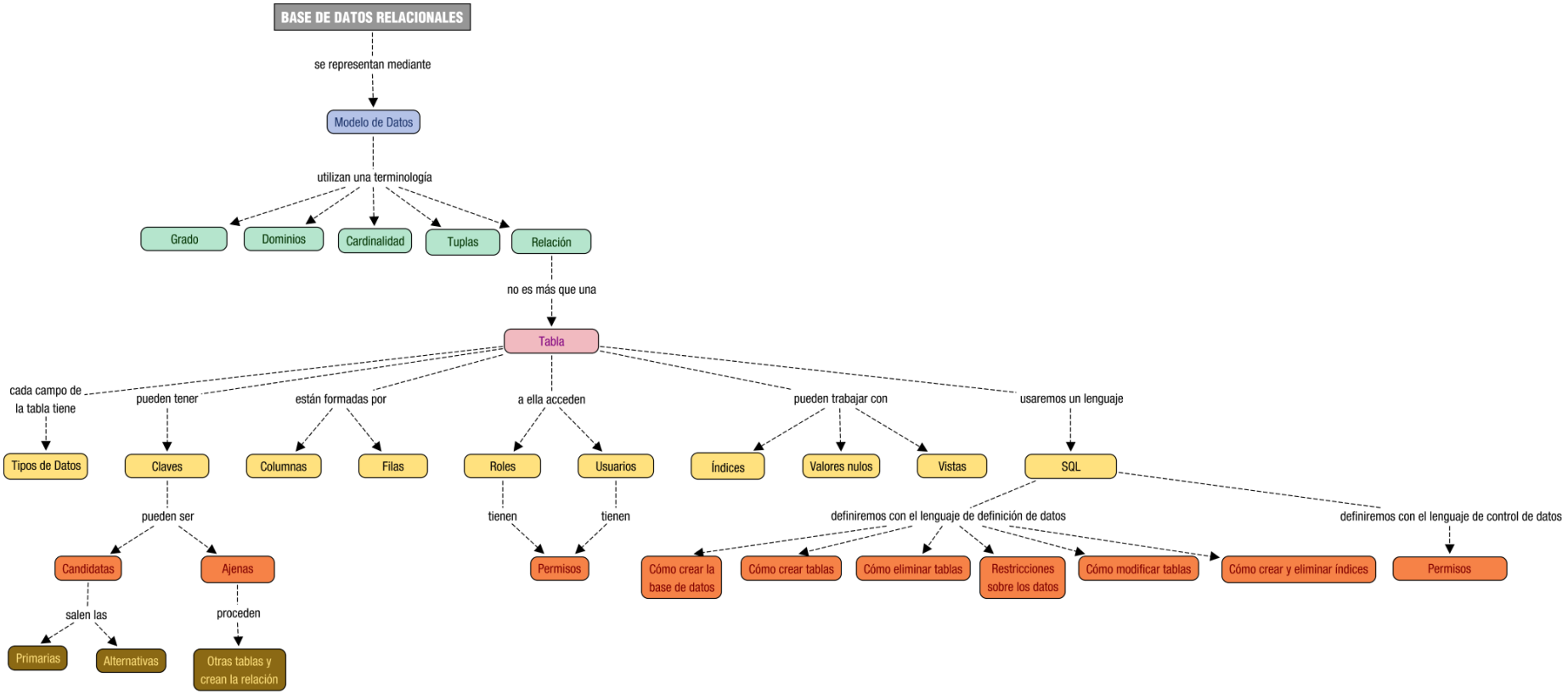
9- Seleccionar el número de orden y el coste total de las ordenes hechas por el cliente número 3.

```
select o.ordnum, o.coste_total() from ordenes_tab o
   where o.cliente.clinum=3;
ORDNUM O.COSTE_TOTAL()
---------- ----------------
3001 817566.44
3002 1006800
```

