



1. Almacenamiento de la información

Class	Bases de datos
Column	(X) Xerach Casanova
Last Edited time	@Mar 29, 2021 9:21 PM

1. Introducción

2. Los ficheros de información

 2.1 ¿Qué es un fichero?

 2.2. Tipos de ficheros

 2.3. Los soportes de información

 2.4. Métodos de acceso

 2.5. Ficheros secuenciales

 2.6. Ficheros de acceso directo

2.6.1. Ficheros de acceso directo

2.6.2. Ficheros indexados

 2.7. Otros métodos de acceso

 2.7.1. Secuenciales indexados

 2.7.2. Ficheros de acceso calculado o hash

 2.8. Parámetros de utilización

3. Bases de datos

 3.2. Conceptos

 3.2. Usos

 3.2. Ubicación de la información

4. Modelos de bases de datos

 4.1. Modelo jerárquico (1^a generación)

 4.2. Modelo en red (1^a generación)

 4.3. Modelo relacional (2^a generación)

 4.4. Modelo orientado a objetos (3^a generación)

 4.5. Otros modelos

 Modelo objeto-relacional (BDOR)

 Modelo de bases de datos deductivas

 Bases de datos multidimensionales

 Bases de datos transaccionales

Modelo de bases de datos orientadas a documentos

5. Tipos de bases de datos

5.1. Según su contenido

BBDD con información actual

Directorios

Documentales

5.2. Según su uso

Individual

Compartida

De acceso público

Propietarias o banco de datos.

5.3. Según la variabilidad de la información

De datos estáticas

De datos dinámicas

5.4. Según la localización de la información

Centralizadas

Distribuidas

5.5. Según el organismo productor

De organismos públicos o administración.

De instituciones sin ánimo de lucro

De entidades privadas o comerciales

Realizadas por cooperación en red

5.6. Según el modo de acceso

Acceso local

En CD-ROM

En línea

5.7. Según cobertura temática

Científico - tecnológicas:

Económico - empresariales:

De medios de comunicación

De ámbito sanitario

Para el gran público

7. Sistemas gestores de bases de datos comerciales

8. Sistemas gestores de bases de datos libres

9. Bases de datos centralizadas

10. Bases de datos distribuidas

10.2 Fragmentación

Mapa conceptual

1. Introducción

Analizando la mayoría de ámbitos de actividad de hoy en día, podemos encontrarnos con multitud de casos en los que se emplean bases de datos:

- Los canales de la TDT
- Agendas de móviles
- Cajeros automáticos
- Certificados en organismos públicos
- GPS
- Plataformas de aprendizaje OnLine
- La consulta del médico
- etc...

El volumen de utilización de bases de datos requiere la existencia de técnicos formados capaces de trabajar con ello.

2. Los ficheros de información

2.1 ¿Qué es un fichero?

Un fichero o archivo es el conjunto de información relacionada, tratada como un todo y organizada de forma estructurada. Es una secuencia de dígitos binarios que organiza la información relacionada con un mismo aspecto.

En la década de los 70 se pasó de tener los datos de contabilidad y facturación de las empresas en formato papel a una primera informatización, proceso en la cual se adaptaron las herramientas para que los elementos que se manejaban en ordenador se parecieran a los que utilizaban manualmente: carpetas, ficheros, directorios, etc...

El elemento que permitió llevar a cabo el almacenamiento de datos de forma permanente en dispositivos de memoria masiva fue el fichero o archivo.

Los ficheros están formados por registros lógicos que contienen datos relativos a un mismo elemento u objeto-

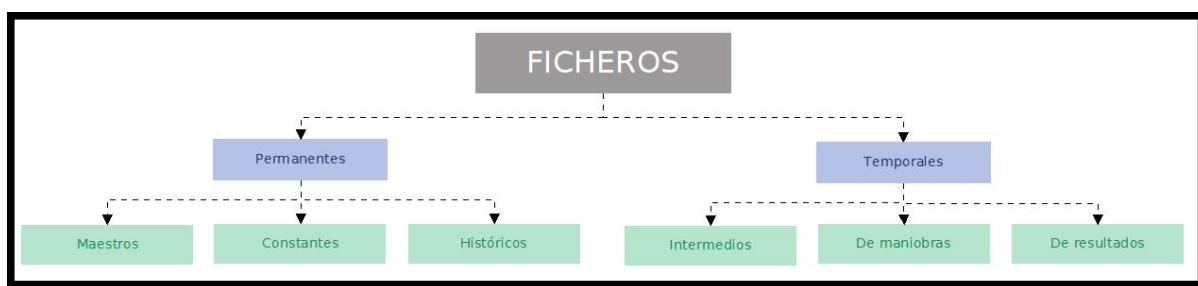
Los ficheros están formados por registros lógicos con datos relativos a un mismo elemento, que a su vez se dividen en campos de información elemental de ese registro. Ejemplo (usuarios de una plataforma educativo y sus campos:

nombre, email, dirección...). Estos ficheros se almacenan de manera que se puedan añadir, suprimir, actualizar o consultar.

Debido al volumen que suelen tener solo se carga en memoria parte de ellos.

- **Registro físico o bloque:** Cantidad de información transferida en una sola operación entre el soporte de almacenamiento y la memoria.
- **Factor de bloqueo:** Cantidad de registros físicos transferidos en cada operación de lectura/grabación.
- **Bloqueo de registros:** Es la operación de agrupar varios registros en un solo bloque.

2.2. Tipos de ficheros



Los ficheros se clasifican de varias maneras según su función.

- **Ficheros permanentes:** contienen información relevante necesaria para el funcionamiento de una app.
 - **Ficheros maestros:** Contienen el estado actual de los datos que pueden modificarse. Es el núcleo de la app. En definitiva, es el conjunto de registros que se refieren a algún aspecto importante de las actividades de una organización (por ejemplo el archivo de vendedores de una empresa).
 - **Ficheros constantes:** Son ficheros que contienen datos fijos para la app y no suelen ser modificados. Se accede a ellos para realizar consultas (como por ejemplo códigos postales),
 - **Ficheros históricos:** Ficheros que fueron considerados actuales en un periodo anterior (por ejemplo los alumnos que han sido dados de baja de una plataforma educativa).

- **Ficheros temporales:** Almacenan información útil para una parte de la app y son generados a partir de los ficheros permanentes. Tienen un periodo corto de existencia.
 - **Ficheros intermedios:** Almacenan resultados de una app que serán utilizados por otras.
 - **Ficheros de maniobras:** Son ficheros auxiliares creados durante la ejecución de una app que almacenan datos que no pueden ser mantenidos en memoria principal por falta de espacio.
 - **Ficheros de resultados:** Almacenan datos que serán transferidos a un dispositivo de salida.

2.3. Los soportes de información

Los ficheros se almacenan en soportes de información manejados por periféricos que permiten leer y grabar datos. Los más utilizados son las cintas magnéticas, ópticos y maneto-ópticos. Los soportes se distinguen en dos tipos en función de la manera que tienen de acceder a los datos.

Soportes de acceso secuencial. Se recorre todo el soporte hasta que llega a la posición del dato que se quiere leer. Inicialmente se utilizaban tambores de cintas magnéticas en los primeros sistemas de almacenamiento.

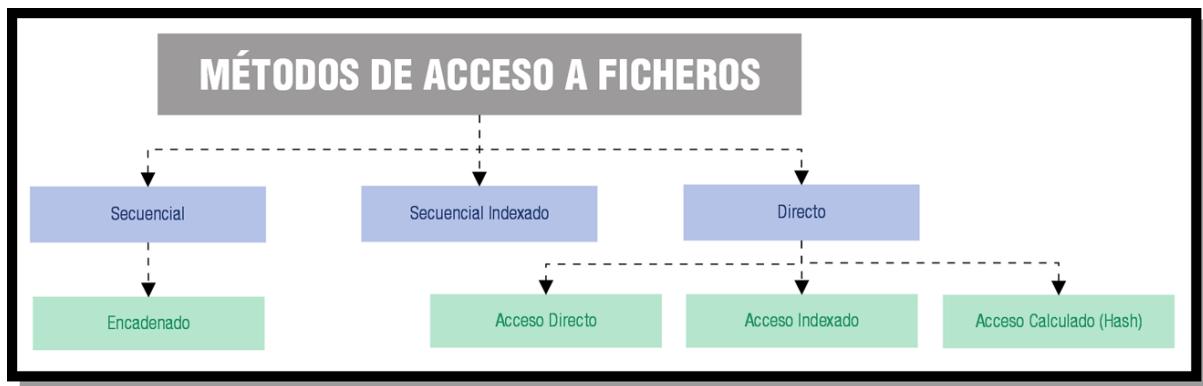
Soportes de acceso directo a los datos. Se puede colocar en la posición que nos interesa y leer a partir de ella. Son los más empleados: discos, díquetes, etc...

2.4. Métodos de acceso

El acceso a la información contenida en ficheros varía según los avances de hardware en función de los siguientes objetivos:

- Acceso rápido a registros.
- Economización de almacenamiento.
- Facilitar actualización de registros.
- Permitir que la estructura refleje la organización real de la información.

A las distintas formas de organizar un fichero en un soporte se les conoce como métodos de acceso.



2.5. Ficheros secuenciales

Se caracterizan por estar almacenados de manera contigua. La única manera de acceder es leyendo un registro tras otro desde el inicio hasta el final. Al final hay una marca de fin de fichero (end of file / EOF).

Se suelen utilizar en soportes no direccionables como cintas magnéticas, pero también en cd y dvd en la que música e imágenes se almacenan en espiral continua.

Los registros se identifican por el campo clave o llave, con la que es más rápido realizar cualquier operación de lectura o escritura.

Sus características principales son:

- Lectura de atrás a alante.
- Ficheros monousuario (no permiten acceso simultáneo).
- Todos los registros deben aparecer en orden.
- El modo de apertura del fichero condiciona su lectura o escritura.
- No deja huecos vacíos en el soporte.
- Pueden grabarse en soportes secuenciales y direccionables.
- Todo lenguaje de programación puede trabajarlos.
- No se pueden insertar registros entre los que ya están grabados.

2.6. Ficheros de acceso directo

En estos ficheros se accede a cada uno de sus registros indicando la posición relativa dentro del archivo a través de un campo clave que a su vez es parte

del mismo registro. El campo clave permite identificar y localizar un registro de manera ágil y organizada.

Cada registro se guarda en una posición física aleatoria dentro del dispositivo de almacenamiento, para acceder a ella se utiliza un índice, de esta manera no se recorre todo el fichero para llegar a un registro determinado.

A través de la clave se obtiene la dirección física del registro. Según esta transformación existen diferentes métodos de acceso:

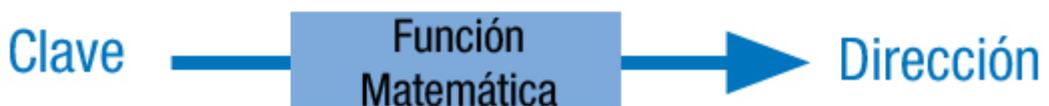
ACCESO DIRECTO



ACCESO INDEXADO



ACCESO CALCULADO



2.6.1. Ficheros de acceso directo

La clave coincide con la dirección, es numérica comprendida dentro del rango de valores de direcciones. Sus características principales son:

- Posicionamiento inmediato.
- Registros de longitud fija.
- Apertura del fichero en modo mixto para lectura y escritura.

- Permite múltiples usuarios.
- Los registros se borran colocando un cero en su posición física.
- Permiten algoritmos de compactación de huecos en memoria.
- Los archivos se crean con un tamaño ya definido.
- Solo aplicable en soportes direccionables.
- Acceso a un registro es por medio de la misma clave y la velocidad de acceso es lo que más importa.
- Los registros se actualizan en el mismo fichero sin necesidad de copiarlo.
- Procesos de actualización en tiempo real.

2.6.2. Ficheros indexados

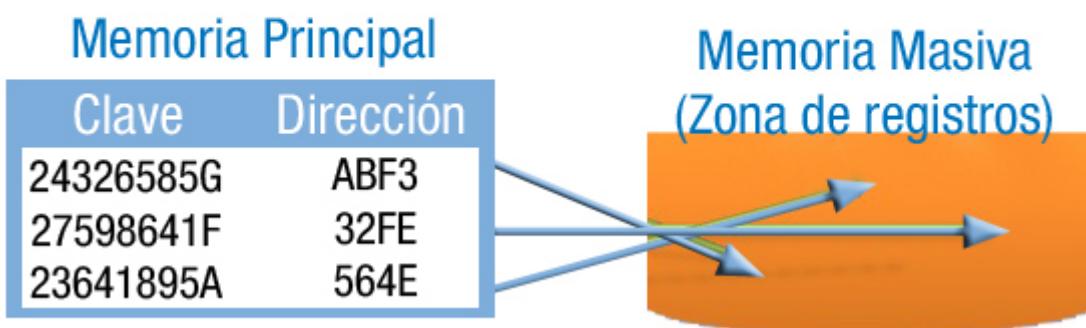
Se basan en la utilización de índices que permiten el acceso a un registro del fichero de manera directa sin tener que leer los anteriores, como si del índice de un libro se tratase para acceder a un capítulo en concreto.

Por tanto estos ficheros cuentan con una zona de índices donde se encuentra una tabla con claves de registros y posiciones físicas donde están. Esta tabla está ordenada por el campo clave.

La tabla de índices se carga en memoria principal para realizar en ella la búsqueda de la fila correspondiente a la clave del registro y con ella, su posición en memoria. El mayor inconveniente es el de no saber determinar el tamaño de la tabla y mantener ordenados los valores de la clave.

- El diseño del registro debe tener uno o varios campos combinados que permitan identificar cada registro de forma única. Este es el campo clave y servirá de índice. Un fichero puede tener más de un campo clave, pero al menos uno no permite valores duplicados (clave primaria). El resto son claves alternativas.
- Permiten acceso secuencial para leer registros.
- Para acceso directo: Se accede a los ficheros conociendo el campo clave del registro que quiere localizar. Con este dato se consulta el índice para conocer la posición del registro dentro del fichero.
- Para acceso secuencial: se leen los registros ordenados por el campo clave, los cuales están ordenados
- Solo se pueden utilizar en soportes direccionables.

FICHEROS INDEXADOS



2.7. Otros métodos de acceso

2.7.1. Secuenciales indexados

También llamados parcialmente indexados. También tienen zona de índice y zona de registro de datos, pero esta última se divide en segmentos o bloques de registros ordenados.

En la tabla de índice cada fila hace referencia a cada uno de los segmentos. La clave corresponde al último registro y el índice al primero del segmento. Cada segmento se recorre de manera secuencial.

- Permite acceso secuencial y los registros se leen ordenados por el campo clave.
- Permite acceso directo a los registro.
- Se actualizan los registros en el mismo fichero.
- Ocupa más espacio que los ficheros secuenciales debido al área de índices.
- Solo en soportes direccionables.
- Inversión económica mayor de programas y hardware más sofisticado.

2.7.2. Ficheros de acceso calculado o hash

A partir de la clave se genera la dirección de cada registro del archivo con una función matemática. El problema que presenta este tipo de fichero es que a

partir de diferentes claves se obtenga la misma dirección al aplicar la función matemática. Este problema se denomina colisión

Para resolver este problema se aplican diferentes métodos como por ejemplo tener un bloque de excedentes o zona de sinónimos, crear un archivo de sinónimos, etc.

La transformación se realiza de distintas formas. Destacamos las siguientes:

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

Una buena transformación es aquella que produzca el menor número de colisiones. Así, la mejor función es buscar una función biunívoca.

2.8. Parámetros de utilización

Según el uso que se le da a los ficheros se utilizarán unos tipos de organización u otros. Mediante los parámetros de referencia podemos determinar el uso de un fichero:

- **Capacidad o volumen:** Se calcula multiplicando el nº previsto de registros por la longitud media de ellos.
- **Actividad:** Permite conocer la cantidad de consultas y modificaciones del fichero. Se debe tener en cuenta:
 - **Tasa de consulta o modificación:** Porcentaje de nº de registros consultados o modificados respecto al nº total de registros.
 - **Frecuencia de consulta o modificación:** Nº de veces que se accede al fichero en un periodo de tiempo fijo.
- **Volatilidad:** Mide la cantidad de inserciones y borrados en un fichero.
 - **Tasa de renovación:** porcentaje de registros renovados en cada tratamiento del fichero respecto al total de registros.
 - **Frecuencia de renovación:** nº de veces que se accede al fichero para renovarlo en un periodo de tiempo fijo.
- **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.

3. Bases de datos

El concepto de base de datos aparece para solucionar el problema de que las aplicaciones dependan de ficheros o archivos, ya que se pierde independencia y se genera información duplicada, incoherencia de datos, fallos de seguridad, etc..

Una base de datos permite reunir toda la información en un único sistema de almacenamiento. Permite a cualquier aplicación utilizarla de manera independiente y mejora el tratamiento de la información.

3.2. Conceptos

Una base de datos es una colección de datos relacionados lógicamente entre sí, con definición y descripción comunes estructurados de una determinada manera. Los datos están estructurados como entidades y sus interrelaciones. Se almacenan con la mínima redundancia y se posibilita el acceso a ellos eficientemente por parte de varias aplicaciones y usuarios.

Las BBDD también almacenan una descripción de los datos denominada metadatos. Se almacena en el diccionario o catálogo y permite que haya independencia de datos lógica y física.

Las BBDD cuentan con:

- **Entidades:** Objeto real o abstracto con características diferenciadoras de otros, del que se almacena información en la BBDD. Por ejemplo, en una clínica veterinaria podrían ser: doctor, consulta, ejemplar....
- **Atributos:** Son los datos o propiedades que se almacenan en la entidad. Por ejemplo: raza, color, nombre, nº de identificación, etc.
- **Registros:** donde se almacena la información de cada entidad. Conjunto de atributos que contienen los datos que pertenecen a una misma repetición de entidad. Por ejemplo: Sultán, Podenco, Gris, etc...
- **Campos:** Donde se almacenan cada uno de los atributos del registro. Ejemplo: Podenco.

Las ventajas de las BBDD son:

- **Acceso múltiple:** Tanto de usuario como de aplicaciones.
- **Utilización múltiple:** Cada usuario o app. podrá tener una visión particular de su estructura y accederá solo a su parte correspondiente.

- Flexibilidad: Accesos establecidos de maneras diferentes, tiempos de respuesta reducidos.
- Confidencialidad y seguridad. Control de acceso a datos a usuarios y apps. Impidiendo a usuarios no autorizados la utilización de la BBDD.
- Protección contra fallos. Mecanismos para recuperación de datos fiable.
- Independencia física. El cambio de soporte físico no afecta.
- Independencia lógica. Los cambios en la BBDD no afectan a las apps.
- Redundancia controlada. los datos no se repiten y si fuera necesario se hará de forma controlada.
- Interfaz de alto nivel. Utilización de bbdd sencilla y cómoda mediante lenguajes de programación de alto nivel.
- Consulta directa. Existe una herramienta para acceder a los datos de manera interactiva.

3.2. Usos

Existen cuatro tipo de personas que pueden usar las BBDD

- **Administrador.**
 - Se encarga de la creación e implementación física de la BBDD.
 - Escoge los tipos de ficheros, los índice y su ubicación.
 - Toma las decisiones con el funcionamiento físico del almacenamiento de la información.
 - Establece políticas de seguridad de acceso.
- **Diseñadores.** Se encargan de diseñar como será la BBDD y llevan a cabo la identificación de los datos e interrelación entre ellos, restricciones, etc. Debe conocer a fondo los datos y procesos a representar en la BBDD y debe implicar en el proceso a todos los usuarios de la BBDD lo más pronto posible.
- Programadores de apps. Los programadores se encargan de implementar programas que realicen consultas, inserción, actualización o eliminación de datos.
- Usuarios finales. Clientes finales de la BBDD. Se busca cumplir los requisitos establecidos por los usuarios para gestionar la información.

3.2. Ubicación de la información

Las bases de datos, dependiendo de su tamaño pueden almacenarse en discos duros u otros dispositivos de almacenamiento a través del ordenador o pueden necesitar servidores en lugares diferentes.

Los sistemas de almacenamiento más utilizados para el despliegue de bases de datos son:

- **Dicos SATA:** Interfaz de transferencia de datos entre la placa base y dispositivos de almacenamiento como discos duros, lectores y grabadores de CD/DVD, etc... SATA proporciona mayores velocidades y mejor aprovechamiento cuando hay varias unidades.... Por velocidades están SATA-150 de 150 MBps., SATA II a 300 MBbps. y SATA III hasta 600 MBps.
- **Discos SCSI.** Interfaces preparadas para discos duros de gran capacidad y velocidad. Su tiempo medio de acceso puede llegar a 7 milisegundos y su velocidad puede alcanzar los 5 MB/s en los SCSI estándares, 10 MB/s en los discos SCSI rápidos y 20 MBps en SCSI en los discos anchos-rápidos. Un controlador SCSI puede manejar 7 HDD SCSI.
- **RAID.** Redundant Array of Independent Disks Es un contenedor de almacenamiento redundante de dos o más discos duros que forman un bloque de trabajo para obtener una ampliación de capacidad, hasta mejoras en velocidad y seguridad de almacenamiento.
- **Sistemas NAS.** Network Attached Storage es un sistema de almacenamiento masivo en red. Permiten compartir la capacidad de almacenamiento de un servidor con otros ordenadores personales o servidores a través de una red. Se hace uso de un S.O. optimizado para dar acceso a los datos. Suelen ser dispositivos de almacenamiento con capacidades altas de varios Terabytes.
- **Sistemas SAN.** Storage Area Network. Se trata de una red concebida para conectar servidores, matrices de discos y bibliotecas de soporte. Recursos disponibles para varios servidores de una red de área local amplia. La información no reside directamente en ningún servidor de la red.

4. Modelos de bases de datos

Existen tres modelos de bases de datos: jerárquico, en red y relacional. El más extendido es el relacional, aunque sus variantes distribuidas y orientadas a

objetos son las que más expansión tienen en los últimos tiempos.

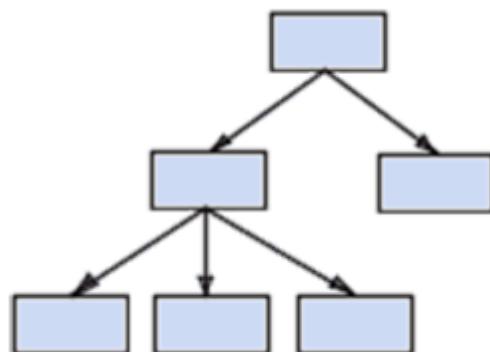
4.1. Modelo jerárquico (1^a generación)

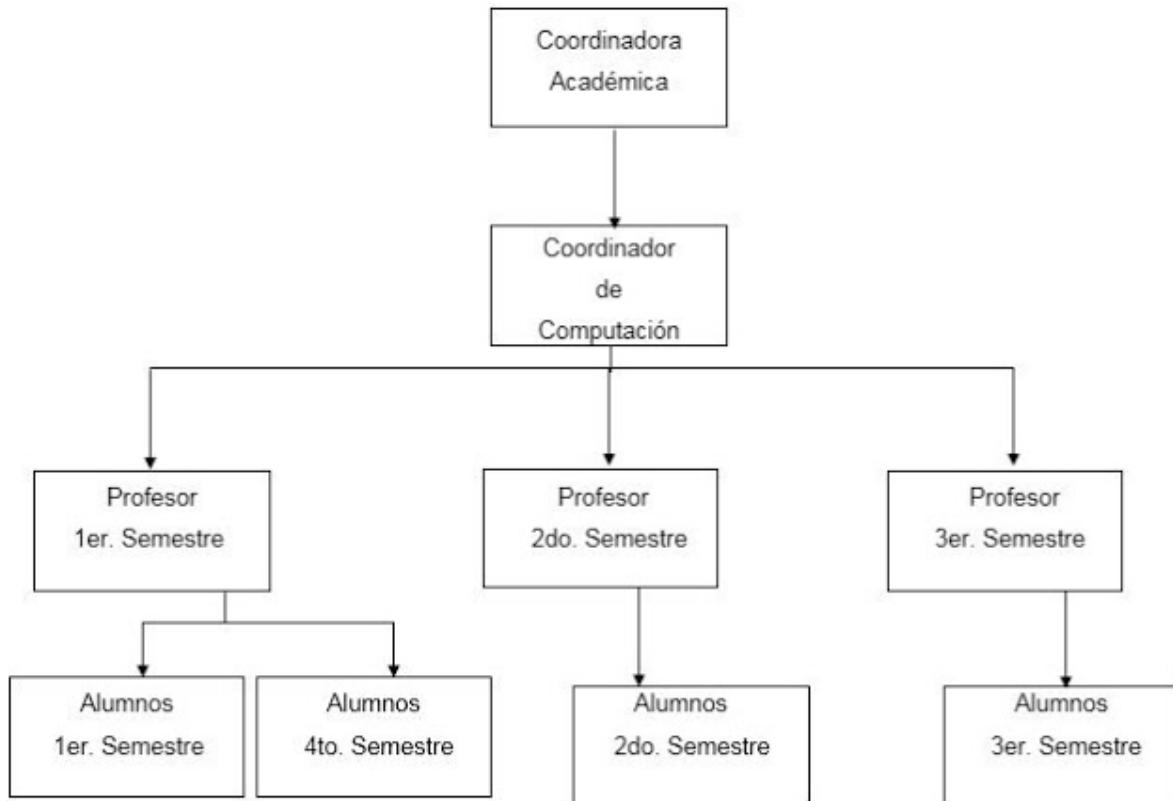
Utilizadas en los años 70 por la gran mayoría de sistemas de gestión. Creado por IBM.

Recibe el nombre de modelo en árbol y su estructura es de árbol invertido. La información se organiza en relaciones de entidades en jerarquía padre/hijo.

Existen nodos que contienen atributos o campos y que se relacionan con sus nodos hijos. Cada nodo puede tener más de un hijo, pero no más de un padre.

Estos datos se almacenan en segmentos que se relacionan entre sí utilizando arcos. Modelo en desuso debido a sus limitaciones.



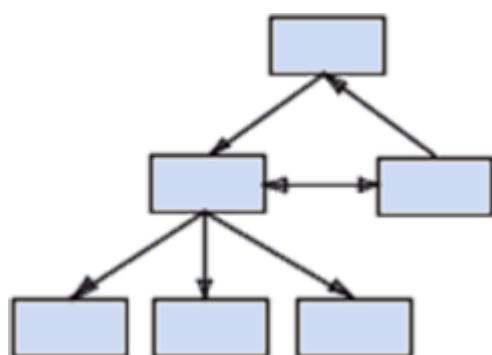


4.2. Modelo en red (1^a generación)

Aparece como respuesta a las limitaciones del modelo jerárquico en cuanto a representación de relaciones complejas.

En él, la información se organiza en registros o nodos, donde se almacenan datos y en enlaces que permiten enlazarlos. Son parecidas a las jerárquicas pero en estas puede haber más de un parente.

Se puede representar cualquier tipo de relación entre datos pero son complicadas de manejar.



4.3. Modelo relacional (2^a generación)

Posterior a los dos anteriores y las más utilizadas hoy en día.

La BBDD consiste en un conjunto de tablas bidimensionales a nivel lógico, ya que a nivel físico puede tener distintas estructuras.

Cada relación (tabla) posee un nombre único y tiene columnas. Se divide en

- Campo o atributo a cada columna de la tabla.
 - De denomina dominio al conjunto de valores de cada columna.
- Registro, entidad o tupla a cada fila de la tabla.

Las tablas deben cumplir los siguientes requisitos:

- Todos los registros son del mismo tipo.
- No existen campos o atributos repetidos.
- No existen registros duplicados.
- No existe orden de almacenamiento.
- Cada registro o tupla tiene una clave formada por uno o varios campos o atributos.



Se utiliza SQL (Structured Query Language) para construir las consultas a las BBDD. Durante el diseño de una BBDD relacional pasa por un proceso que llamamos normalización de una BBDD.

4.4. Modelo orientado a objetos (3^a generación)

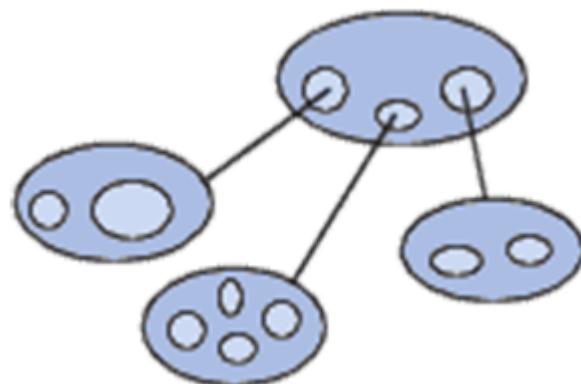
En la actualidad, aunque no han reemplazado a las BBDD relacionales, son las que más están creciendo. En una BBDD orientada a objetos, los objetos con la misma estructura y comportamiento pertenecen a una clase y las clases se organizan en jerarquías.

Las operaciones de cada clase se especifican en términos de procedimientos predefinidos llamados métodos.

Algunas BBDD relacionales han ido incorporando conceptos orientados a objetos. A estos modelos se les llama sistemas objeto-relacionales.

Los conceptos más importantes del paradigma de objetos son:

- **Encapsulación.** Oculta información al resto de objetos, impidiendo accesos incorrectos o conflictos.
- **Herencia.** Los objetos heredan comportamiento dentro de una jerarquía de clases.
- **Polimorfismo.** Propiedad que puede ser aplicada a distintos tipo de objetos.



4.5. Otros modelos

Modelo objeto-relacional (BDOR)

Son un híbrido entre estos dos tipos y surgen como solución al inconveniente de los costes de la conversión de relacional a orientado a objetos.

Los datos se siguen almacenando en tuplas, aunque la estructura de las tuplas no es restringida sino que las relaciones pueden ser definidas en función de otras (herencia directa).

SQL99 es el estándar que ofrece la posibilidad de añadir a las BBDD relacionales procedimientos almacenados de usuarios, triggers, tipos definidos por el usuario, consultas recursivas, bases de datos OLAP, tipos LOB, etc...

Permite incorporar funciones en código de lenguaje de programación.

Modelo de bases de datos deductivas

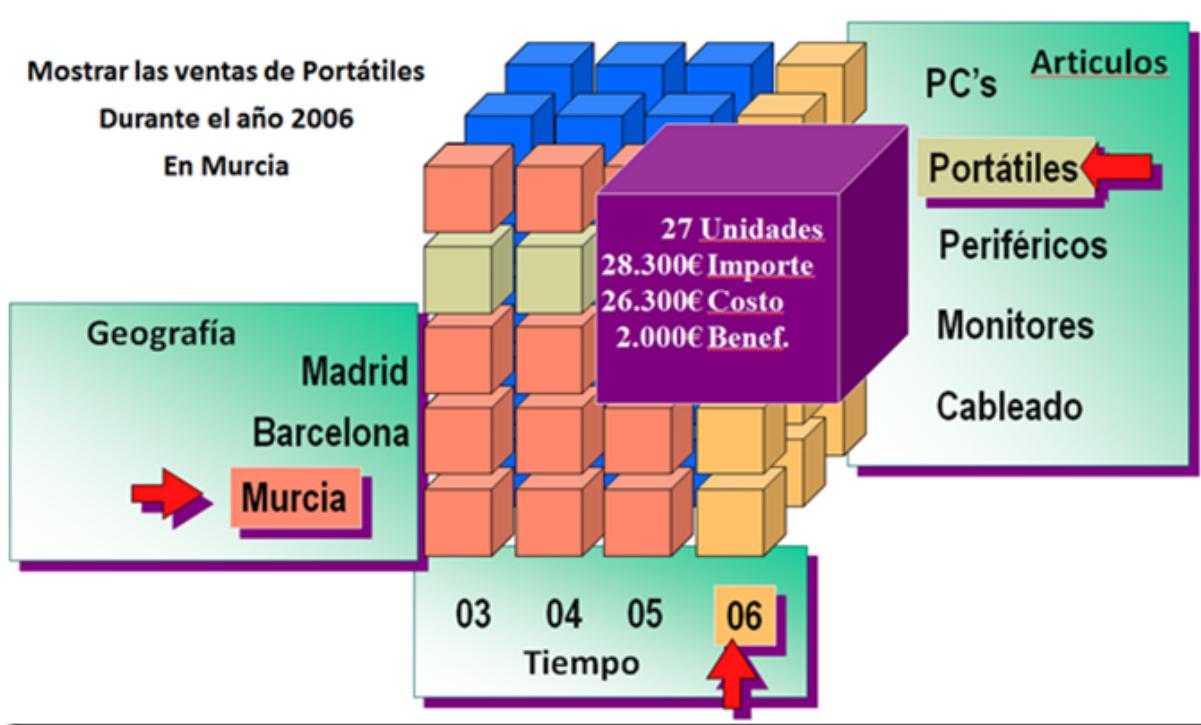
Se almacena información permitiendo realizar deducciones a través de inferencias. Se derivan nuevas informaciones a partir de las que se han introducido en la BBDD.

También se les llama BBDD lógicas, pues se basan en lógica matemática.

Bases de datos multidimensionales

Ideadas para desarrollar aplicaciones muy concretas. Se almacenan datos en varias dimensiones.

La información se representa como matrices multidimensionales, cuadros de múltiples entradas o funciones de varias variables sobre conjuntos finitos.



Bases de datos transaccionales

Se caracterizan por su velocidad para gestionar el intercambio de información. Se utilizan para sistemas bancarios, análisis de calidad y datos de producción industrial. Son BBDD muy fiables, cada inserción, actualización o borrado se realiza completamente o se descarta.

Modelo de bases de datos orientadas a documentos

El principal objeto de gestión son documentos semiestructurados almacenados en algún formato como XML.

5. Tipos de bases de datos

5.1. Según su contenido

BBDD con información actual

Información muy concreta y actualizada, normalmente de tipo numérico: estadísticas, series históricas, resultados de encuestas...

Directories

Recogen datos de personas instituciones especializadas en actividades concretas. Directorios de profesionales, investigadores, bibliotecas...

Documentales

Cada registro se corresponde a un documento (una publicación impresa, documento audiovisual, etc...). Dependiendo de si incluyen el contenido o no de los documentos podemos tener:

- Bases de datos de texto completos. Documentos en formato electrónico volcados en texto.
- Archivos electrónicos de imágenes. Con referencias a enlaces directos con la imagen del documento original.
- Bases de datos referenciales. No contienen el texto original, solo la información fundamental para obtener referencias para localizarlos posteriormente en otro servicio.

5.2. Según su uso

Individual

BBDD utilizada por una persona, administrada y controlada por el mismo usuario. Almacenada en un disco duro o en un servidor de una red de área local.

Compartida

Múltiples usuarios que pertenecen a la misma organización. Se almacena en un ordenador potente bajo el cuidado de un administrador. Se accede a los datos en una red de área local o extensa

De acceso público

Accesibles por cualquier persona, puede pagarse o no canon por el uso de los datos.

Propietarias o banco de datos.

BBDD de gran tamaño desarrolladas por una organización y contienen temas especializados o de carácter particular. Gratis o de pago.

5.3. Según la variabilidad de la información

De datos estáticas

De solo lectura, almacenan datos históricos para analizar y utilizar el estudio del comportamiento de datos a través del tiempo.

De datos dinámicas

Información almacenada que se modifica con el tiempo permitiendo operaciones de modificación y adición de datos además de consultas.

5.4. Según la localización de la información

Centralizadas

Ubicadas en el mismo lugar y un único ordenador. Pueden ser monousuario en ordenadores personales o sistemas de BBDD de alto rendimiento en grandes sistemas. Facilita mantenimiento pero son más vulnerables. Limita su acceso.

- Basadas en anfitrión. La máquina de cliente y servidor son la misma.
- Cliente/Servidor. La BBDD reside en una máquina servidor y los usuarios acceden desde su máquina a través de red.

Distribuidas

Los datos pueden no almacenarse en un único punto, sino en lugares diferentes. Es la unión de BBDD mediante redes. Los usuarios se vuelcan a los servidores de BBDD mediante una red amplia de comunicación.

5.5. Según el organismo productor

De organismos públicos o administración.

Bibliotecas y centros de documentación de ministerios, instituciones públicas, etc...

- De acceso público (gratis o no).
- De uso interno. Con información de acceso restringido.

De instituciones sin ánimo de lucro

Fundaciones, asociaciones, sindicatos, ONG...

De entidades privadas o comerciales

Los centros de documentación y archivos de empresa pueden elaborar distintos sistemas de información.

- De uso interno para facilitar intercambio de datos dentro de la empresa.
- De uso interno que en ocasiones ofrecen servicio al exterior.
- Comerciales, diseñadas para ser utilizadas por usuarios externos.

Realizadas por cooperación en red

Se trata de sistemas elaborados por diversas instituciones, con diversos centros nacionales.

5.6. Según el modo de acceso

Acceso local

Para consultarlas es necesario ir al organismo productor en monopuesto o varios puntos de red local.

En CD-ROM

Por compra o suscripción o por una biblioteca o centro de documentación que permita consultas a sus usuarios.

En línea

Se consultan desde cualquier ordenador conectado a internet, gratis o exigiendo clave personal de entrada

- Acceso vía telnet o internet. El usuario realiza una conexión al host donde se halla la BBDD. Cuando entra en la BBDD se establece una sesión de trabajo interactiva con el gesto de la BBDD.
- Acceso vía web. Conexión a través de un formulario en la web para lanzar preguntas a la BBDD.

Se puede ofrecer acceso a la BBDD en CD-ROM y en línea, con la diferencia que en línea puede estar actualizada diariamente mientras que en CD-ROM no.

5.7. Segundo criterio: según cobertura temática

Científico - tecnológicas:

Contenido destinado a investigadores científicos o técnicos.

- Multidisciplinares: Que abarcan varias disciplinas.
- Especializadas: Recopilan y analizan documentos para una disciplina concreta.

Económico - empresariales:

Contenido dirigido a empresas, entidades financieras, etc.

De medios de comunicación

Información de interés para profesionales de radio, prensa, tv...

De ámbito sanitario

Además de las especializadas también las hay de información de interés sanitario: historiales, médicos, etc..

Para el gran público

Cubren necesidades de información general para los usuarios.

7. Sistemas gestores de bases de datos comerciales

Existen multitud de SGBD comerciales. A veces, el sistema más avanzado según los entendidos puede no serlo para el tipo de proyecto que estemos desarrollando.

- ORACLE: reconocido como uno de los mejores. Multiplataforma, potente a nivel transaccional, confiable y seguro. Modelo de BBDD relacional.
- MYSQL: Muy extendido, licencia comercial y libre. Relacional, multihilo, multiusuario y multiplataforma. Gran velocidad para consulta de BBDD y web.
- DB2: Multiplataforma, integra XML de manera nativa, almacena documentos completos para realizar operaciones y búsquedas relacionales.
- Informix: relacional basado en SQL, multiplataforma, funciones avanzadas de conectividad y relacionadas con tecnología internet/intranet, xml, etc.
- Microsoft SQL Server: relacional, solo funciona bajo windows, arquitectura cliente/servidor.

- Sybase: sistema relacional, escalable, de alto rendimiento, soporta grandes volúmenes de datos, transacciones y usuarios y es de bajo costo.

8. Sistemas gestores de bases de datos libres

Son sistemas gestores open source.

- MySQL: ya explicado en los comerciales.
- PostgreSQL: Sistema relacional orientado a objetos, multiplataforma y accesible desde múltiples lenguajes de programación.
- Firebird: Relacional, multiplataforma, bajo consumo de recurso, buena gestión de concurrencia soporte para diferentes lenguajes.
- Apache Derby: Escrito en java, reducido tamaño, multilenguaje y multiplataforma, portable pero también cliente/servidor.
- SQLite: Sistema relacional basado en biblioteca escrita en C, reduce tiempos de acceso, multiplataforma y soporte para varios lenguajes.

9. Bases de datos centralizadas

Es aquella que está implantada en una sola plataforma u ordenador donde se gestiona de modo centralizado la totalidad de los recursos. Tecnologías sencillas muy experimentadas y de gran robustez.

Los sistemas antiguos eran totalmente centralizados y se caracterizan por:

- Se almacena todo en ubicación central.
- No posee múltiples elementos de procesamiento ni mecanismos de intercomunicación.
- Sus componentes son: los datos, el software de gestión y dispositivos de almacenamiento secundario.
- Sistemas en los que la seguridad puede verse comprometida fácilmente.

Ventajas:

- Se evita redundancia.
- Se evita inconsistencia.

- La seguridad se centraliza.
- Puede conservarse la integridad.
- Mayor rendimiento de procesamiento de datos.
- Mantenimiento más barato.

Inconvenientes:

- En comparación con sistema distribuido no tiene mayor poder de cómputo.
- Si falla, se pierde toda la disponibilidad de procesamiento.
- En caso de desastre o catástrofe la recuperación es difícil
- Las cargas de trabajo se ejecutan en la misma máquina.
- Los departamentos de sistemas retienen el control de toda la organización.
- Requieren mantenimiento central de datos.