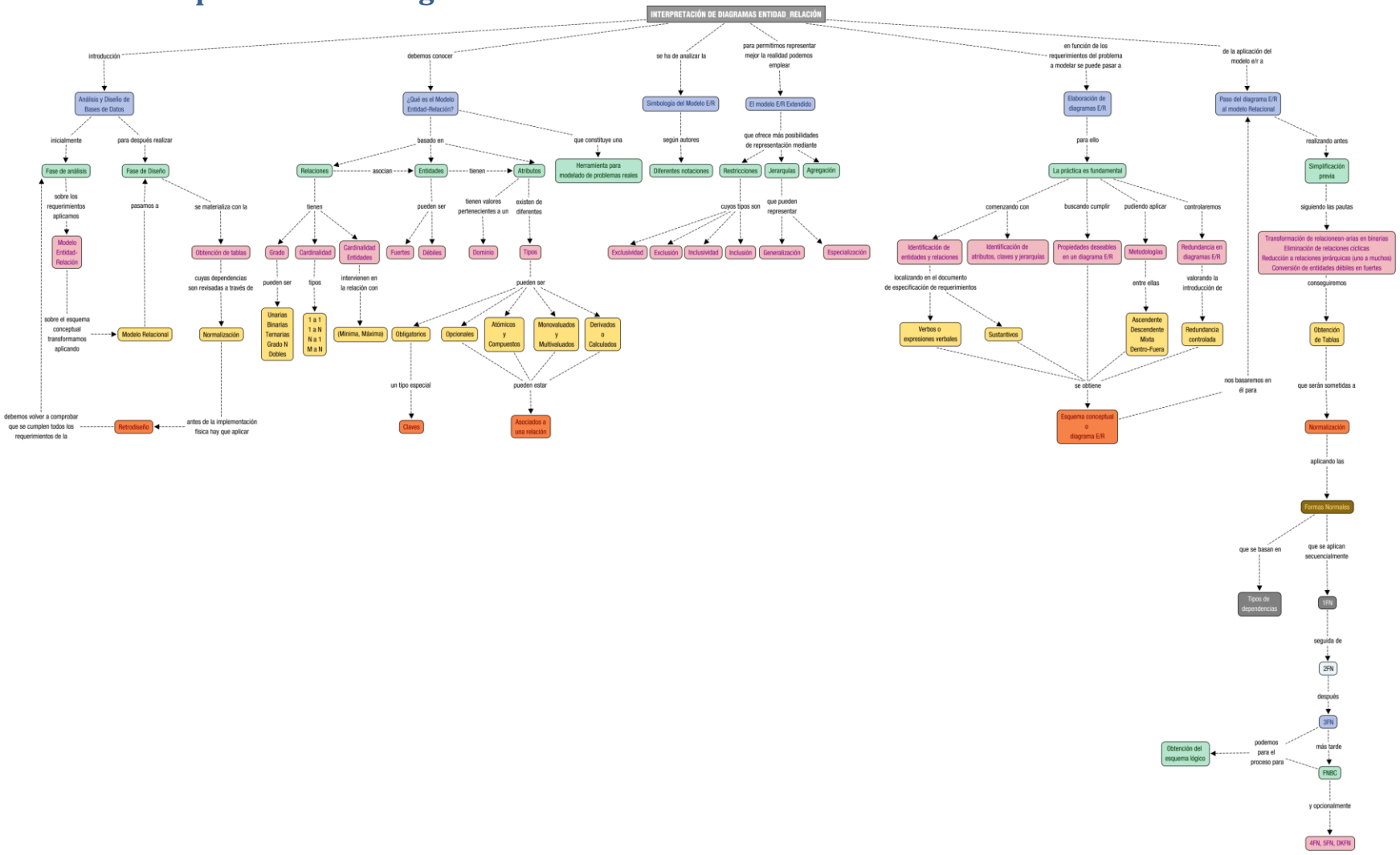
Tema 1 - Almacenamiento de la información