10. Bases de datos distribuidas

BDD: Es un conjunto de múltiples bases de datos lógicamente relacionadas distribuidas entre diferentes nodos interconectados por red.

SBDD: Sistema de bases de datos distribuida es un sistema en el cual múltiples BBDD están ligados por un sistema de comunicaciones, así desde cualquier parte de la red se puede acceder a los datos como si estuvieran en el mismo sitio.

SGBDD: Sistema gestor de bases de datos distribuida. Aquel que maneja la BDD y proporciona mecanismos de acceso que hace que la distribución sea transparente a los usuarios (como si un solo SGBD se estuviera ejecutando en una sola máquina que administra esos datos).

Las SGBDD se utilizan en organizaciones de estructura descentralizada y desarrollan su trabajo a través de un conjunto de sitios o nodos que poseen un sistema de procesamiento de datos completo. Se pueden conectar a través de WAN o una LAN.

Ventajas:

- Acceso y procesamiento de datos más rápido gracias a que los nodos comparten la carga de trabajo.

- Acceso a información alojada en distintos lugares desde cualquier ubicación.
- Coste inferior a las BBDD centralizadas.
- Si un nodo deja de funcionar no deja de funcionar el sistema completo gracias a la replicación.
- Se adapta a la estructura de las organizaciones.
- Independencia local aunque los nodos estén interconectados.

Inconvenientes:

- Probabilidad creciente de violaciones de seguridad.
- Complejidad para coordinar nodos.
- Inversión inicial menor, mantenimiento costoso.
- Los datos son replicados por tanto los mecanismos de recuperación son más complejos.
- Sobrecarga de intercambio de mensajes debido a la necesidad de coordinación entre nodos.
- Dificultad para asegurar corrección de algoritmos durante un fallo o recuperación.

10.2 Fragmentación

En las BBDD distribuidas la información está repartida en varios lugares y los datos consultados pueden estar fragmentados en distintas tablas de distintas BBDD en distintos servidores.

El grado de fragmentación es determinante a la hora de ejecutar consultas. No es adecuada la fragmentación nula pero tampoco un nivel alto de fragmentación.

Se deben cumplir las siguientes reglas:

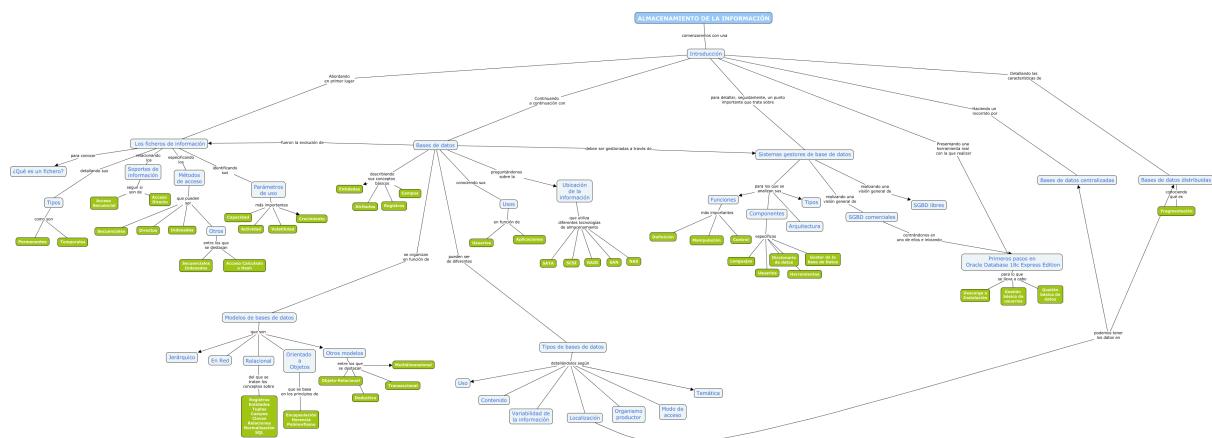
- **Completitud.** Si R se descompone en R1, R2... cada elemento de R se debe encontrar en uno o varios fragmentos de Rn.
- **Reconstrucción.** La reconstrucción de la relación de los fragmentos R1, R2... asegura que se preservan las restricciones sobre datos.

- **Disyunción.** Si se descompone verticalmente, la clave se repite en todos sus fragmentos.

Los tres tipos de fragmentación son:

- **Fragmentación horizontal.** Se realiza sobre las tuplas. Existen dos variantes: la primaria y la derivada.
- **Fragmentación vertical.** Se basa en los atributos para ejecutar la división. Cada uno de ellos contiene el subconjunto de atributos y la clave primaria. Es más complicada que la horizontal.
- **Fragmentación híbrida o mixta.** Si tras la fragmentación vertical se lleva a cabo otra horizontal se llama fragmentación mixta HV para el caso contrario es VH.

Mapa conceptual





2. Bases de datos relacionales

Class	Bases de datos
Column	(X) Xerach Casanova
Last Edited time	@Mar 29, 2021 9:22 PM

- 1. Modelo de datos
- 2. Terminología del modelo relacional
 - 2.1. Relación o tabla. Tuplas. Dominios
 - 2.2. Cardinalidad
- 3. Relaciones. Características de una relación (tabla)
 - 3.1. Tipos de relaciones (tablas)
- 4. Tipos de datos
- 5. Claves
 - 5.1. Clave candidata, clave primaria, clave alternativa
- 6. Índices. Características
- 7. El valor null. Operaciones con este valor
- 8. Vistas
- 9. Usuarios. Roles. Privilegios.
- 10. SQL
 - 10.1. Elementos del lenguaje. Normas de escritura
- 11. Lenguaje de descripción de datos (DDL).
 - 11.1. Creación de bases de datos. Objetos de la base de datos
 - 11.2. Creación de tablas
 - 11.3. Restricciones
 - 11.3.1. Restricción NOT NULL
 - 11.3.2. Restricción UNIQUE
 - 11.3.3. Restricción PRIMARY KEY
 - 11.3.4. Restricción REFERENCES. FOREIGN KEY
 - 11.3.5. Restricción DEFAULT y CHECK.
 - 11.4. Eliminación de tablas.
 - 11.5. Modificación de tablas
 - 11.6. Creación y eliminación de índices.
- 12. Lenguaje de control de datos.
 - 12.1. Dar permisos
 - 12.2. Retirar permisos
- Anexo. Elementos del lenguaje SQL
 - Comandos
 - Comandos DDL.** Lenguaje de definición de datos:

Comandos DML. Lenguaje de manipulación de datos:

Comandos DCL. Lenguaje de control de datos.

Cláusulas

Operadores

Operadores Lógicos

Operadores de comparación

Funciones

Funciones de agregado

Funciones literales

Mapa Conceptual

1. Modelo de datos

Un modelo de datos es un conjunto de métodos y reglas que indican como se ha de almacenar la información y como se han de manipular los datos.

En informática el modelo de datos se implementa con un lenguaje utilizado para la descripción de la BBDD. Se pueden describir las estructuras de datos (tipos y relaciones), restricciones de integridad (condiciones que se deben cumplir según las necesidades de nuestro modelo en la realidad) y las operaciones de manipulación de datos (insertar, modificar, borrar).

Este lenguaje está dividido en dos sublenguajes:

- **Lenguaje de definición de datos (DDL).** Describe de una forma abstracta las estructuras de datos y las restricciones de integridad.
- **Lenguaje de manipulación de datos DML.** Describe las operaciones de manipulación de los datos.

Las fases de modelado se caracterizan por el nivel de abstracción de las mismas (lo alejado que estén del mundo real).

1. **Modelo de datos conceptual.** Representación normalmente gráfica de estructuras de datos y restricciones de integridad. Se realizan en fase de análisis y representan los elementos que intervienen y sus relaciones. Cualquier error en esta fase se arrastra a las siguientes.
2. **Modelo lógico.** Criterios de almacenamiento y operaciones de manipulación de datos dentro de un tipo de entorno informático.
3. **Modelo Físico:** se utiliza en el diseño físico y es la implementación física del modelo anterior. Estructuras de bajo nivel dentro del sistema gestor de bases de datos.

2. Terminología del modelo relacional

Se trata de un modelo lógico que establece una estructura sobre los datos, independientemente del modo en que luego los almacenemos.

El diseño de las tablas del modelo relacional es la segunda fase del diseño de la BBDD, ya que previamente se habrá realizado el diseño conceptual.

El modelo relacional se basa en los subconjuntos del producto cartesiano resultante de dos o más conjuntos.

El producto cartesiano nos da la relación de todos los elementos de un conjunto con todos los elementos de otros conjuntos de ese producto:

Marcas

Nombre
Nissan
Opel
Toyota

Modelos

Nombre
Qashqai
Corsa
Auris

Producto Cartesiano Marcas X Modelos



Marca.Nombre	Modelo.Nombre
Nissan	Qashqai
Opel	Qashqai
Toyota	Qashqai
Nissan	Corsa
Opel	Corsa
Toyota	Corsa
Nissan	Auris
Opel	Auris
Toyota	Auris

2.1. Relación o tabla. Tuplas. Dominios

Una relación es una tabla con filas (tuplas o registros) y columnas (atributos).



Tabla: Alumnos

Atributos:

Tupla →

DNI	NOMBRE	APELLIDOS
1	PEDRO	FERNÁNDEZ
2	LUIS	GÓMEZ
3	PEDRO	GÓMEZ
4	MIGUEL	RIVERO
5	CARLOS	GARCÍA

- **Atributo:** Nombre de cada dato que se almacena en la relación. En la imagen de arriba hay 3 atributos: DNI, nombre y apellidos. Debe describir el significado de la información que representa y a veces debe añadir una pequeña descripción para aclarar más el contenido.
- **Tuplas (registros):** Cada elemento de la relación o tabla. Deben cumplir dos requisitos:
 - Debe corresponderse con un elemento del mundo real.
 - No puede ver dos tuplas con todos los valores iguales.

Un atributo tiene asociado un **dominio** de valores, que son valores pertenecientes a un conjunto previamente establecido. Por ejemplo, en el atributo "Sexo", solo se podrá definir un dominio de valores posibles que sean "M" o "F".

Los dominios deben ser atómicos. Los valores contenidos en un atributo no pueden separarse en valores de dominios más simples.

Un dominio debe tener **nombre, definición lógica, tipo de datos y formato**.

Ejemplo:

- Nombre: Sueldo
- Def. Lógica: Sueldo neto del empleado
- Tipo de datos: Entero.
- Formato: 9.999€

2.2. Cardinalidad

- **Cardinalidad:** número de tuplas de una tabla.
- **Grupo:** tamaño de una tabla en base a sus atributos.

En la siguiente tabla se muestra el producto cartesiano de tres relaciones o tablas (A,B,C), donde A es el nombre de los alumnos, B son las asignaturas y C es si están aprobados o suspendidos.

Producto Cartesiano AxBxC.

A={Carlos, María}	B={Matemáticas, Lengua}	C={Aprobado, Suspensión}
CARLOS	MATEMÁTICAS	APROBADO
CARLOS	MATEMÁTICAS	SUSPENSIÓN
CARLOS	LENGUA	APROBADO
CARLOS	LENGUA	SUSPENSIÓN
CARLOS	INGLÉS	APROBADO
CARLOS	INGLÉS	SUSPENSIÓN
MARÍA	MATEMÁTICAS	APROBADO
MARÍA	MATEMÁTICAS	SUSPENSIÓN
MARÍA	LENGUA	APROBADO
MARÍA	LENGUA	SUSPENSIÓN
MARÍA	INGLÉS	APROBADO
MARÍA	INGLÉS	SUSPENSIÓN

Subconjunto del Producto Cartesiano AxBxC con cardinalidad 5.

A={Carlos, María}	B={Matemáticas, Lengua}	C={Aprobado, Suspensión}
CARLOS	MATEMÁTICAS	APROBADO
CARLOS	LENGUA	APROBADO
CARLOS	INGLÉS	APROBADO
MARÍA	MATEMÁTICAS	APROBADO
MARÍA	INGLÉS	SUSPENSIÓN

Los términos según la nomenclatura utilizada se resume en la siguiente tabla:

Nomenclatura relacional

Nomenclatura tabla

Nomenclatura ficheros

relación = tabla = fichero

tupla = fila = registros

atributo = columna = campos

grado = n° columnas = n° campos

cardinalidad = n° filas = n° registros

3. Relaciones. Características de una relación (tabla)

- Cada tabla o relación debe tener un nombre distinto.

PROVEEDORES

CLIENTES

EMPLEADOS

- Cada atributo o columna toma un solo valor en cada tupla o fila.
- Cada atributo o columna tiene un nombre distinto en cada tabla pero puede ser el mismo en distintas.
- No puede haber dos tuplas o filas completamente iguales.

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

- El orden de las tuplas o filas y el de los atributos o columnas no importa.

<table border="1"><tr><td>a</td><td>20</td></tr><tr><td>b</td><td>30</td></tr><tr><td>c</td><td>70</td></tr></table>	a	20	b	30	c	70	=	<table border="1"><tr><td>a</td><td>20</td></tr><tr><td>c</td><td>70</td></tr><tr><td>b</td><td>30</td></tr></table>	a	20	c	70	b	30
a	20													
b	30													
c	70													
a	20													
c	70													
b	30													
<table border="1"><tr><td>a</td><td>20</td></tr><tr><td>b</td><td>30</td></tr><tr><td>c</td><td>70</td></tr></table>	a	20	b	30	c	70	=	<table border="1"><tr><td>a</td><td>20</td></tr><tr><td>c</td><td>70</td></tr><tr><td>b</td><td>30</td></tr></table>	a	20	c	70	b	30
a	20													
b	30													
c	70													
a	20													
c	70													
b	30													

- Todos los datos de un atributo o columna deben ser del mismo dominio. Si por ejemplo designamos que el atributo Nota solo admite valores "Aprobado" o "Suspenso" el dato "Notable" es incorrecto.

Carlos	Matemáticas	Aprobado
Carlos	Lengua	Suspenso
Carlos	Inglés	NOTABLE
Maria	Matemáticas	Suspenso
Maria	Lengua	Suspenso

3.1. Tipos de relaciones (tablas)

Existen varios tipos de relaciones o tablas:

- **Persistentes:** Solo 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 solo almacenan una definición de consulta. Este resultado procede de otras tablas base o de otras vistas e instantáneas.

Cuando cambian los datos de las tablas base, también cambiarán los de la vista que los utilizan.

- **Instantáneas:** son vistas, pero sí almacenan los datos que muestran además de la consulta que las creó. Modifican su resultado cada cierto tiempo. Es una fotografía de la tabla o relación que dura un tiempo concreto.
- **Temporales:** son tablas eliminadas automáticamente por el sistema.

4. Tipos de datos

Los atributos se mueven dentro de un dominio o conjunto de valores. A ese conjunto de valores se le especifica **el tipo de dato de forma general** y **el conjunto de valores que puede tomar de forma restringida** (estos últimos se indican en la definición de la tabla si el SGBD lo permite). Por ejemplo:

En el atributo "Género" el tipo de dato será "carácter" o "texto de longitud 1". El conjunto de valores que puede tomar es "M" o "F".

Cada campo o atributo debe poseer nombre que esté relacionado con los datos que va a contener y debe tener un tipo de dato que determinará que valores puede tomar y qué operaciones se pueden realizar con ellos.

Los tipos de datos más comunes son:

- **Texto:** cadenas y números que no van a realizar operaciones.
- **Numérico.**
- **Fecha.**
- **Sí/No:** almacena datos que solo tienen posibilidad de verdadero/falso.
- **Autonumérico:** numéricos secuenciales automáticos en cada registro nuevo.
- **Memo** (texto largo).
- **Moneda.**
- **Objeto OLE:** gráficos, imágenes o textos creados por otras apps.

Para determinar el tipo de dato de cada atributo se tiene en cuenta el conjunto de valores y las operaciones que se van a realizar.

5. Claves

Es un atributo o un conjunto de atributos que identifiquen de modo único a cada tupla o fila de la relación o tabla. A ese conjunto se le llama **superclave**.

Una tupla completa, al no poderse repetir se puede considerar una superclave.

En una tabla usuario podríamos utilizar distintas superclaves:

- {Nombre, Apellidos, Login, E-Mail, F_nacimiento}
- {Nombre, Apellidos, Login, E-mail}
- {Login, E-Mail}
- {Login}

5.1. Clave candidata, clave primaria, clave alternativa

Se debe elegir la clave que mejor se adapta a tus necesidades.

- **Clave candidata:** es el atributo o conjunto de atributos que se identifican de manera única en cada tupla. Una tabla debe tener al menos una clave candidata. E-Mail podría ser clave candidata, también podría serlo el conjunto de los atributos Nombre, Apellidos y F_nacimiento. Las claves candidatas deben cumplir los siguientes requisitos:
 - **Unicidad:** No puede haber dos tuplas con los mismos valores para esos atributos.
 - **Irreductibilidad:** si se elimina alguno de los atributos deja de ser única.

La manera de identificar las claves candidatas es conociendo el significado real de los atributos o campos. De esa manera se pueden desechar claves como candidatas fijándonos en valores que podemos llegar a tener. Por ejemplo, sabemos que Nombre + Apellidos no pueden ser clave candidata porque se pueden repetir.

- **Clave primaria:** es aquella que se escoge para identificar sus tuplas de modo único. Siempre hay una clave candidata ya que las tuplas no pueden repetirse y por tanto siempre hay clave primaria. Normalmente son un pequeño subconjunto de los atributos de la tupla. También podemos crear un campo único que identifique las tuplas (por ejemplo código de usuario) que pueden estar constituidos por valores autonuméricos.
- **Claves alternativas.** Son las claves candidatas que no son escogidas como clave primaria. Por ejemplo, si tenemos login como clave primaria, Nombre, Apellidos y F_nacimiento pueden ser claves alternativas

5.3. Clave externa, ajena o secundaria.

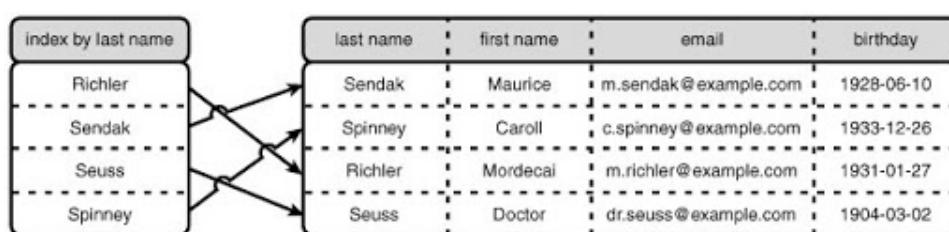
Una clave ajena, externa, foránea o secundaria es un atributo o conjunto de ellos cuyos valores coinciden con los valores de la clave primaria de alguna otra relación o de sí misma. Representan relaciones entre datos.

Las claves ajenas pueden repetirse en la tabla y tienen como objetivo establecer una conexión con la clave primaria que referencian.

Los valores de una clave ajena deben estar presentes como clave primaria en la tabla a la que hacen referencia, o bien deben ser valores nulos. De lo contrario supondría información inconsistente (no fiable).

6. Índices. Características

Un índice es una estructura de datos que permite acceder a diferentes filas de una tabla a través de uno o varios campos definidos como tales, permitiendo acceso mucho más rápido ya que se busca a partir de ellos. Ejemplo:



Los índices son independientes, lógica y físicamente de los datos. Se crean y eliminan sin afectar a las tablas o a otros índices.

Los cambios en los datos de las tablas (agregar, actualizar o borrar) son incorporados de manera automática en los índices con total transparencia. El inconveniente es que estas operaciones se ralentizan al tener que modificar tanto la tabla como el índice. A cambio, se gana velocidad en el acceso a datos.

No hay límite de columnas a indexar, sin embargo debemos elegir columnas en las que vamos a realizar operaciones de búsqueda grandes.

El SGBD utiliza índices para gestión de claves ajenas y primarias y en el caso de claves primaria serán índices únicos que no admiten valores repetidos.

7. El valor null. Operaciones con este valor

Null significa ausencia de dato. Todos los campos pueden tomar este valor independientemente del dominio al que pertenezca.

En claves secundarias, un valor nulo indica que esa tupla no está relacionada con otra.

Un valor nulo NO ES un espacio en blanco (insertar un espacio en un valor perteneciente al dominio texto es distinto a "ausencia de valor") y tampoco será lo mismo que el valor CERO.

Es imprescindible saber como actúa un NULL en operaciones lógicas. Si ambos campos son nulos no podremos obtener ni verdadero ni falso.

- Verdadero AND nulo = Nulo.
- Falso AND nulo = Falso.
- Verdadero OR nulo = Verdadero.
- Falso OR Nulo = Falso.
- Not nulo = Nulo.
- IS NULL devuelve verdadero si el dato que se compara es falso.

8. Vistas

Una vista es una tabla virtual cuyas filas y columnas se obtienen a partir de una o varias tablas de nuestro modelo. No se almacena la tabla en sí, sino su definición, actuando como filtro de las tablas a las que referencia.

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

No hay restricciones para crear vistas pero sí para modificar datos de manera que estas restricciones establecen integridad y consistencia de datos.

Las vistas se crean por dos razones:

- **Seguridad.** Los usuarios solo tienen acceso a parte de la información.
- **Comodidad:** en lenguaje de bbdd trabajar con vistas es menos complejo.

Las vistas no tienen copia física, de manera que los cambios en las tablas afectan a la vista y viceversa, pero no siempre se pueden actualizar datos de una vista. Dependerá de su complejidad y del SGBD.

9. Usuarios. Roles. Privilegios.

Un usuario es un conjunto de permisos que se aplican a una conexión de base de datos. Puede tener además otras características:

- Es el propietario de ciertos objetos (tablas, vistas, etc.).
- Realiza copias de seguridad.
- Tiene asignada una cuota de almacenamiento.
- Tiene asignado un tablespace (una lógica de almacenamiento) por defecto para los objetos Oracle.

Los privilegios son un permiso dado a un usuario para realizar 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.

Un rol de BBDD es una agrupación de permisos de sistemas y de objeto, de manera que si X usuarios tienen el mismo rol, cuando se añaden o quitan permisos se hace a todos los usuarios a la vez.

10. SQL

Structured Query Language es un lenguaje de dominio diseñado para administrar y recuperar información de SGBD relacionales. Permite trabajar con cualquier lenguaje y con cualquier BBDD.

Es recomendable revisar la documentación del SGBD con el que se está trabajando para conocer su sintaxis concreta, ya que no es idéntico en todas.

SQL es potente y versátil y fácil de aprender, con un lenguaje bastante natural. Se considera un lenguaje de cuarta generación.

SQL no es solo un lenguaje de consulta ya que además tiene las siguientes capacidades:

- Definición de la estructura de los datos (DDL).
- Manipulación de datos (DML).
- Especificación de conexiones seguras (DCL).

Se puede trabajar de dos formas con SQL:

- **SQL embebido:** sentencias dentro de un programa escrito en otro lenguaje.
- **SQL interpretado:** entorno gráfico para escribir y ejecutar sentencias o programa en línea de comandos para ejecutar comandos SQL.

10.1. Elementos del lenguaje. Normas de escritura

SQL está compuesto por elementos que se combinan en instrucciones para crear, actualizar y manipular bases de datos. Estos elementos se dividen en:

- **Comandos:** son las instrucciones SQL que se dividen en tres grupos:
 - **De definición de datos (DDL).** Permiten crear datos, tablas, campos, etc.
 - **De manipulación de datos (DML).** Permiten generar consultas para ordenar, filtrar y extraer datos.

- **De control de seguridad.** Administran derechos y restricciones a usuarios.

Estos comandos a su vez están construidos por:

- **Cláusulas:** Condiciones o criterios, palabras especiales para modificar el funcionamiento de un comando. (FROM, WHERE, GROUP, ORDER, HAVING).
- **Operadores:** Aritméticos o lógicos, permiten crear expresiones complejas.
- **Funciones:** Para obtener valores complejos (por ejemplo la función promedio para obtener la media de un salario).
- **Literales:** o constantes, son valores concretos, un número, fecha, conjunto de caracteres...

11. Lenguaje de descripción de datos (DDL).

Se utilizan las sentencias DDL para definir las estructuras donde almacenar información. Modifican, crean y eliminan objetos de la base de datos (metadatos).

En Oracle cada usuario de una base de datos tiene un esquema que tiene el mismo nombre que el usuario y sirve para almacenar objetos que posee ese usuario.

Estos objetos pueden ser: tablas, vistas, índices u otros objetos relacionados con la definición de la BBDD. Solo el propietario y los administradores tienen permiso para manipularlos, a no ser que se modifiquen privilegios para permitir acceso a otros usuarios.

Las instrucciones DDL no se pueden deshacer. Conviene usarlas con precaución y tener copias de seguridad.

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

Una base de datos es un conjunto de objetos que nos servirán para gestionar los datos. Estos objetos están contenidos en esquemas y a su vez están asociados a un usuario.

La creación de la base de datos consiste en crear las tablas que la componen, pero antes se debe definir un espacio de nombres separado para cada conjunto de tablas (**esquemas o usuarios**).

Crear una base de datos implica indicar los archivos y ubicaciones y otras indicaciones técnicas y administrativas que se van a utilizar. Solo lo puede realizar el administrador.

```
CREATE DATABASE NombreDeLaBaseDeDatos;
```

En Oracle no utilizaremos esta sentencia, ya que tiene un sentido distinto a otros SGBD.

11.2. Creación de tablas

Antes de crear la tabla es conveniente recordar que solo se pueden crear si se poseen los permisos necesarios, además, se debe tener a mano la siguiente información que se obtiene de la fase de diseño:

- Nombre de la tabla.
- Nombre para cada columna.
- Tipo y tamaño de datos que vamos a almacenar en cada columna.
- Restricciones sobre los datos.
- Información adicional que necesitemos.

Además se deben cumplir ciertas reglas para los nombres de las tablas:

- Los nombres de las tablas no se pueden duplicar en un mismo esquema.
- Deben comenzar por carácter alfanumérico.
- Longitud máxima de 30 caracteres.
- Solo letras del alfabeto inglés, dígitos o guión bajo.
- No se pueden usar palabras reservadas de SQL.
- No distingue mayúsculas y minúsculas.
- En caso de espacios se entrecilla. No todas las BBDD lo permiten. En Oracle las comillas se utilizan para hacer sensible el nombre de la tabla a mayús/minus.

***CREATE TABLE NombreDeLaTabla (NombreColumna1 Tipo_Dato,
NombreColumna2 Tipo_Dato, ... NombreColumnaN Tipo_Dato);***

Ejemplo:

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

11.3. Restricciones

Una restricción es una condición que una o varias columnas deben cumplir obligatoriamente. Cada restricción debe llevar un nombre. Si no lo ponemos, lo hará el SGBD, pero es conveniente utilizar un nombre único para cada esquema que nos ayude a identificarla.

```

CREATE TABLE NombreDeLaTabla (
    Columna1 Tipo_Dato
    [CONSTRAINT NombreDeRestricción]
    [NOT NULL]
    [UNIQUE]
    [PRIMARY KEY]
    [FOREIGN KEY]
    [DEFAULT valor]
    [REFERENCES NombreTabla [(columna[, columna])]]
    [ON DELETE CASCADE]]
    [CHECK condición],...);

```

Los corchetes indican opcionalidad. Ejemplo:

```

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

```

También se pueden definir las columnas de la tabla y después especificar las restricciones y así referir varias columnas a una única restricción.



Recomendación

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

Prohibe valores nulos para la columna.

```

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

```

O bien:

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

Se debe tener cuidado con los valores NULL en operaciones ($1 * \text{NULL} = \text{NULL}$).

11.3.2. Restricción UNIQUE

Hace que no se puedan repetir valores en una misma columna. Oracle crea un índice cuando se habilita esta restricción y lo borra si se deshabilita.

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

O bien

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

Y para poner esta restricción a varios campos:

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

Usando restricciones para varios campos hay que poner "coma" después del último campo definido, ya que la sentencia de la restricción es independiente, llevando los campos que se quieren restringir después entre paréntesis.

11.3.3. Restricción PRIMARY KEY

Se debe indicar en una tabla que columna o columnas son clave primaria. Solo puede haber una y podrá ser referenciada como clave ajena en otras tablas.

Los campos que forman la primary key son **NOT NULL** y de tipo **UNIQUE**.

Clave de único campo:

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

o bien:

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

Clave formada por más de un campo:

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

Nunca olvidar la "coma" después del último campo.

11.3.4. Restricción REFERENCES. FOREIGN KEY

Cuando creamos la tabla debemos indicar la clave ajena haciendo referencia a la tabla y los campos de donde procede. Por ejemplo para unir una tabla llamada Partidas a través del campo Cod_Partida, crearemos en nuestra tabla usuarios un campo Cod_Partida que haga referencia a la tabla y campo que se quiere referenciar:

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

Si el campo al que se hace referencia es clave principal no es necesario indicar el nombre del campo.

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

Poniendo la definición de clave al final debemos colocar el texto FOREIGN KEY. En el caso de que la clave ajena estuviese formada por dos campos se haría de la siguiente manera:

```
CREATE TABLE USUARIOS (
    Cod_Partida NUMBER(8),
    F_Partida DATE,
    CONSTRAINT Partida_Cod_F_FK FOREIGN KEY (Cod_Partida, F_Partida)
    REFERENCES PARTIDAS);
```

No olvidar la coma después del último campo.

Llamamos **integridad referencial** al hecho de tener la clave secundaria de la tabla incluida en su tabla de procedencia, donde es clave primaria o candidata. Cualquier dato que incluyamos en la clave ajena debe estar previamente en la tabla de la que procede. Esto puede generar algunos errores:

- Si hacemos referencia a una tabla no creada, Oracle la buscará y dará fallo. Se soluciona creando primero las tablas que no tienen claves ajenas.
- Si queremos borrar las tablas primero debemos borrar las que tengan claves ajenas.

Pero existen otras soluciones tras la cláusula **REFERENCE**:

- **ON DELETE CASCADE**: Borra todos los registros cuya clave ajena sea igual a la clave del registro borrado.
- **ON DELETE SET NULL**: Coloca NULL en claves ajenas relacionadas con la borrada.
- **ON DELETE DEFAULT xxxx**: coloca el valor xxxx en todas las claves ajenas relacionadas con la borrada.

Estas opciones valen también en **ON UPDATE**.

11.3.5. Restricción **DEFAULT** y **CHECK**.

Default permite insertar un valor por defecto en un campo:

```
CREATE TABLE USUARIOS (
    País VARCHAR(20) DEFAULT 'España');
```

Se puede añadir distintas expresiones: constantes, funciones SQL y variables:

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

CHECK comprueba que los valores que se introducen son los adecuados.

Esta condición se puede construir con columnas de la tabla:

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

Una misma columna puede tener varios check asociados. Para ello se ponen varios **CONSTRAINT** seguidos y separados por comas.

11.4. Eliminación de tablas.

Eliminamos tablas cuando ya no son útiles para no ocupar espacio:

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

La opción CASCADE CONSTRAINTS se incluye cuando alguna clave es ajena en otra tabla. De esta forma se borran las restricciones donde es clave ajena y después se elimina la tabla.

- Se borrará la tabla y sus datos (filas). También se borra la información de esa tabla en el diccionario de datos.
- El borrado es irreversible y no hay confirmación antes de ejecutarse.
- Las vistas de una tabla siguen existiendo pero ya no funcionan.
- Oracle tiene la orden **TRUNCATE TABLE**. Borra las filas pero no la estructura.
- Solo se permite borrar tablas con permiso de borrado.

11.5. Modificación de tablas

Se pueden modificar tablas después de crearlas, añadir, eliminar o modificar campos, añadir o eliminar restricciones...

- **Cambiar nombre de tabla:**

```
RENAME Nombreviejo TO NombreNuevo;
```

- **Añadir columnas (se añaden al final).**

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

- **Eliminar columnas.** No se puede deshacer la acción, se elimina la definición y sus datos, no se puede eliminar una columna que es la única que forma la tabla:

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

- **Modificar columnas.** Los cambios son posibles si no contiene datos. Si los tiene, solo podremos aumentar la longitud de la columna, aumentar o disminuir nº de

posiciones decimales en tipo NUMBER o reducir anchura siempre que los datos no ocupen todo el espacio reservado.

```
ALTER TABLE NombreTabla MODIFY  
(Columna1 TipoDatos [propiedades]  

```

- **Renombrar columnas.**

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

- **Borrar restricciones.**

```
ALTER TABLE NombreTabla DROP CONSTRAINT NombreRestriccion;
```

- **Modificar nombre de restricciones.**

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

- **Activar o desactivar restricciones.** A veces es conveniente desactivarlas temporalmente para hacer pruebas o saltarse la regla.

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

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

La opción CASCADE desactiva/activa las restricciones que dependan de ésta.

Para consultar las restricciones podemos consultar el diccionario de datos all_constants.

11.6. Creación y eliminación de índices.

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

No se aconseja utilizar índices sobre campos de tablas pequeñas o que se actualicen con frecuencia. Tampoco si esos campos no se usan en consultas de manera frecuente o en expresiones.

Una mala elección ocasiona ineficiencia y un uso excesivo puede dejar la BBDD colgada solo por insertar una fila.

```
DROP INDEX NombreIndice;
```

La mayoría de índices se crean implícitamente en restricciones PRIMARY KEY, FOREIGN KEY O UNIQUE.

12. Lenguaje de control de datos.

Se necesitan cuentas de usuario para acceder a los datos de una BBD, estas claves de acceso se establecen cuando se crea el usuario y las modifica el administrador o propietario de la clave. Se almacenan encriptadas en DBA_USERS. Para crear usuarios se debe contar con privilegios de administrador.

Así se crean usuarios:

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

- **CREATE USER:** Nombre de usuario
- **IDENTIFIED BY:** Contraseña.
- **DEFAULT TABLESPACE:** Asigna al usuario el nombre del tablespace por defecto para almacenar los objetos que cree. Si no se asigna una, será SYSTEM.
- **TEMPORARY TABLESPACE:** Especifica el nombre del Tablespace para trabajos temporales. Por defecto 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 será el perfil por defecto.

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;**

Para modificar usuarios se utiliza:

```
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 solo podrá modificar su propia contraseña.

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

La opción CASCADE borra todos los objetos del usuario antes de eliminarlo. Sin esa opción no deja borrar al usuario si tiene tablas creadas.

12.1. Dar permisos

Los usuarios no pueden llevar a cabo operaciones si no tienen permisos para ello.

Para acceder a objetos de una BBDD se necesitan privilegios que se pueden agrupar formando **roles**.

Estos roles pueden gestionar los comandos que pueden utilizar los usuarios y pueden activarse, desactivarse o protegerse con clave.

Para dar privilegios sobre objetos ya existentes:

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

- **ON** especifica el objeto sobre el que se conceden 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 pueda conceder a otros.
- **PUBLIC** hace que un privilegio esté disponible para todos los usuarios.

Por ejemplo:

```
GRANT INSERT ON usuarios TO Ana; /*concede permisos a Ana para insertar  
datos en la tabla Usuarios.*/
```

```
GRANT ALL ON Partidas TO Ana; /*concede todos los privilegios sobre la tabla Partidas a Ana.*/
```

Para dar privilegios de sistema (estos dan derecho a ejecutar comandos SQL o acciones sobre objetos). La sintaxis es así:

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

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

Por ejemplo:

```
GRANT CONNECT TO Ana; /*Concede a Ana el rol Connect con todos los privilegios que tiene asociados.*/
GRANT DROP USER TO Ana WITH ADMIN OPTION; /*Concede a Ana el priv. de borrar usuarios y además puede conceder el mismo privilegio a otros usuarios.*/
```

12.2. Retirar permisos

Con el comando REVOKE se retiran privilegios.

Sobre objetos.

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

Por ejemplo:

```
REVOKE SELECT, UPDATE ON Usuarios FROM Ana; /*Quitar permiso a Ana de seleccionar y actualizar la tabla de Usuarios*/
```

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}] ...;
```

Por ejemplo:

```
REVOKE DROP USER FROM Ana; /* Quitamos permiso a Ana para  
eliminar usuarios*/
```

Anexo. Elementos del lenguaje SQL

Los elementos SQL se combinan en las instrucciones que se utilizan para crear, actualizar y manipular BBDD.

Comandos

Comandos DDL. Lenguaje de definición de datos:

- **CREATE** (crear nuevas tablas campos e índices).
- **DROP** (eliminar tablas e índices).
- **ALTER** (modificar tablas).

Comandos DML. Lenguaje de manipulación de datos:

- **SELECT** (consultar filas con un criterio determinado).
- **INSERT** (insertar filas en una tabla).
- **UPDATE** (modificar valores de campos y filas específicos).
- **DELETE** (eliminar filas de una tabla).

Comandos DCL. Lenguaje de control de datos.

- **GRANT** (Dar permisos a uno o varios usuarios o roles).
- **REVOKE** (quitar permisos que previamente se han concedido).

Cláusulas

Llamadas también condiciones o criterios. Permiten modificar el funcionamiento de un comando.

- **FROM** (especifica la tabla de la que se seleccionarán las filas).
- **WHERE** (con él se especifican las condiciones que deben reunir las filas que se van a seleccionar).
- **GROUP BY** (se usa para separar las filas en grupos específicos).

- **HAVING** (se usa para expresar la condición que debe tener un grupo).
- **ORDER BY** (Ordena las filas seleccionadas en un orden específico).

Operadores

Permiten crear expresiones complejas

Operadores Lógicos

- **AND** (Evalúa dos condiciones y devuelve un valor de verdad 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

- < (Menor que)
- > (Mayor que)
- <> (Distinto que)
- <= (Menor o igual que)
- >= (Mayor o igual que)
- = (Igual)
- **BETWEEN** (para especificar un intervalo de valores).
- **LIKE** (para comparar)
- **IN** (para especificar filas de una base de datos).

Funciones

Para conseguir valores complejos. Existen muchas, estos son solo algunos ejemplos.

Funciones de agregado

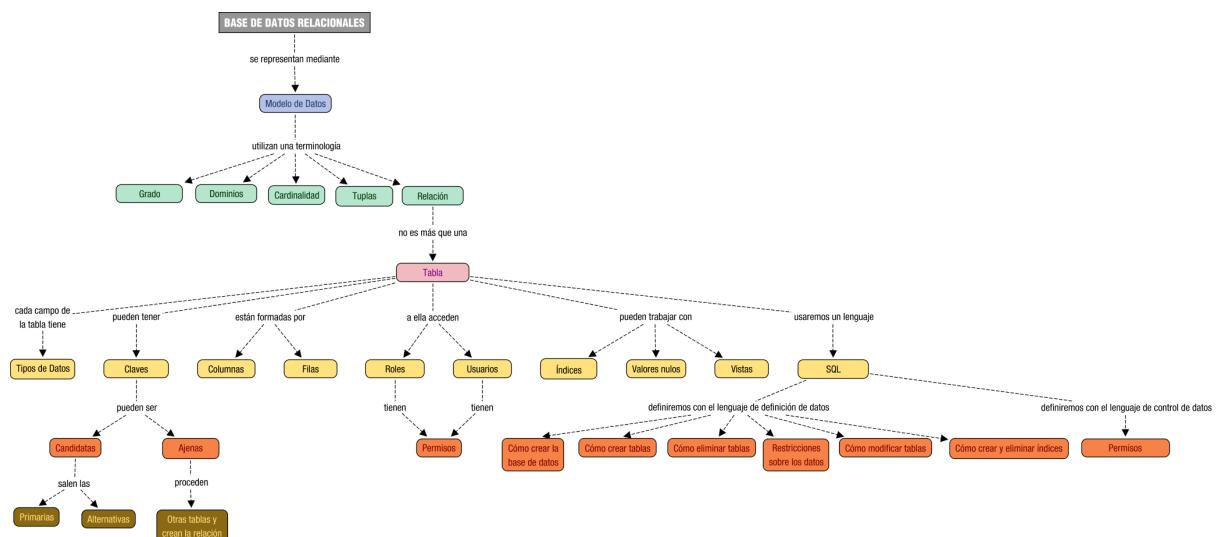
- **AVG** (Calcula el promedio de los valores de un campo determinado).
- **COUNT** (devuelve el nº de filas de la selección)
- **SUM** (devuelve la suma de los valores de un campo determinado).
- **MAX** (devuelve el valor más alto).
- **MIN** (devuelve el valor más bajo).

Funciones literales

También llamadas constantes. Son valores concretos: un número, una fecha, un conjunto de caracteres...

- **23/03/97** (literal fecha).
- **María** (literal caracteres).
- **5** (literal número).