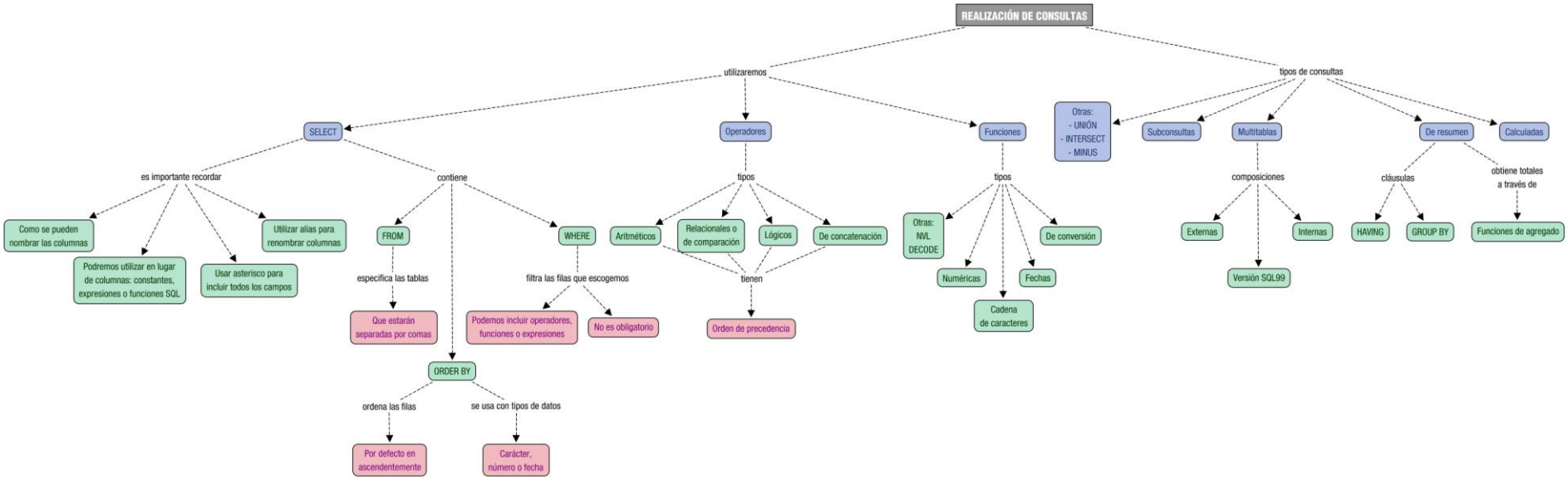
Tema 2 - Bases de datos relacionales



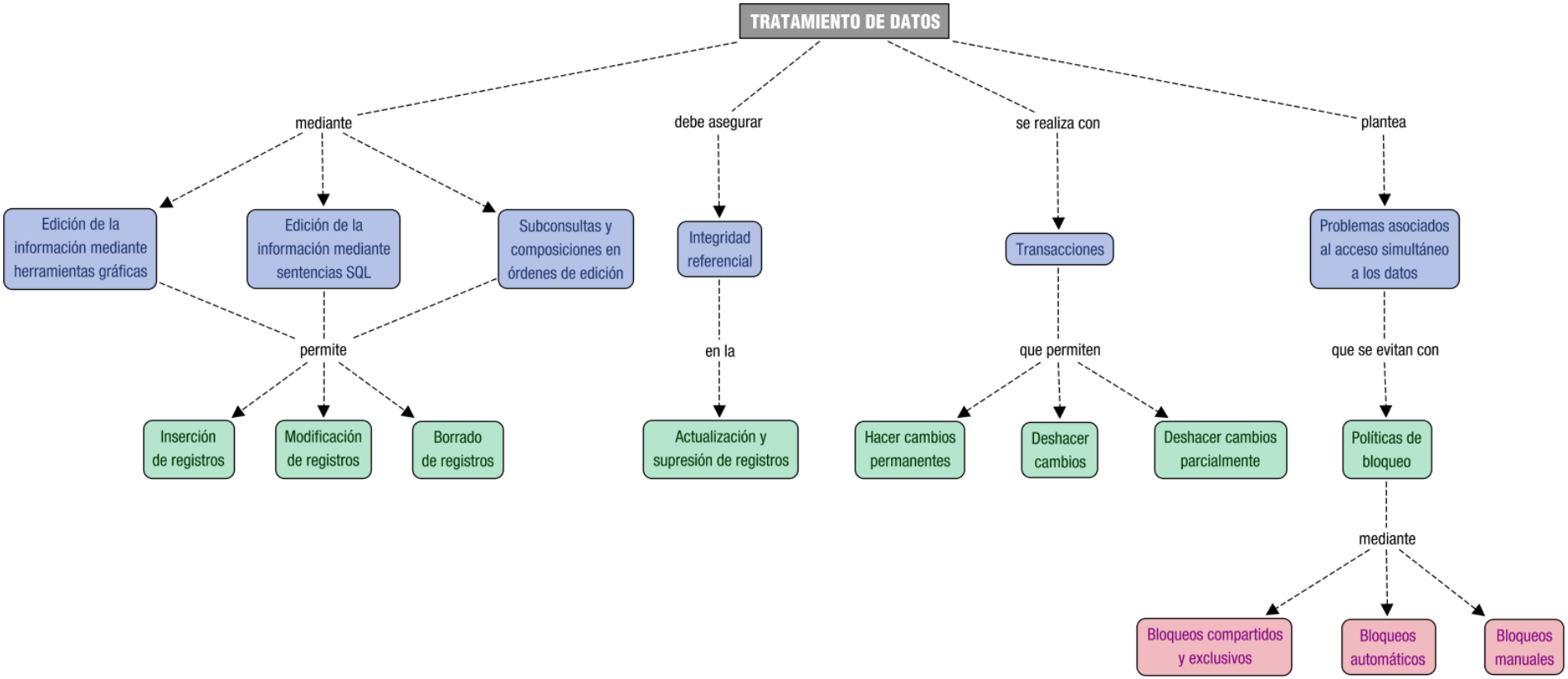
Tema 3 - Interpretación de diagramas Entidad - Relación