Mapa Conceptual





3. Interpretación de diagramas entidad/relación

Class	Bases de datos
Column	(X) Xerach Casanova
Last Edited time	@Mar 29, 2021 9:23 PM

1. Análisis y diseño de bases de datos
2. ¿Qué es el modelo E/R?
3. Entidades
 - Tipos: Fuertes y débiles
4. Atributos
 - 4.1. Tipos de atributos
 - Atributos obligatorios u opcionales
 - Atómicos o compuestos
 - Atributos monovaluados o multivaluados
 - 4.2. Claves
 - 4.3. Atributos de una relación
5. Relaciones
 - 5.1. Grados de una relación
 - 5.2. Cardinalidad de relaciones
 - 5.3. Cardinalidad de entidades
6. Resumen de simbología del modelo E/R
7. El modelo E/R extendido
 - 7.1. Restricciones en las relaciones
 - Restricción de exclusividad
 - Restricción de exclusión
 - Restricción de inclusividad
 - Restricción de inclusión
 - 7.2. Generalización y especialización
 - 7.3. Agregación
8. Elaboración de diagramas E/R
 - 8.1. Identificación de entidades y relaciones
 - 8.2. Identificación de atributos, claves y jerarquías
- 8.3. Metodologías

8.4. Redundancia en diagramas E/R

8.5. Propiedades deseables de un diagrama E/R

9. Primeros pasos del diagrama E/R al modelo relacional

9.1. Simplificación previa de diagramas

Transformación de atributos compuestos

Transformación de atributos multivaluados

Transformación a relaciones jerárquicas

Transformación de relaciones cíclicas

Transformación de relaciones ternarias

10. Paso del diagrama E/R al modelo relacional

Relaciones 1/1 de E/R a esquema relacional

Relaciones reflexivas a esquema relacional

Jerarquías de E/R a esquema relacional

Relaciones de muchos a muchos a esquema relacional

Relaciones N-Arias a esquema relacional

11. Normalización de modelos relacionales

11.1. Tipos de dependencias

Dependencia funcional

Dependencia funcional completa

Dependencia transitiva

11.2. Formas normales

1^a forma normal

2^a forma normal

3^a forma normal

Forma normal de Boyce Codd

Otras formas normales

Mapa Conceptual

El objetivo principal de la arquitectura de los SGBD es separar los programas de aplicación de bases 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 bbdd a través del esquema conceptual, así se representa la información de una manera independiente al SGBD.

En el desarrollo de BBDD se distinguen dos fases: **análisis y diseño.**

Fase de análisis

- Análisis de entidades: localizar y definir las entidades y sus atributos.
- Análisis de relaciones: definir relaciones entre entidades
- Obtención de esquema conceptual a través del modelo E-R.
- Fusión de vistas: Un único esquema con todos los esquemas existentes en función de diferentes vistas de perfil de usuario.
- Aplicación de enfoque de datos relacional.

Fase de diseño

- Diseño de tablas
- Normalización
- Aplicación de retrodiseño si fuese necesario (volver al análisis si fuese necesario).
- Diseño de transacciones: localización del conjunto de operaciones o transacciones que operarán sobre el esquema conceptual
- Diseño de sendas de acceso: se formalizan los métodos de acceso dentro de la estructura de datos.

Una correcta fase de análisis determina el éxito de la bbdd. Saltarse el esquema conceptual implica pérdida de info con respecto al problema real y debe reflejar los aspectos relevantes del mundo real que se va a modelar.

El modelo entidad/relación extendido (ERE) es el más aceptado y evoluona del original modelo E/R.

2. ¿Qué es el modelo E/R?

Es una herramienta que representa conceptualmente los problemas del mundo real y su objetivo es facilitar el diseño de la BBDD a través de un esquema que representa toda su estructura.

El esquema tiene descripciones textuales de la realidad que establecen los requerimientos del sistema, buscando ser lo más fiel posible al comportamiento del mundo real.

Al representar los datos semánticamente no está orientado a ningún sistema físico o informático. Se puede incluso utilizar para describir procesos de producción, estructuras de empresa, etc.

Este sistema utiliza símbolos y expresiones determinados, así como objetos básicos llamados entidades y relaciones entre esos objetos.

3. Entidades

En definitiva, una entidad es un objeto real o abstracto con características diferenciadoras capaces de hacerse distinguir de otros objetos, de la que además se desea guardar información.

CLIENTE

Un conjunto de entidades es un grupo de entidades con las mismas características o propiedades. Por ejemplo: conjunto de clientes, conjunto de ríos existentes en una determinada zona, etc...

En el modelo E/R, una entidad se representa mediante un rectángulo con el nombre de la entidad dentro.

Tipos: Fuertes y débiles

A su vez, las entidades se clasifican en:

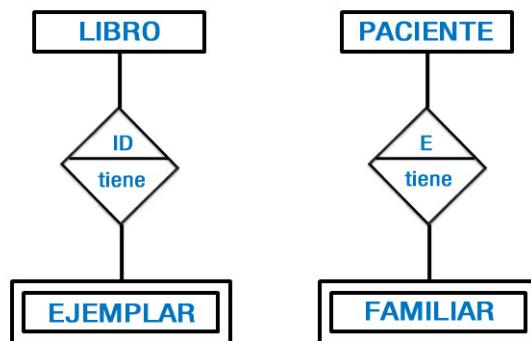
- **Fuertes o regulares:** su existencia no depende de otras entidades y se representan como se ha descrito anteriormente. Ejemplo: Doctor, Paciente.
- **Entidades débiles:** es un tipo de entidad cuyas propiedades o atributos no la identifican completamente, solo de forma parcial, por tanto debe participar en una relación con otra entidad que ayude a identificarla.

AULA

Estas entidades a su vez presentan dos tipos de dependencia.

- **Dependencia de existencia.** Si desaparece una instancia de entidad fuerte desaparecen las entidades débiles. Se representa con una E en el interior de una relación débil.
- **Dependencia de identificación.** También existe dependencia de la entidad fuerte, pero además no se puede identificar por sí misma y necesita de la ID de la entidad fuerte para poder identificarse. Se representa con "ID" en el interior de la relación débil. Por ejemplo. Existen una entidad fuerte llamada "Almacén" y otra débil llamada "Pedidos". Queremos tener una lista de pedidos numerados en cada almacén y puede existir un pedido de idéntica numeración en dos almacenes (por ejemplo 20340). La

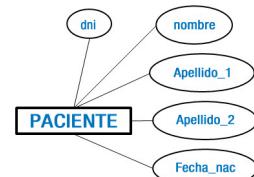
identificación única de un pedido será la combinación de la ID de Almacén con la ID del pedido (1,20340 y 2,20340).



4. Atributos

Los atributos son cada una de las propiedades o características que tiene un tipo de entidad o un tipo de relación.

En el modelo E/R se identifican mediante el nombre del atributo rodeado de una elipse y se conecta con la entidad con una línea recta. Cada atributo debe tener nombre único.



Los valores permitidos dentro de cada atributo son los dominios. Varios atributos pueden estar definidos dentro del mismo dominio. Por ejemplo "nombre, apellido_1 y apellido_2. Los dominios pueden ser amplios, pero se deben establecer límites en el SGBD para garantizar la integridad de datos.

4.1. Tipos de atributos

Atributos obligatorios u opcionales

- **Atributo obligatorio.** Siempre ha de estar definido en cada ocurrencia de una entidad o relación. Una clave o llave es un atributo obligatorio.
- **Atributo opcional.** Puede ser definido o no en cada ocurrencia de la entidad.

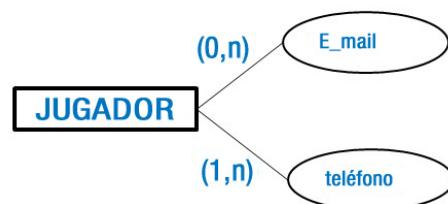
Atómicos o compuestos

- **Atributo atómico o simple.** Es un atributo que no puede dividirse en partes más pequeñas. Por ejemplo el atributo DNI.
- **Atributo compuesto.** es un atributo que puede ser dividido en subpartes que constituyen otros atributos con significado propio. Ejemplo: Dirección (puede estar compuesto por calle, número y localidad).

Atributos monovaluados o multivaluados

- **Atributo monovaluado:** es aquel que tiene un único valor para cada ocurrencia. Por ejemplo: DNI.
- **Atributo Multivaluado:** puede tomar diferentes valores para cada ocurrencia de entidad. Por ejemplo, el e-mail de un empleado puede tomar varios valores para quien posea varias cuentas de correo. Hay que tener en cuenta:
 - **Cardinalidad de un atributo:** Indica el número mínimo y máximo de valores que puede tomar para cada ejemplar de la entidad o relación a la que pertenece.
 - **Cardinalidad mínima.** Indica la cantidad mínima (casi siempre 0 y 1).
 - **Cardinalidad máxima.** Indica la cantidad de máxima (entre 1 y n).

En el ejemplo, el atributo e-mail puede tener de cero a n valores y el teléfono puede tener de 1 a n valores



- **Atributos derivados, calculados o almacenados:** Son obtenidos del valor o valores de otros atributos. Por ejemplo, el atributo edad puede ser calculado a partir del atributo fecha de nacimiento. No conviene almacenar edad, ya que es un elemento que varía con el tiempo.

4.2. Claves

Las claves son un atributo especial obligatorio.

Los valores de los atributos de una entidad deben permitir identificar únicamente a la entidad. Es decir, no puede haber dos ocurrencias exactamente iguales en una entidad.

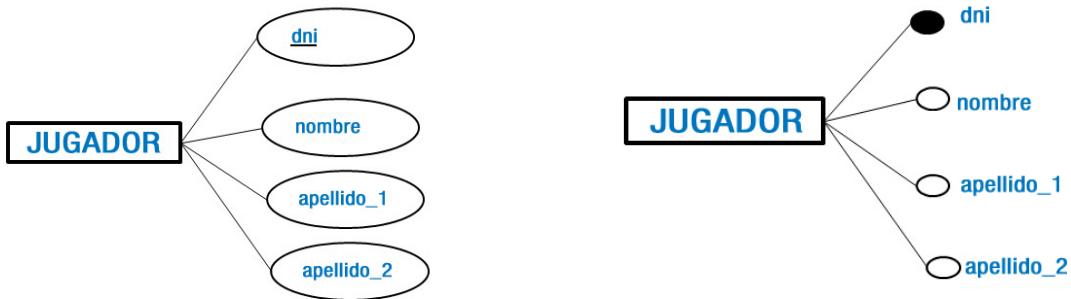
- **Súperclave (superllave).** Es un conjunto de atributos que permite identificar de forma única a cada ocurrencia de la entidad. Puede tener atributos no obligatorios (no identificarían por si solos a la ocurrencia).
- **Clave candidata.** Puede ser un atributo único o una superclave siempre que no se pueda elegir un subconjunto que pueda ser superclave. Una clave candidata debe ser única (no puede haber dos tuplas con los mismos valores para esos atributos) e irreducible (si se elimina algún atributo deja de ser única). Para elegir las claves candidatas debemos tener en cuenta lo siguiente:
 - Sus valores deben ser distintos de nulo.
 - La memoria que ocupen debe ser la menor posible.
 - Codificación sencilla.
 - El contenido de los valores no debe variar.
- **Clave primaria (Primary Key).** De todas las claves candidatas, el diseñador debe escoger una, que será la clave primaria. Es un atributo o conjunto de ellos que toman valores únicos y distintos para cada ocurrencia. No puede tener valores nulos. Entre varias claves candidatas se debe elegir la clave primaria atendiendo a los siguientes criterios:
 - La de menos longitud.
 - Simples antes que compuestas.
 - Numéricas sobre no numéricas.
 - Codificadas sobre no codificadas.
 - Ámbito local sobre ámbito más general.

Una vez elegida la clave primaria, las restantes claves candidatas son denominadas claves alternativas o secundarias.

La representación de modelo E/R de las claves primarias se realiza de dos formas:

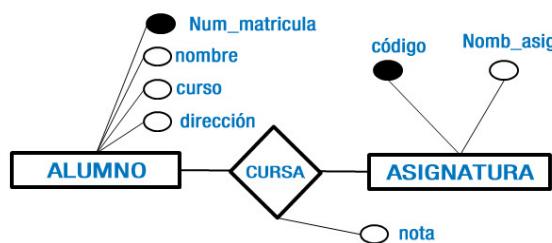
- En representación con elipses se subraya los atributos que formen la clave primaria

- En representación con círculo se utilizan círculos negros en los que formen la clave primaria



4.3. Atributos de una relación

Una relación puede tener atributos que la describan y se representa de la misma manera que cualquier otro atributo, pero con la línea recta uniéndolo con la propia relación.



5. Relaciones

La relación o interrelación es un elemento del modelo E/R que permite relacionar datos entre sí. Es una asociación entre diferentes entidades. No puede aparecer la misma ocurrencia relacionada dos veces en la misma relación.



La representación gráfica es un rombo con un verbo en su interior y líneas rectas que pueden o no acabar en punta de flecha uniendo las distintas relaciones.

5.1. Grados de una relación

Es el número de entidades que participan en una relación.

- **Relación Unaria o grado 1.** Participa una única entidad (reflexivas o recursivas).
- **Relación Binaria o grado 2.** Participan 2 entidades. En general se busca que el esquema conceptual de la bbdd tenga este tipo de relaciones.
- **Relación Ternaria o grado 3.** Participan 3 entidades al mismo tiempo.
- **Relación N-aria o grado n.** Involucra a un número n de entidades. Deben ser simplificadas y hacerlas de menor grado.
- **Relación dobe.** dos entidades se relacionan a través de dos relaciones.
Complejidad de manejo.

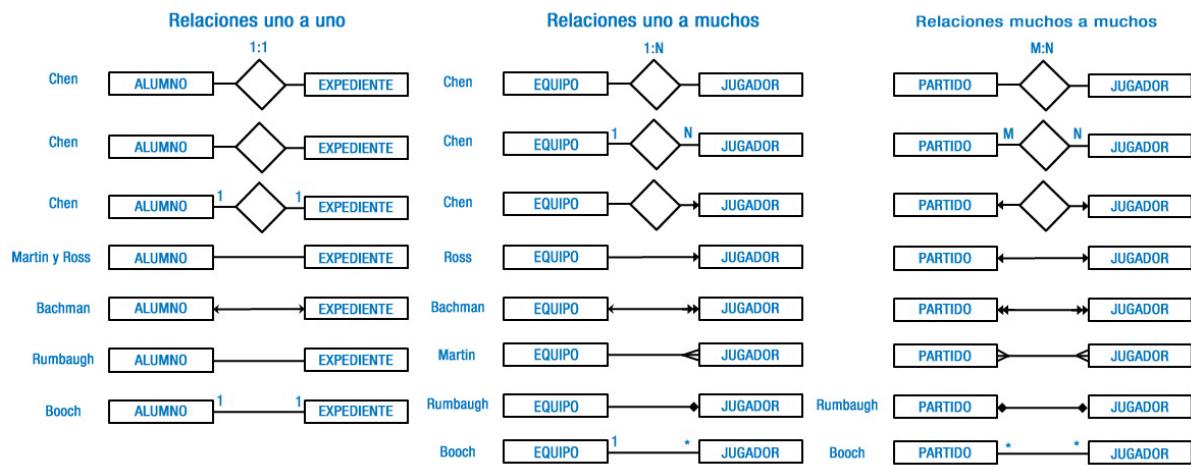
5.2. Cardinalidad de relaciones

Es el número máximo de ocurrencias de cada entidad que pueden intervenir en una ocurrencia de relación. Viene expresada para representaciones entre dos entidades y pueden ser:

Dadas las entidades A y B...

- **Relaciones de uno a uno (1:1).** una ocurrencia de A se relaciona con una sola ocurrencia de B y viceversa. Ejemplo: un alumno solo puede tener un expediente y un expediente puede pertenecer a un solo alumno.
- **Relaciones uno a muchos (1:N).** una ocurrencia de A se relaciona con muchas ocurrencias de B y una ocurrencia de B solo se relaciona con una ocurrencia de A. Ejemplo: un docente puede tener varias ocurrencias de la entidad asignatura, pero la entidad asignatura solo tendrá una ocurrencia ligada con la entidad docente.
- **Relaciones muchos a uno (N:1).** una ocurrencia de A solo se relaciona con una ocurrencia de B, pero una ocurrencia de B puede relacionarse con muchas ocurrencias de A. Ejemplo: un jugador pertenece a un único equipo, pero a un equipo pueden pertenecer muchos jugadores.
- **Relaciones muchos a muchos (N:N).** Un ejemplar de A puede relacionarse con varias ocurrencias de B y viceversa. Ejemplo: un alumno se matricula en varias asignaturas y en una asignatura pueden estar matriculados varios alumnos.

La representación de las cardinalidad de las relaciones se puede hacer de varias maneras en los esquemas E/R:



5.3. Cardinalidad de entidades

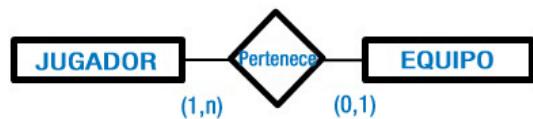
Es el número mínimo y máximo de correspondencias en las que puede tomar parte cada ocurrencia de la entidad.

Dadas las entidades A y B, la participación de A en una relación sería obligatoria (total) si la existencia de cada una de sus ocurrencias necesita como mínimo de una ocurrencia de B. En caso contrario la participación es opcional. La representación gráfica es de una etiqueta tipo (0,1), (1,1), (0,N) o (1,N) indicando los parámetros de cardinalidad mínima y cardinalidad máxima.

- **Cardinalidad mínima.** El número mínimo de asociaciones en las que aparecerá cada ocurrencia de la entidad. Es cero si es opcional o uno si es obligatoria (aunque pueda tener más de 1).
- **Cardinalidad máxima.** El número máximo de relaciones en las que aparecerá cada ocurrencia de entidad. Puede ser 1, otro valor concreto o N).

Ejemplo:

Un jugador, como mínimo puede pertenecer a ningún equipo y como máximo a 1 (0,1). En cambio, a un equipo pertenece como mínimo un jugador y como máximo varios (1,N). **Se representa colocando la etiqueta junto a la entidad con la que se relaciona, es decir, al lado opuesto de la relación.**



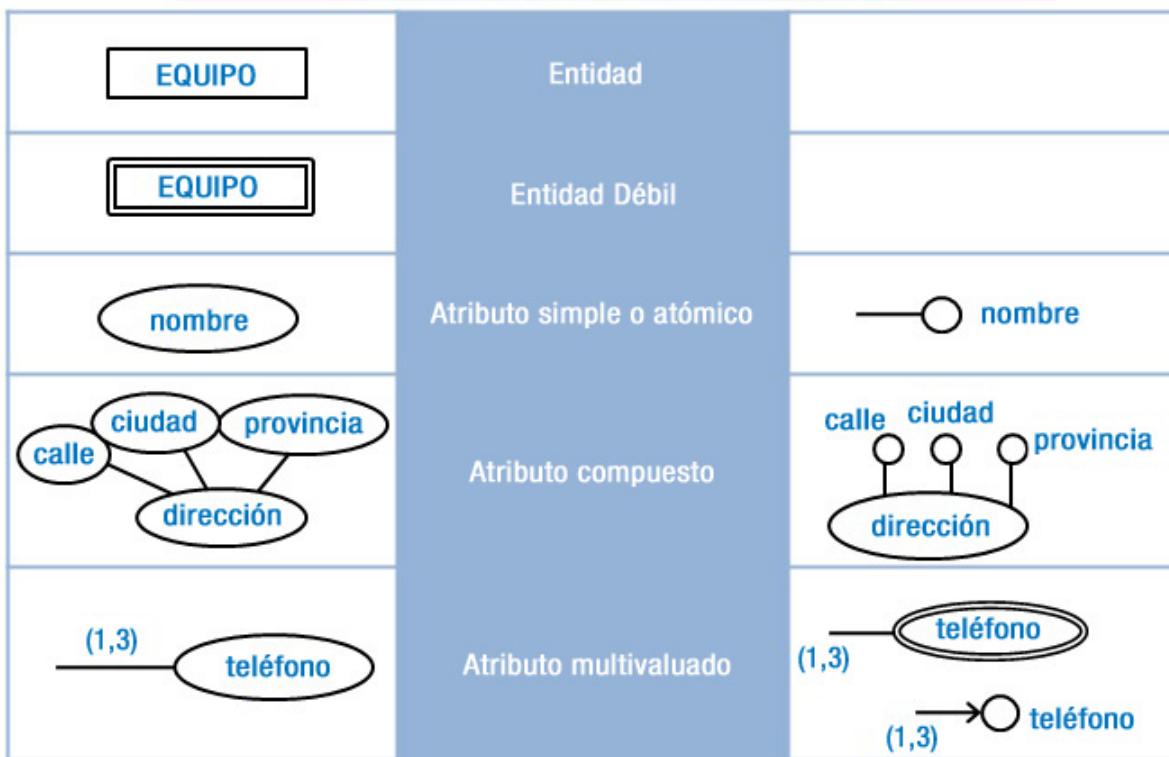
Otras maneras de representar la cardinalidad entre entidades:

Notación alternativa para representar cardinalidad de entidades

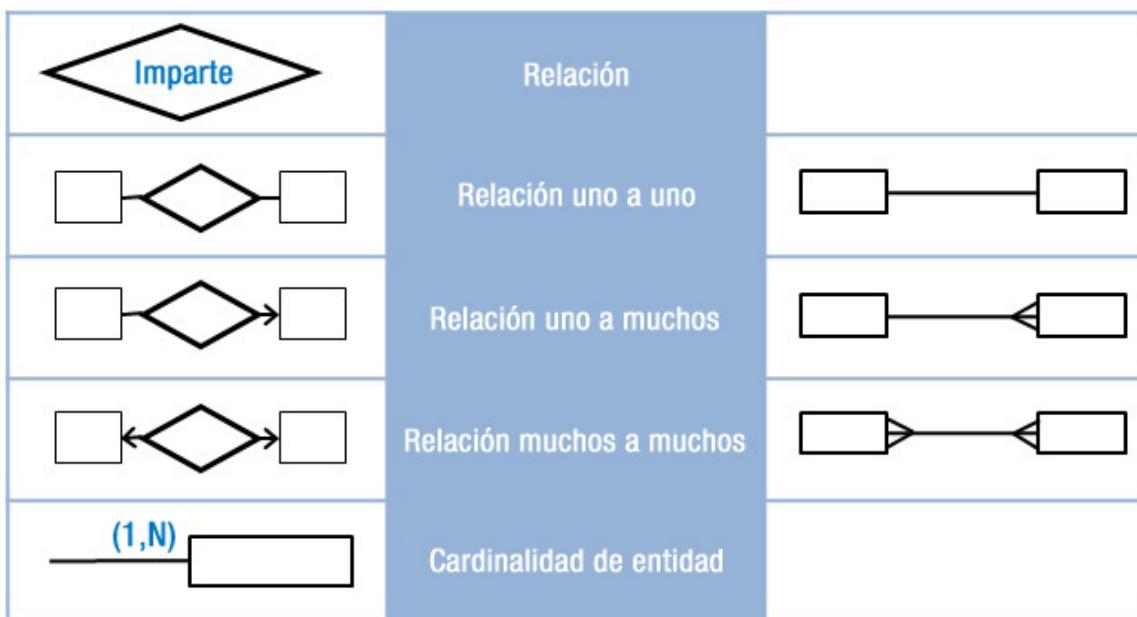
—→	Muchos
—+—	Uno
—○—→	De cero a muchos
—→—○—	De uno a muchos
—○—+—	De cero a uno

6. Resumen de simbología del modelo E/R

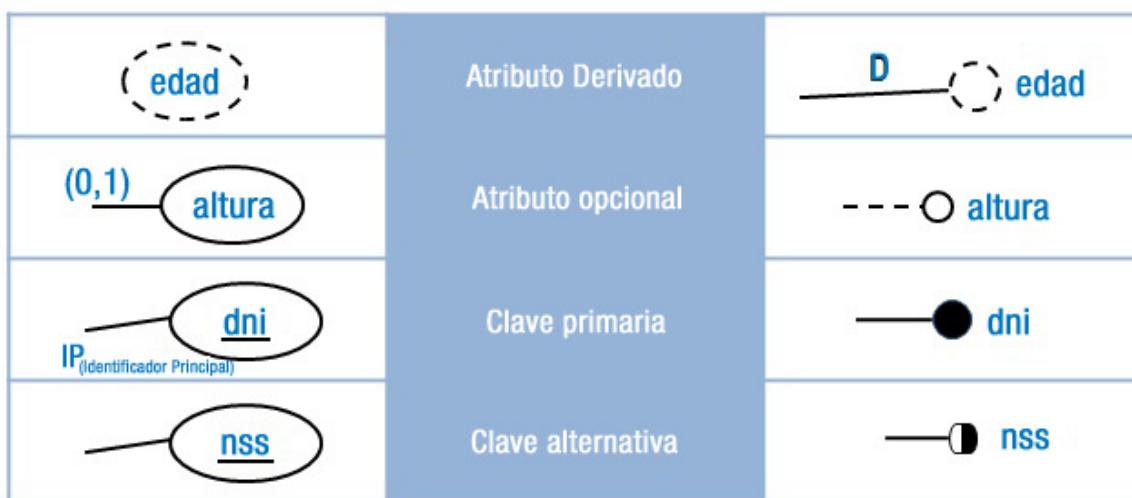
Notaciones del modelo Entidad/Relación



Notaciones del modelo Entidad/Relación



Notaciones del modelo Entidad/Relación



7. El modelo E/R extendido

En el modelo E/R extendido se incorporan nuevas extensiones que permiten mejorar la capacidad para representar circunstancias especiales que el modelo E/R tradicional no puede representar.

7.1. Restricciones en las relaciones

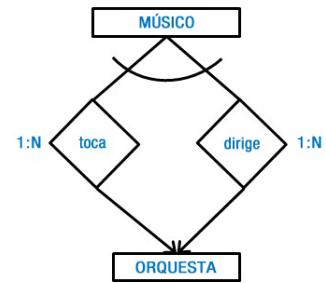
Son restricciones sobre la relación de los ejemplares.

Restricción de exclusividad

Existe si una entidad participa en dos o más relaciones, pero las ocurrencias de esa entidad solo pueden participar en una de las relaciones y no en varias a la vez.

Ejemplo: un músico puede dirigir una orquesta o tocar en ella, pero no puede hacer las dos cosas a la vez. Existen dos entidades (músico y orquesta), existen dos relaciones (dirigir y tocar) y existe relación de exclusividad (o dirige o toca).

La representación es mediante un arco que englobe todas las relaciones.

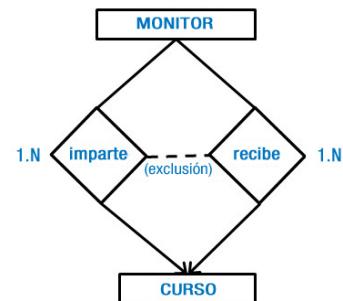


Restricción de exclusión

Esta restricción se produce cuando las ocurrencias de las entidades solo pueden asociarse utilizando una única relación. En el ejemplo del músico y la orquesta, podríamos modificar su funcionamiento diciendo que un músico puede dirigir orquestas y también puede tocar en ellas, pudiendo usar las dos relaciones a la vez, pero lo que no puede hacer es dirigir y tocar en la MISMA ORQUESTA.

Otro ejemplo sería un monitor que puede impartir diferentes cursos orientados a monitores y además también puede recibirlos. Pero lo que no puede hacer es recibir el mismo curso que imparte.

Se representa con una línea discontinua entre las dos relaciones.



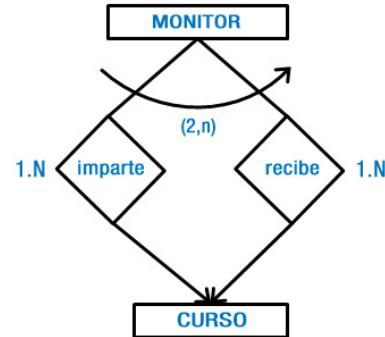
Restricción de inclusividad

Dos ocurrencias de entidad que se asocian a través de una relación no pueden hacerlo hasta que se hayan asociado a través de otra relación.

Por ejemplo. Imaginemos que el requisito para que un monitor imparta un curso en concreto, primero debe haber recibido otros dos cursos. El curso que imparte no tiene por qué ser el mismo que los que previamente ha recibido.

Aplicando esta restricción estaremos indicando que cualquier ocurrencia de una entidad que

participa en una de las relaciones, debe participar obligatoriamente en la otra.



Se representa con un arco acabado en flecha que partirá desde una relación hasta la relación que ha de cumplirse y entre paréntesis pondremos el número mínimo y máximo de ocurrencias de dicha restricción.

Restricción de inclusión

Es una restricción más fuerte que la de inclusividad, haciendo que no se pueda cumplir una relación de las ocurrencias de dos entidades hasta que no se haya cumplido otra relación entre esas mismas ocurrencias.

En el caso anterior, un monitor no podrá impartir un curso en concreto si previamente no lo ha recibido. De modo que toda la ocurrencia de la entidad monitor que esté asociada a una ocurrencia 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.

Se representa con una flecha discontinua desde una relación que depende de que la otra se haya cumplido, hasta la relación que debe cumplirse.

7.2. Generalización y especialización

Son nuevos tipos de relaciones que permiten modelar de una manera más fiel, reciben el nombre de jerarquías y se basan en conceptos de generalización especialización y herencia.



Son entidades que poseen características comunes, permitiendo crear una entidad de nivel más alto que englobe dichas características, que a su vez se divide en entidades que tienen características diferenciadoras. Es un proceso de refinamiento ascendente/descendente y expresan entidades de nivel superior (superclase o supertipo) que engloban a entidades de nivel inferior (subclase o subtipo).

Esto hace que podamos realizar una especialización de una superclase en subclases y a su vez una generalización de subclases en superclases.

La jerarquía se caracteriza por la herencia, los atributos de superclase los hereda la subclase y si una superclase interviene en una relación también lo harán las subclases.



Se representan mediante un triángulo invertido con la notación IS A (es un) dentro de él.

Una generalización/especialización puede tener las siguientes restricciones:

- **Totalidad:** si todo ejemplar de la superclase pertenece a alguna de las subclases.
- **Parcialidad:** si no todos los ejemplares de la superclases pertenecen a alguna de las subclases.
- **Solapamiento:** si un mismo ejemplar de la superclase puede pertenecer a más de una subclase.
- **Exclusividad:** si un mismo ejemplar de la superclase pertenece a una sola subclase.

7.3. Agregación

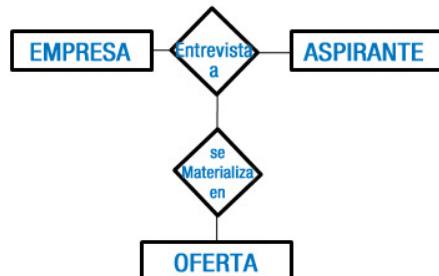
En el modelo de entidad relación tradicional, no se pueden representar relaciones entre relaciones. En este caso la solución pasa por convertir las entidades y su relación en una sola entidad de nivel más alto, siendo utilizada para expresar otra relación con otra entidad (o grupos de entidades+relación).

Por ejemplo. Una empresa selección de personal realiza entrevistas a diferentes aspirantes. Estas entrevistas pueden derivar o no en oferta de

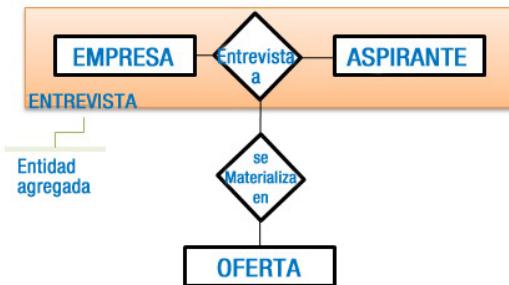
empleo.



Solución 1: Errónea, ya que estaríamos representando que, por cada entrevista realizada por una empresa a un aspirante, se genera una oferta de empleo



Solución 2: Errónea, porque en el modelo E/R no pueden establecerse relaciones entre varias relaciones



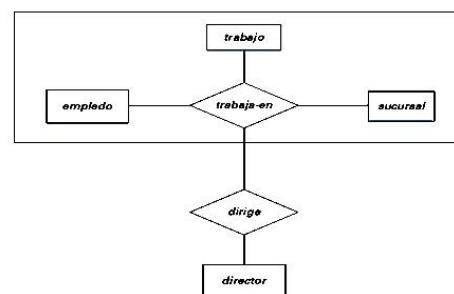
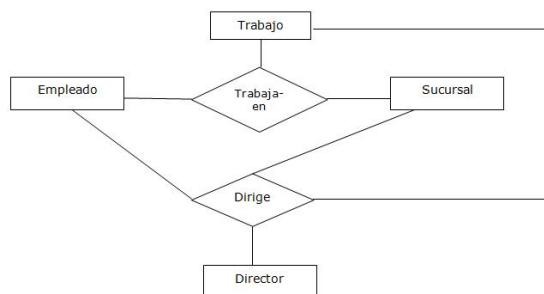
Solución 3: En el modelo E/R Extendido, puede crearse una entidad agregada llamada ENTREVISTA, compuesta por la relación “Entrevista a” que existe entre EMPRESA y ASPIRANTE. Entre esta nueva entidad y OFERTA si puede establecerse una relación “se materializa en”

Otro ejemplo.

Supongamos una entidad Director, la cual dirige a la entidad Empleado, la entidad Trabajo y la entidad Sucursal.

El conjunto de Empleado Trabajo y Sucursal están unidas por la relación trabajo. Este conjunto de entidades se puede relacionar con director formando una relación cuaternaria entre empleado, sucursal, trabajo y director.

Sin embargo, utilizando la agregación se crea una relación binaria dirige entre trabaja-en y director para representar quien dirige las tareas



8. Elaboración de diagramas E/R

Saltase el proceso de fase de diseño de la base de datos y el diagrama E/R supone la pérdida de información de la base de datos. La práctica de estos diagramas es fundamental, a veces hay que retocarlos e incluso rehacerlos.

8.1. Identificación de entidades y relaciones

Las etapas para la creación de diagramas E/R son:

1. **Identificación de entidades:** es un proceso intuitivo, se analiza la especificación de requerimientos en busca de nombres o sustantivos, si estos nombres son objetos importantes del problema seguramente serán entidades. También se ha de identificar que nombres referidos a características, cualidades o propiedades no serán entidades.

Otra forma de identificarlos es localizar los objetos o elementos que existen por sí mismos, acerca de los cuales se quiere guardar información, esa información serán atributos. En ocasiones también se pueden identificar elementos a través de los atributos que lo componen.

Para identificar entidades se deben cumplir 3 reglas:

- Existencia propia.
- Cada ejemplar de una entidad debe poder ser diferenciado del resto.
- Todos los ejemplares de una entidad deben tener las mismas propiedades.

El número de entidades obtenidas debe ser manejable y **se irán identificando con nombres, en mayúscula, singular y representativos de su función.**

2. **Identificación de relaciones.** Utilizaremos verbos o expresiones verbales que conecten las entidades previamente identificadas entre ellas. Se debe aplicar la lógica para no repetir las relaciones. La gran mayoría de las veces las relaciones se establecen entre dos entidades, pero pueden existir relaciones recursivas o unarias y relaciones entre más entidades.

El nombre de la relación debe ser preferiblemente un verbo en minúsculas y representativo del significado o acción de la relación. Si el identificador está compuesto por varias palabras se utilizará guión bajo para unirlas.

El siguiente paso será representar la cardinalidad mínima y máxima de las entidades participantes en cada relación y el tipo de correspondencia (1 a 1, 1 a muchos, o muchos a muchos).

Si hemos encontrado alguna relación recursiva, reflexiva o unaria, representaremos en el esquema los roles desempeñados por la entidad en dicha relación.

8.2. Identificación de atributos, claves y jerarquías

1. **Identificación de atributos.** Sobre el documento de especificación de requerimientos buscaremos nombres relativos a características, propiedades, identificadores o cualidades de las entidades o relaciones. Puede que no todos los atributos queden reflejados en el documento de especificación de requerimientos.

Hay que tener en cuenta si los atributos localizados son simples, compuestos, derivados o calculados y si algún atributo o conjunto de ellos se repite en varias entidades. Si esto último sucede podemos plantear establecer una jerarquía de especialización.

Se les debe asignar un nombre, preferiblemente en minúsculas, representativo de su contenido o función. Además se debe recopilar la siguiente información:

- Nombre y descripción.
- Atributos simples que lo componen (si es compuesto).
- Método de cálculo si es atributo derivado.

2. **Identificación de claves:** se establecen una o varias claves candidatas, se escoge una de ellas como clave primaria, formada por uno o varios atributos que identifican de manera única a cada ocurrencia. Este proceso permitirá identificar si la entidad es fuerte (al menos una clave candidata) o débil (ninguna).
3. **Determinación de jerarquías.** Además de identificar entidades con características comunes generalizadas en una entidad de nivel superior, también se ha de expresar en el esquema las particularidades de diferentes ejemplares de un tipo de entidad, creando subclases de superclases analizando con detenimiento el documento de especificación de requerimientos.

8.3. Metodologías

Son estrategias disponibles para la elaboración de esquemas conceptuales:

- **Metodología descendente (top-down).** A partir de un esquema general se descompone en niveles, cada uno con mayor número de detalles. Se parte de objetos muy abstractos y se van refinando hasta llegar al esquema final
- **Metodología ascendente (bottom-up).** Se parte desde el nivel más bajo, los atributos, se agrupan en entidades y después se crean las relaciones y jerarquías.
- **Metodología dentro-fuera (inside-out).** Se comienza el esquema en una parte del papel y se van completando con entidades y relaciones hasta ocupar todo el documento.
- **Metodología mixta.** Para problemas complejos, se dividen los requerimientos en subconjuntos y se van analizando independientemente. Se crea un esquema estructura en el que se interconectan los subconjuntos creados. Se utiliza la técnica descendente para dividir los requerimientos y en cada subconjunto se usará la ascendente.

Se utiliza la metodología que más útil resulte aplicar e incluso, alguna nueva a partir de las ya existentes.

8.4. Redundancia en diagramas E/R

La redundancia de datos es el almacenamiento de los mismos datos varias veces en diferentes lugares. Generan problemas como:

- Aumento de la carga de trabajo (grabación o actualización de datos varias veces).
- Gasto extra de espacio de almacenamiento.
- Inconsistencia. Los datos que están repetidos pueden contener distintos valores porque se han actualizado en unos sitios y en otros no.

Nuestros diagramas deben controlar la redundancia y para ello debemos analizar el esquema y valorar que puede estar incorporando redundancia. Para ello debemos buscar:

- **Atributos redundantes**, cuyo contenido se calcula en función de otros. Un atributo derivado puede ser origen de redundancia.

- **Ciclos.** Varias entidades unidas circularmente a través de varias relaciones.

En caso de existir debemos tener en cuenta:

- 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 sea el mismo.
- Que si la relación eliminada tenía atributos asociados estos puedan ser asignados a alguna entidad participante del esquema sin perder significado.

No siempre en la existencia de un ciclo estamos ante una redundancia.

No toda redundancia es perjudicial y a veces es conveniente de una manera controlada. Por ejemplo ante la existencia de atributos derivados de cálculo complejo que ralenticen el funcionamiento de la BBDD.

8.5. Propiedades deseables de un diagrama E/R

Debemos intentar materializar la gran mayoría de propiedades para que nuestros diagramas o esquemas tengan mejor calidad. Estas son:

- **Completitud.** Que todos los requerimientos estén representados en dicho diagrama y que cada representación tenga su equivalente en requerimientos.
- **Corrección:** Será correcto si emplean de manera adecuada todos los elementos del modelo E/R.
 - **Corrección sintáctica:** Que no se produzcan representaciones erróneas en el diagrama.
 - **Corrección semántica:** Que signifique lo que realmente está estipulado en requerimientos. Estos errores pueden ser: usar atributos en lugar de una entidad, una entidad en lugar de una relación, mismo identificador para dos entidades o dos relaciones, indicar mal u omitir cardinalidades...
- **Minimalidad:** es mínimo si se puede verificar que al eliminar algún concepto del diagrama se pierde información. Si es redundante no será mínimo.

- **Sencillez:** se representan los requerimientos de manera fácil de comprender.
- **Legibilidad:** Se puede interpretar fácilmente (aspectos estéticos del diagrama).
- **Escalabilidad:** Si es capaz de incorporar cambios derivados de nuevos requerimientos.

9. Primeros pasos del diagrama E/R al modelo relacional

El diagrama E/R permite una gran independencia en cuestiones relativas a la implementación física de la BBDD, la elección del SGBD, aplicaciones, lenguajes de programación o hardware no afectan hasta este momento.

Una vez hemos revisado, modificado y verificado que cumplimos los requerimientos del problema a modelar es hora de pasar al diseño lógico de la bbdd.

Consiste en construir un esquema lógico de la información relativo al problema, basado en un modelo de base de datos concreto, utilizando elementos y características del modelo de datos de la SGBD. Este modelo puede ser: modelo en red, jerárquico y los que se usan hoy en día: relacional y orientado a objetos.