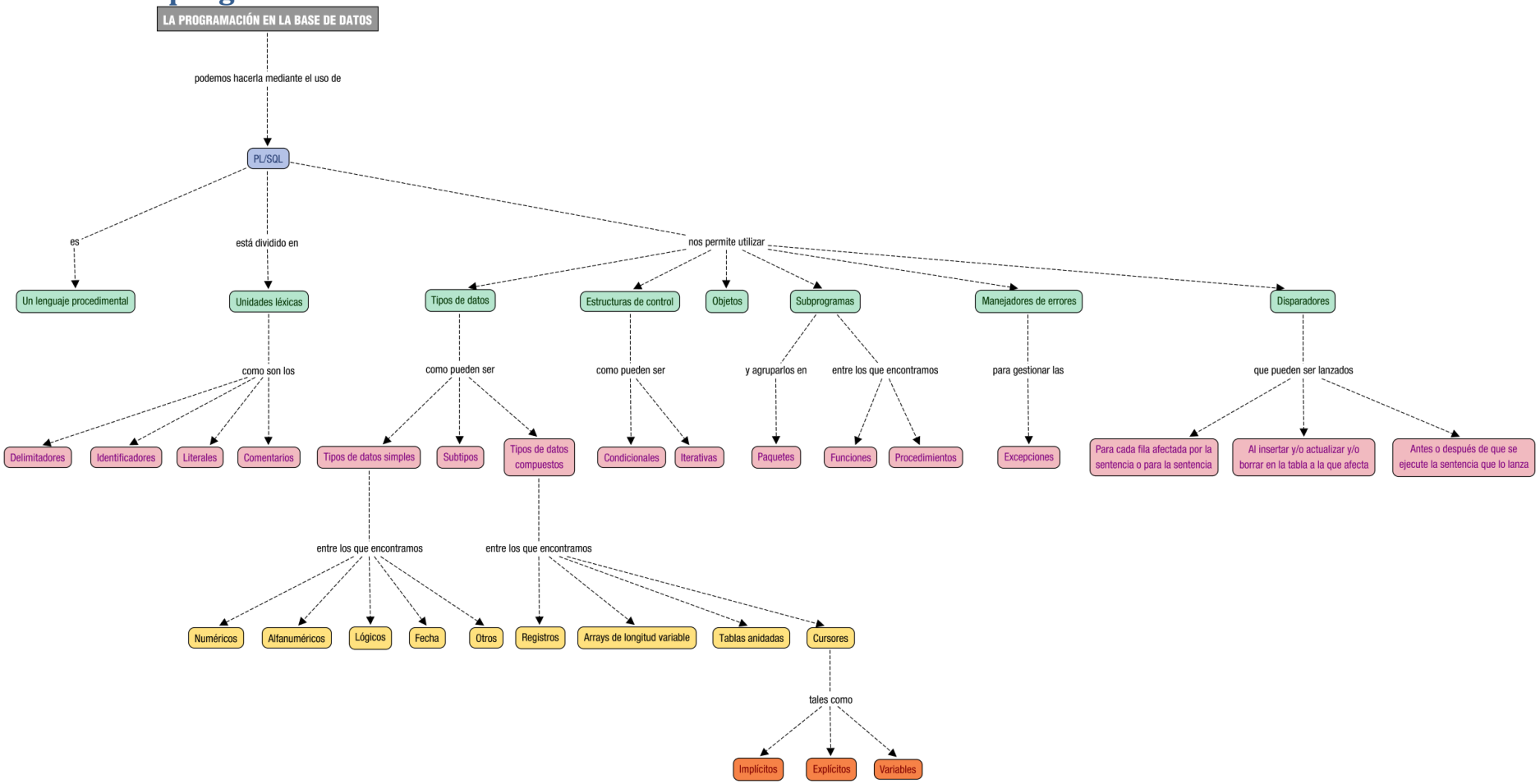
Tema 4 - Realización de consultas



Tema 5 - Tratamiento de datos



Tema 6 - La programación en la base de datos



Tema 7 - Bases de datos objeto-relacionales