Esta transformación parte del esquema obtenido en la fase de diseño conceptual y requiere aplicar algunas reglas que garanticen equivalencia entre esquema conceptual y lógico.

El siguiente paso es la normalización, que consiste en diseñar de forma correcta la estructura lógica de los datos, usando un conjunto de técnicas que permiten validar esquemas lógicos basados en modelo relacional.

9.1. Simplificación previa de diagramas

Es un conjunto de procedimientos para aplicar a los diagramas E/R, obteniendo una transformación al modelo lógico basado en el modelo relacional, correcto y casi automático:

- Transformación de relaciones n-arias en bianarias.
- Eliminación de relaciones cíclicas.

- Reducción a relaciones jerárquicas (uno a muchos).
- Conversión de entidades débiles en fuertes.

Transformación de atributos compuestos

Los atributos compuestos han de ser descompuestos en atributos simples, ya que el modelo relacional no los permite.

Transformación de atributos multivaluados

Los atributos multivaluados se convierten en entidad relacionada con la entidad que le procede. El modelo relacional no los permite.

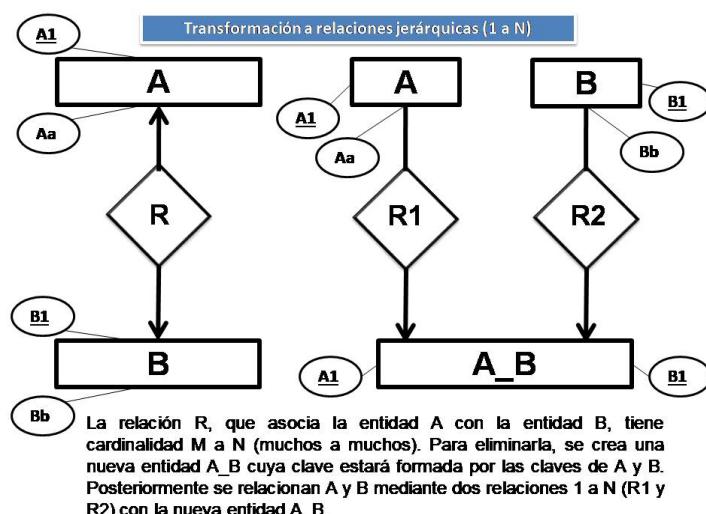
Esta nueva entidad tendrá un único atributo, a no ser que no sea posible que dicho atributo funcione como clave primaria. En ese caso se trata de una entidad débil y debemos ajustar las claves primarias.

Transformación a relaciones jerárquicas

Transformamos las relaciones de cardinalidad muchos a muchos (M/N) a relaciones con cardinalidad uno a muchos. La manera de hacerlo es creando una entidad intermedia con relación 1/N entre las entidades existentes y la nueva entidad. Los atributos pertenecientes a la relación N-Aria pasan a ser parte de la nueva entidad..

Por ejemplo, ante dos entidades A y B relacionadas con cardinalidad N/M, crearemos una tercera entidad llamada A_B, cuya clave estará formada por las claves foráneas de A y B. Posteriormente se relacionan A y B con la nueva entidad A_B mediante relaciones 1/N.

Los atributos que formaban parte de la relación anterior.



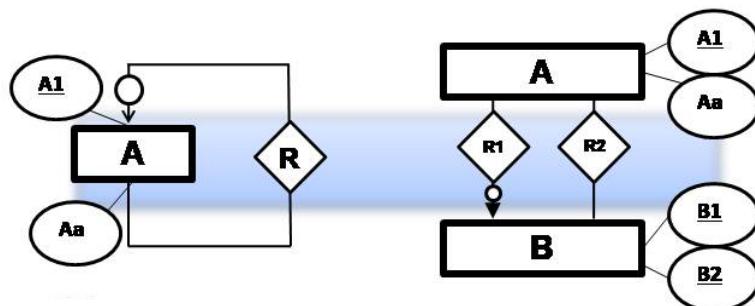
Transformación de relaciones cíclicas

Normalmente, para eliminar transformaciones cíclicas, se crea una nueva entidad, cuya clave estará formada por dos atributos, que contienen las claves de las ocurrencias relacionadas y se crean dos relaciones, cuya cardinalidad depende de la cardinalidad que tenían al principio. Este tipo de conversiones se entiende mejor a la hora de pasarlas al modelo relacional, en el siguiente apartado.

Reflexivas N/M:



Reflexivas 0/N:



Reflexivas 1/1:

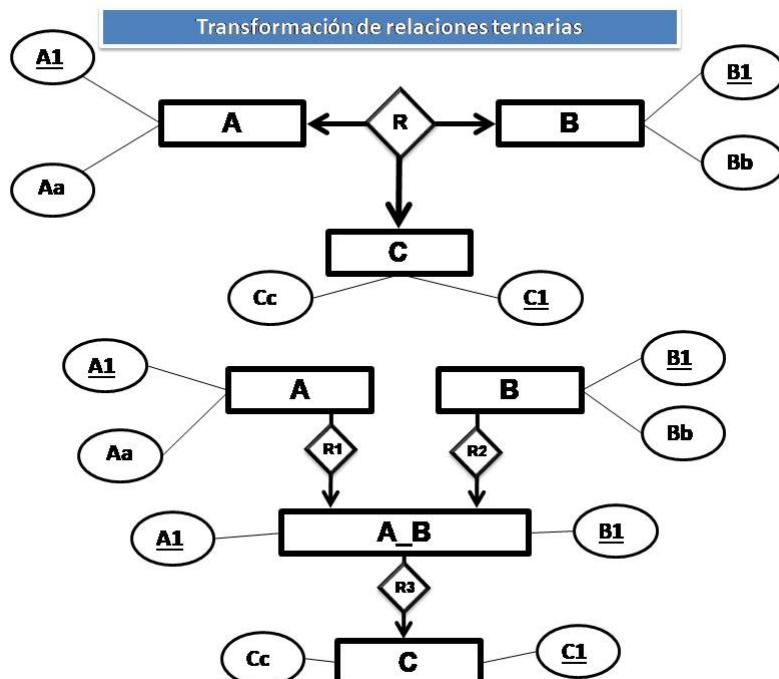


Transformación de relaciones ternarias

Una relación ternaria puede considerarse como una relación binaria a la que se le asocia una entidad. La solución es convertir la relación en una nueva entidad que asocie dos de las entidades y a su vez, crear una nueva relación entre esta nueva entidad y la tercera.

Por ejemplo, entre las entidades A, B y C hay una relación. Esta relación la transformaremos en una entidad llamada A_B, que contendrá las claves de A y B como primarias y las entidades A y B tendrán una relación 1/N con la entidad A_B.

A su vez, esta nueva entidad A_B tendrá una relación 1/N con la entidad C.



10. Paso del diagrama E/R al modelo relacional

Después de la simplificación habremos logrado disponer de un esquema conceptual modificado (ECM), en el cual solo existen entidades fuertes con sus atributos relaciones jerárquicas. Ahora debemos tener en cuenta las siguientes cuestiones:

- Toda entidad se transforma en tabla.
- Todo atributo se transforma en columna de tabla.

- El atributo clave de la entidad se convierte en clave primaria y se representa subrayado en tabla.
- Cada entidad débil genera una tabla que incluirá sus atributos, además de los atributos que son clave primaria de la entidad fuerte con la que esté relacionada. Estos atributos son clave foránea referenciando a la entidad fuerte y por último se escoge una clave primaria de la tabla creada.
- Las relaciones uno a uno pueden generar una nueva tabla o propagar la clave en función de la cardinalidad de las entidades.

Relaciones 1/1 de E/R a esquema relacional

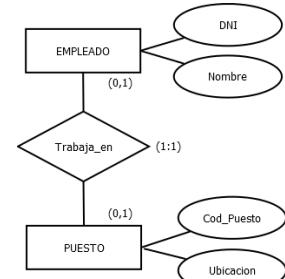
- Si ambas entidades poseen cardinalidades 0,1, la relación se convierte en tabla.

Ejemplo:

EMPLEADO (DNI, Nombre)

EMPLEADO_PUESTO (DNI, Cod_Puesto)

PUESTO (Cod_Puesto, Ubicacion)

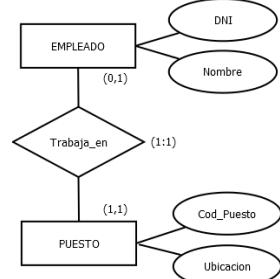


- Si una entidad posee cardinalidad 0,1 y la otra 1,1, se propaga la clave de la entidad 1,1 a la entidad 0,1

Ejemplo:

EMPLEADO (DNI, Nombre)

PUESTO (Cod_puesto, Ubicacion, DNI)



- Si ambas entidades tienen cardinalidad 1,1 se propaga la clave de cualquiera de ellas a la otra tabla. Siguiendo el ejemplo anterior, podría realizarse la misma conversión, o la siguiente:

EMPLEADO (DNI, Nombre, Cod_Puesto)

PUESTO (Cod_puesto, Ubicacion).

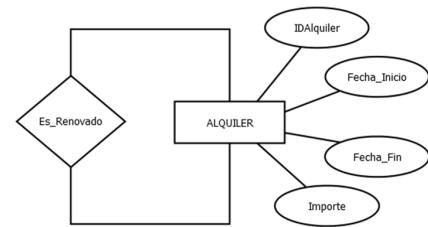
Relaciones reflexivas a esquema relacional

En este tipo de relaciones se debe tener muy en cuenta la cardinalidad. Se pueden dar los siguientes casos.

- **Relación 1 a 1.** No genera tabla nueva. La clave de la entidad se repite pero con identificadores distintos. Un identificador será PK y el otro FK de ella misma.

Ejemplo. Queremos almacenar los distintos alquileres y si los alquileres han sido renovados, almacenar la id del alquiler anterior. Aunque en este ejemplo no se ha identificado la cardinalidad de la entidad, podría asignarse una cardinalidad 0,1, de manera que si dicho alquiler nunca ha sido renovado, no contendrá la ID de ningún alquiler anterior:

ALQUILER (IDAlquiler, Fecha_Inicio, Fecha_Fin, Importe, IDAlquiler_Anterior).



- **Relación 1 a M.** Se pueden dar dos casos.

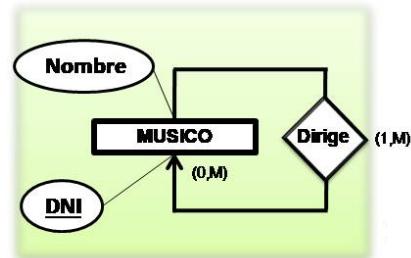
- Si la entidad "Muchos" es siempre obligatoria, actuaremos igual que en una relación 1 a 1.
- Si la entidad "Muchos" no es obligatoria crearemos una nueva tabla cuya clave será la del lado "Muchos" y se propaga hasta esta nueva tabla la clave como FK.

Ejemplo:

MUSICO (DNI, Nombre)

DIRIGE (DNI, DNI_Director)

DNI_Director es el DNI del músico que juega el rol de director y es clave foránea de la tabla MUSICO

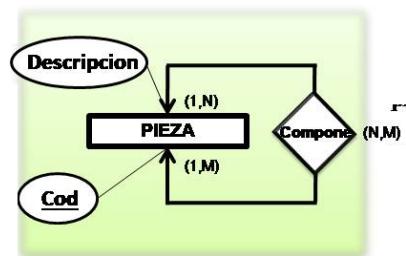


- **Relación N a M.** También se crea una nueva tabla, pero en este caso la PK será la combinación de las dos claves de la otra entidad.

Ejemplo:

PIEZA (Cod_Pieza, Descripcion)
 PIEZA_COMPUESTA (Cod_Pieza,
Cod_Pieza_Comp)

Cod_Pieza es la clave de la pieza en cuestión y Cod_Pieza y Cod_pieza_Comp es la pieza o piezas que la componen. Las dos son PK.



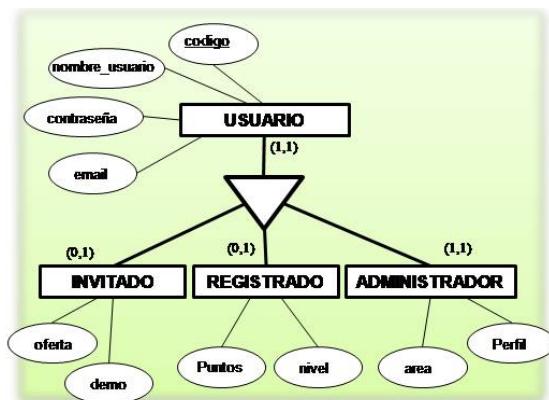
Jerarquías de E/R a esquema relacional

Hay tres formas de tratar este tipo de relaciones:

- Crear una única entidad que aglutine todos los subtipos y sus atributos, esta unión permite simplicidad pero puede provocar valores nulos en atributos propios de cada subtipo.

Ejemplo:

USUARIO (Codigo,
 Nombre_Usuario, Contraseña,
 Email, Oferta, Demo, Puntos,
 Nivel, Area, Perfil).

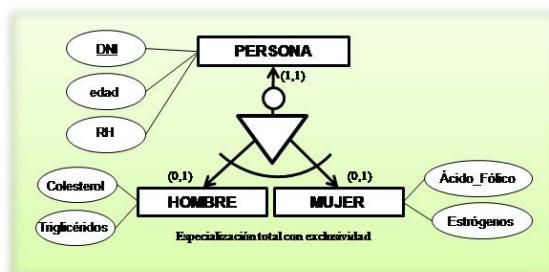


- Anulación del supertipo, de modo que sus atributos pasan a todos los subtipos y las relaciones del supertipo se han de reproducir en cada subtipo. La clave de supertipo pasa a subtipo.

Ejemplo:

HOMBRE (DNI, edad, RH,
 Colesterol, Triglicéridos)

MUJER (DNI, edad, RH,
 Ácido_Fólico, Estrógenos)



- Añadir relaciones 1 a 1 entre el supertipo y los subtipos. Los atributos de supertipo se mantienen y cada uno de los subtipos tendrá clave foránea del supertipo.

Ejemplo:

PERSONA (DNI, edad, RH)

HOMBRE (DNI, Colesterol, Triglicéridos)

MUJER (DNI, Ácido_Fólico, Estrógenos)

Relaciones de muchos a muchos a esquema relacional

Se transforman en una nueva tabla que tendrá como clave primaria la unión de las claves primarias de las entidades que asocia.

Ejemplo:

PRODUCTO (Ref, Descripción)

TRABAJADOR (DNI, Nombre)

ELABORA (Ref; DNI,

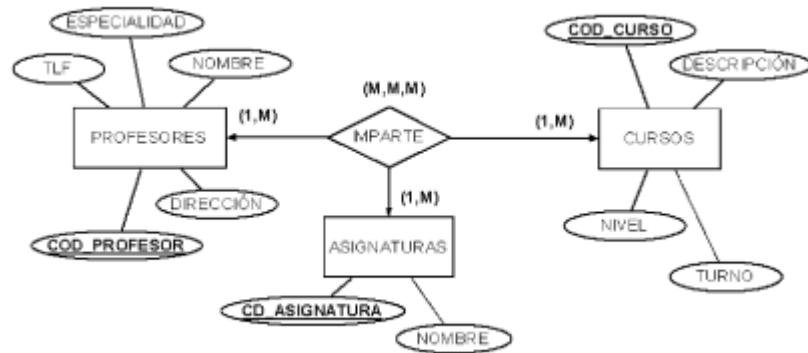
Fecha_Elaboración)



Relaciones N-Arias a esquema relacional

Cada entidad se convierte en tabla y también la relación que contendrá sus atributos propios más las claves de todas las entidades. La clave principal serán todas las claves de entidades. Se dan dos casos dependiendo de la cardinalidad.

- **N:M:N.** Si todas las entidades que participan tienen cardinalidad máxima de muchos, la clave resultante será la unión de todas las entidades que relaciona.



Ejemplo:

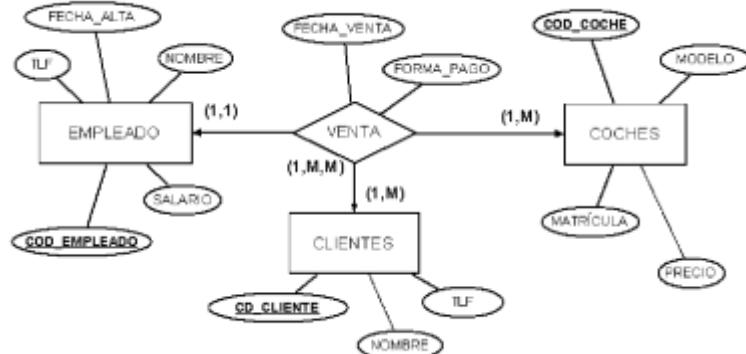
PROFESORES (CodProfesor, Dirección, Nombre, Teléfono Especialidad)

CURSOS (CodCurso, Descripción, Nivel, Turno)

ASIGNATURAS (CodAsignatura, Nombre)

IMPARTE (CodProfesor (FK), CodCurso(FK), CodAsignatura(FK))

- **1:N:M.** En caso de que una de las entidades participe con cardinalidad máxima 1, la clave de esta entidad será un atributo más de la nueva tabla, pero no será PK.



Ejemplo:

CLIENTES (CodCliente, Nombre Teléfono)

EMPLEADO (CodEmpleado, Nombre, Telefono, Salario, FechaAlta)

COCHES (CodCoche, Matrícula, Modelo, Precio)

Venta (CodCoche(FK), CodCliente(FK), CodEmpleado(FK), FormaPago, FechaVenta)

No obstante, Las relaciones N-Arias no deberían existir en nuestro esquema conceptual modificado (ECM) si hemos realizado la tarea de simplificación correctamente.

11. Normalización de modelos relationales

Es el proceso que consiste en imponer a las tablas del modelo relacional una serie de restricciones a través de un conjunto de transformaciones consecutivas. Esto garantiza que las tablas contengan los atributos necesarios y suficientes para describir la realidad de la entidad que representan y permite separar los atributos que por su contenido podrían generar la creación de otra tabla.

La técnica de normalización se utiliza como proceso de refinamiento que debe aplicarse después de lo que conocemos "paso a tablas" o traducción del esquema conceptual al esquema lógico. Este proceso consigue:

- Suprimir dependencias erróneas entre atributos.
- Optimizar procesos de inserción, modificación y borrado de la BBDD.

Esta normalización se aplica de 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 y no se puede pasar a la siguiente hasta que no se satisface completamente.

Estas formas normales son: Primera, Segunda, Tercera Boyce-Codd, Cuarta, Quinta y Dominio-Clave. Cada forma normal es más restrictiva que la anterior.

Es recomendable aplicar la normalización hasta tercera forma normal o hasta Boyce-Codd.

11.1. Tipos de dependencias

Para aplicar las formas normales es necesario conocer este concepto.

Dependencia funcional

Dados los atributos A y B, B depende funcionalmente de A sí, y solo sí, para cada valor de A sólo puede existir un valor de B. A es atributo determinante, ya que determina el valor de B. Su representación es la siguiente: $A \rightarrow B$. A y B podrían ser un solo atributo o conjunto de ellos. Ejemplo. Entre DNI y NOMBRE existe una dependencia funcional: $DNI \rightarrow Nombre$.

Se estudian las DF para encontrar claves candidatas y obtener el mínimo conjunto posible de atributos que sirvan de clave principal.

Dependencia funcional completa

Dados los atributos A₁, A₂,...A_k y B, B depende funcionalmente de forma completa de A₁, A₂,...A_k, si y solo si B depende funcionalmente del conjunto completo de atributos pero no de ninguno de sus posibles subconjuntos.

Ejemplo. Entre DNI.Empresa → Nombre no existe DF completa, sino parcial, porque nombre no depende de empresa. En cambio si que existiría entre los atributos DNI.Empresa→Sueldo.

Las dependencias totales se utilizan para tratar anomalías y solucionarlas y se tratan en la 2^a FN.

Dependencia transitiva

A y C tienen dependencia transitiva si B→C y si A→B

Ejemplo. Tenemos los atributos Num_matrícula, grupo_asignado y aula_grupo con el condicionante de que un alumno solo tiene un grupo asignado y un grupo siempre corresponde a un único aula.

Num_Mat→ Grupo_Asig | Aula_grupo

Grupo_Asig→ Aula_Grupo

El atributo Aula_Grupo es transitivamente dependiente de Num_Mat.

11.2. Formas normales

1^a forma normal

Una tabla está en 1FN o FN1 si, y solo si, todos los atributos de la misma contienen valores atómicos (si los atributos no clave, dependen funcionalmente de la clave).

1. Se crea a partir de la tabla inicial una nueva tabla cuyos atributos son los que presentan dependencia funcional de la clave primaria. Esta tabla estará ya en 1FN.
2. Con los atributos restantes se crea otra tabla y se elige la clave primaria. Comprobamos si esta segunda tabla está en 1FN y si no tomaremos la segunda tabla como tabla inicial y repetimos el proceso.

2^a forma normal

Una tabla está en segunda forma normal si, y solo si, está en 1FN y además todos los atributos que no pertenecen a la clave dependen funcionalmente de forma completa de ella.

1. Se crea a partir de la tabla inicial una nueva tabla con atributos que dependen funcionalmente de forma completa de la clave. La clave será la misma que la clave primaria de la tabla inicial.
2. 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. Si esta tabla está en 2FN, la tabla inicial ya está normalizada y el proceso termina, si no, tomaremos esta segunda tabla como inicial y repetimos el proceso.

3^a forma normal

Una tabla está en tercera forma normal sí y solo sí está en 2FN y además cada atributo que no está en la clave primaria no depende transitivamente de la clave primaria.

1. Se crea a partir de la tabla inicial una nueva tabla con los atributos que no poseen dependencia transitiva de la clave primaria.
2. Con los atributos restantes se crea otra tabla con los dos atributos no clave que intervienen en la dependencia transitiva, se elige uno como clave primaria si cumple requisitos. Se comprueba si la tabla está en 3FN, si es así, la tabla inicial está normalizada y el proceso termina, si no, tomamos la segunda tabla como inicial y repetimos el proceso.

Forma normal de Boyce Codd

Una tabla está en FNBC o BCFN si, y solo si, está en 3FN y todo determinante es clave candidata. Un determinante será todo atributo simple o compuesto del que depende funcionalmente de forma completa algún otro atributo de la tabla.

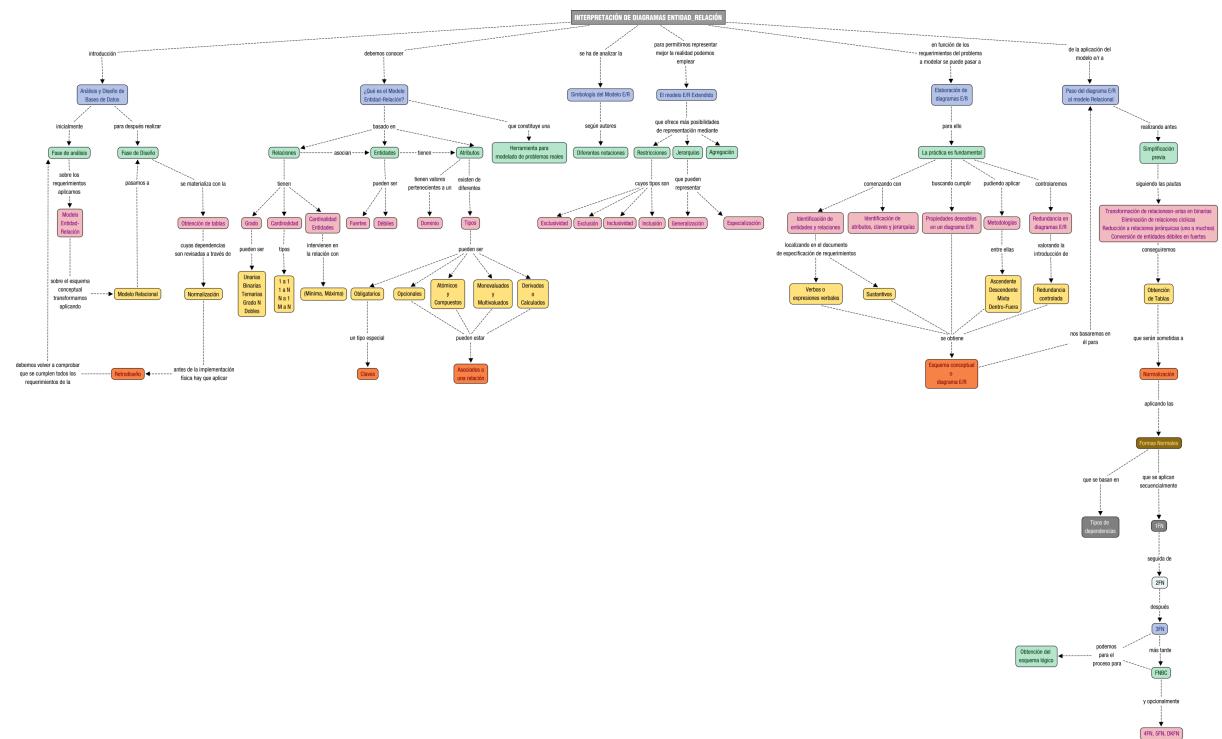
Aquellas tablas en las que sus atributos forman parte de la PK estarán en FNBC. Si encontramos un determinante que no es clave candidata, la tabla no estará en FNBC.

Tendremos que descomponer la tabla inicial en dos, siendo cuidadosos para evitar pérdida de información.

Otras formas normales

La 4FN se basa en el concepto de dependencias multivaluadas, la 5FN en las dependencias de Join o reunión y la DKFN en restricciones impuestas sobre dominios y claves.

Mapa Conceptual





4. Realización de consultas.

Class	Bases de datos
Column	Xerach Casanova
Last Edited time	@Dec 28, 2020 8:53 AM

- 1. Introducción
- 2. La sentencia SELECT
 - 2.1. Cláusula SELECT
 - 2.2. Cláusula FROM
 - 2.3. Cláusula WHERE
 - 2.4. Cláusula ORDER BY
- 3. Operadores
 - 3.1. Operadores de comparación
 - 3.2. Operadores aritméticos y de concatenación
 - 3.3. Operadores lógicos
 - 3.4. Precedencia
- 4. Consultas calculadas
- 5. Funciones
 - 5.1. Funciones numéricas
 - 5.2. Funciones de cadena de caracteres
 - 5.2. Funciones de manejo de fechas
 - 5.4. Funciones de conversión
 - 5.2. Otras funciones: NVL y DECODE
- 6. Consultas de resumen
 - 6.1. Funciones de agregado: SUM y COUNT
 - 6.2. Funciones de agregado: MIN y MAX
 - 6.3. Funciones de agregado: AVG, VAR y STDEV
- 7. Agrupamiento de registros
- 8. Consultas multitableas
 - 8.1. Composiciones internas
 - 8.2. Composiciones externas
 - 8.3. Composiciones en la versión SQL99
- 9. Otras consultas multitableas: Unión, intersección y la diferencia de consultas
- 10. Subconsultas

1. Introducción

SQL es el lenguaje utilizado por la mayoría de las aplicaciones donde se trabaja con datos, para acceder a ellos, crearlos y actualizarlos.

Nació a partir de la publicación "A relational model o data for large shared data banks" de Edgar Frank Codd. IBM creó un primer lenguaje llamado SEQUEL (Structured English Query Language), que con el tiempo evolucionó a SQL (Structured Query Language). En 1979 Relational Software (actualmente Oracle Corporation), saca al mercado la primera implementación comercial de SQL.

En 1992 ANSI e ISO estandarizan SQL definiendo las sentencias básicas de SQL (SQL92 o ANSI-SQL). Todas las bases de datos comerciales cumplen con ese estándar, incluyendo sus propias mejoras.

La primera fase del trabajo comienza con sentencias DDL (Lenguaje de definición de datos), para crear la estructura de la base de datos: tablas.

La siguiente fase es la de manipular los datos con sentencias DML (lenguaje de manipulación de datos). Consta de 4 sentencias básicas: INSERT, DELETE, UPDATE y SELECT.

2. La sentencia SELECT

La sentencia SELECT sirve para recuperar o seleccionar datos, de una o varias tablas. Consta de cuatro partes básicas:

- Cláusula SELECT seguida de la descripción de lo que se desea ver (columnas), separadas por comas simples. Obligatoria.
- Cláusula FROM seguida del nombre de la tabla o tablas de las que proceden las columnas del SELECT. Obligatoria.
- Cláusula WHERE seguida de un criterio de selección condición. Opcional.
- Cláusula ORDER BY seguida de un criterio de ordenación. Opcional.

```
SELECT [ALL | DISTINCT] columna1, columna2, ... FROM tabla1, tabla2, ... WHERE condición1, condición2, ... ORDER BY ordenación;
```

Las cláusulas ALL (muestra todas las filas aunque estén repetidas) y DISTINCT (suprime las filas que tengan igual valor que otras), son opcionales.

2.1. Cláusula SELECT

Se debe tener en cuenta lo siguiente:

- Se pueden nombrar las columnas anteponiendo el nombre de la tabla. Es opcional si no existe el mismo nombre de columna en dos tablas sobre las que se está realizando el SELECT. Ejemplo: NombreTabla.NombreColumna.
- Se usa el asterisco para incluir todas las columnas de una o varias tablas: SELECT * FROM NombreTabla;
- Podemos poner alias a los nombres de las columnas poniendo entre comillas el alias que demos a la columna: SELECT F_Nacimiento "Fecha de Nacimiento" FROM USUARIOS;
- Se puede sustituir el nombre de la columna por constantes, expresiones o funciones SQL: SELECT 4*3/100 "MiExpresión", Password FROM USUARIOS;

2.2. Cláusula FROM

En la cláusula FROM se definen los nombres de las tablas de las que proceden las columnas, si se utiliza más de una tabla, deben aparecer separadas por comas y se le denomina consulta combinada o join, para las cuales se necesita usar la cláusula WHERE. Ejemplo: SELECT LOGIN,NOMBRE,APELLIDOS FROM USUARIOS;

También puedes añadir el nombre del usuario que es propietario de las tablas: USUARIO.TABLA. Ya que a veces las tablas pueden tener el mismo nombre, perteneciendo a usuarios distintos.

Se pueden asociar alias a tablas para abreviar, en este caso no es necesario entrecomillar. Ejemplo: SELECT * FROM USUARIOS U;

2.3. Cláusula WHERE

Sirve para restringir la selección a un subconjunto de filas, para lo cual debemos especificar una condición que deben cumplir aquellos registros que queremos seleccionar.

El criterio de búsqueda puede ser más o menos sencillo y se pueden conjugar operadores de diversos tipos, funciones o expresiones.

```
SELECT nombre, apellidos  
FROM USUARIOS  
WHERE sexo = 'M';
```

2.4. Cláusula ORDER BY

Se utiliza para especificar un criterio de ordenación en la respuesta a nuestra consulta. Solo se puede ordenar por campos de tipo carácter, número o fecha.

```
SELECT [ALL | DISTINCT] columna1, columna2, ...
FROM tabla1, tabla2, ...
WHERE condición1, condición2, ...
ORDER BY columna1 [ASC | DESC], columna2 [ASC | DESC], ..., columnaN [ASC | DESC];
```

El tipo de ordenación se incluye con las palabras reservadas ASC o DESC. Por defecto la ordenación es ascendente.

Se puede ordenar por más de una columna. También se puede ordenar a través de expresiones creadas con columnas, funciones o constantes.

```
SELECT nombre, apellidos
FROM USUARIOS
ORDER BY apellidos, nombre;
```

También se puede colocar el número de orden del campo por el que quieras ordenar, en vez de indicar el nombre de la columna, si se coloca un número mayor al número de campos en el select, nos dará error.

```
SELECT nombre, apellidos, localidad
FROM usuarios
ORDER BY 3;
```

3. Operadores

Los operadores en SQL se dividen en:

- Relacionales o de comparación.
- Aritméticos.
- De concatenación.
- Lógicos.

3.1. Operadores de comparación

Nos permiten comparar expresiones, que pueden ser valores concretos, campos, variables, etc.

Si el resultado de la comparación es correcto la expresión se considera verdadera, en caso contrario se considera falsa.

OPERADOR	SIGNIFICADO
=	Igualdad.
!=, <>, ^=	Desigualdad. Distinto
<	<
>	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.

NULL significa valor inexistente o desconocido, es tratado distinto a otros valores. Si queremos verificar un valor null, se debe utilizar IS NULL O IS NOT NULL y devolverá verdadero o falso.

Los valores nulos se representan en primer lugar en ORDER BY de manera ascendente y en último lugar en descendente.

```
SELECT nombre FROM EMPLEADOS WHERE SALARIO > 1000;
```

```
SELECT nombre FROM EMPLEADOS WHERE APELLIDO1 LIKE 'R %';
```

Nombres de empleados que empiecen por R.

3.2. Operadores aritméticos y de concatenación

Con los operadores aritméticos es posible obtener salidas en las cuales una columna sea el resultado de un cálculo y no un campo de una tabla.

En este ejemplo seleccionamos el salario aumentado un 5% de los empleados que cobran menos de 1000.

```
SELECT SALARIO*1,05
FROM TRABAJADORES
WHERE SALARIO<=1000;
```

Una expresión aritmética sobre null devuelve null.

Para concatenar cadenas existe el operador de concatenación ||. Oracle puede convertir automáticamente valores numéricos a cadena para una concatenación.

En este ejemplo mostraremos los dos apellidos juntos:

```
SELECT Nombre, Apellido1 || Apellido2
FROM EMPLEADOS;
```

Podemos poner un alias al resultado de la concatenación para que se utilice como cabecera en la salida. En el siguiente ejemplo, además, colocamos un espacio entre los dos apellidos:

```
SELECT Nombre, Apellido1 || ' ' ||Apellido2 Apellidos  
FROM EMPLEADOS;
```

3.3. Operadores lógicos

Se utilizan para comprobar si se cumple una u otra condición, ninguna o varias de ellas.

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.

```
SELECT empleado_dni  
FROM HISTORIAL_SALARIAL  
WHERE salario <=800 OR salario>2000;
```

3.4. Precedencia

Para saber que expresiones se evalúan primero en una consulta, se debe conocer el orden de precedencia en Oracle. En casos de igualdad se evalúa de izquierda a derecha.

1. multiplicación (*) y división (/) al mismo nivel
2. A continuación sumas (+) y restas (-).
3. Concatenación (||).
4. Todas las comparaciones (<, >, ...).
5. Después evaluaremos los operadores IS NULL, IS NOT NULL, LIKE Y BETWEEN
6. NOT
7. AND
8. OR

Para variar el orden se debe trabajar con paréntesis.

4. Consultas calculadas

Son operaciones realizadas con algunos campos para obtener información derivada de ellos. Para ello haremos uso de la creación de campos calculados.

Estos campos calculados se obtienen a través de la sentencia SELECT, poniendo a continuación la expresión que queramos. Esta consulta no modifica los valores originales de la columna o de la tabla, solo se muestra una columna nueva, a la cual se le puede añadir un alias añadiendo detrás de la expresión la palabra AS y el alias.

```
SELECT Nombre, Credito, Credito + 25 as CreditoNuevo  
FROM USUARIOS;
```

5. Funciones

Las funciones son operaciones que se realizan sobre los datos y que realizan un determinado cálculo. Para ello se necesitan parámetros de entrada o argumentos y en función de estos, se realiza el cálculo de la función utilizada. Los parámetros se especifican entre paréntesis.

Se pueden incluir en las cláusulas SELECT, WHERE Y ORDER BY

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

Se pueden anidar funciones dentro de funciones y existen una gran variedad para cada tipo de datos:

- numéricas
- de cadenas de caracteres
- de manejo de fechas
- de conversión
- otras...

Oracle tiene una tabla en la que se pueden hacer pruebas, esta tabla se llama Dual y contiene un campo llamado DUMMY con una sola fila.

5.1. Funciones numéricas

- **ABS(n)** - Calcula el valor absoluto de n.

```
SELECT ABS(-17) FROM DUAL; -- Resultado: 17
```

- **EXP(n)** - Calcula 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 en 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 en 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** - Si el argumento n es positivo retorna 1, si es negativo devuelve -1 y si es 0 devuelve 0

```
SELECT SIGN(-23) FROM DUAL; -- Resultado: -1
```

5.2. Funciones de cadena de caracteres

Se utilizan para manipular campos de tipo carácter o 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('0') FROM DUAL; --Resultado: 79
```

- **CONCAT (cad1, cad2)** - Devuelve las dos cadenas concatenadas o unidas, es equivalente al operador ||.

```
SELECT CONCAT('Hola', 'Mundo') FROM DUAL; --Resultado: HolaMundo
```

- **LOWER(cad)** - Devuelve la cadena cad con todos los caracteres en minúsculas.

```
SELECT LOWER('En MINÚSCULAS') FROM DUAL; --Resultado: en minúsculas
```

- **UPPER(cad)** - Devuelve la cadena cad con todos los caracteres en mayúsculas.

```
SELECT UPPER('En MAYÚSCULAS') FROM DUAL; --Resultado: EN MAYÚSCULAS
```

- **LOWER Y UPPER** son muy utilizadas, sobre todo para comparar cadenas de caracteres de las que desconocemos si están en mayúsculas o minúsculas:

```
SELECT * FROM JUEGOS WHERE UPPER(NOMBRE)='AJEDREZ';
SELECT * FROM JUEGOS WHERE LOWER(NOMBRE)='ajedrez';
```

- **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)** - Obtiene una subcadena de una cadena, 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 izquierda y derecha, así como los espacios dobles del interior de cad.

```
SELECT TRIM(' Hola de nuevo ') FROM DUAL; --Resultado: Hola de nuevo
```

- **LTRIM(cad)** - Elimina los espacios a la izquierda de cad.

```
SELECT LTRIM(' Hola') FROM DUAL; --Resultado: Hola
```

- **RTRIM(cad)** - Elimina los espacios a la derecha de 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 a partir de 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
```

5.2. Funciones de manejo de fechas

Oracle tiene dos tipos de datos para manejar fechas:

- DATE almacena fechas concretas incluyendo a veces la hora.
- TIMESTAMP almacena un instante de tiempo más concreto, hasta fracciones de segundo.

Se pueden realizar operaciones numéricas con fechas:

- Sumar números suma días, si ese número tiene días, se suman días, horas, minutos y segundos.
- Restar números.

- La diferencia o resta entre dos fechas nos dará el número de días entre esas fechas.

En Oracle tenemos las siguientes funciones

- **SYSDATE** - Devuelve fecha y hora actuales

```
SELECT SYSDATE FROM DUAL; --Resultado: 15/08/20
```

- **SYSTIMESTAMP** - Devuelve fecha y hora actuales en formato TIMESTAMP

```
SELECT SYSTIMESTAMP FROM DUAL; --Resultado: 15/08/20 11:40:41,969000 +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 fecha 2.

```
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 en 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, puede ser texto o número de la semana 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: 8
```

En Oracle los operadores aritméticos de más y menos se pueden emplear en fechas:

```
SELECT SYSDATE - 5 FROM DUAL; -- Devuelve la fecha correspondiente a 5 días antes de la fecha actual
```

```

SQL> SELECT SYSDATE HOY FROM DUAL;
HOY
-----
15/08/20

SQL> SELECT SYSTIMESTAMP INSTANTE FROM DUAL;
INSTANTE
-----
15/08/20 11:53:47,879000 +02:00

SQL> SELECT ADD_MONTHS('27/07/20', 5) FROM DUAL;
ADD_MONTH
-----
27/12/20

SQL> SELECT MONTHS_BETWEEN('12/07/20','12/03/20') FROM DUAL;
MONTHS_BETWEEN('12/07/20','12/03/20')
-----
4

SQL> SELECT LAST_DAY('27/07/20') FROM DUAL;
LAST_DAY
-----
31/07/20

SQL> SELECT NEXT_DAY('31/12/20','LUNES') FROM DUAL;
NEXT_DAY
-----
04/01/21

SQL> SELECT EXTRACT(MONTH FROM SYSDATE) FROM DUAL;
EXTRACT(MONTHFROMSYSDATE)
-----
8

SQL> SELECT SYSDATE - 5 FROM DUAL;
SYSDATE-
-----
10/08/20

SQL>

```

5.4. Funciones de conversión

Se puede pasar un tipo de dato a otro, Oracle convierte automáticamente datos de manera que el resultado de una expresión tenga sentido, pero a veces debemos realizar conversiones de modo explícito:

- **TO_NUMBER(Cad_Formato)** - Convierte textos en números con formato:

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 suele utilizar con fechas.
- **TO_DATE(cad, formato)** - Convierte textos a fechas con el formato que queramos.

Para ambas funciones con fechas usamos los siguientes formatos:

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

5.2. Otras funciones: NVL y DECODE

Cualquier operación con NULL devuelve NULL, de esta manera, cuando se divide por cero, también se devuelve nulo.

El resultado de una función también puede dar nulo. Las funciones con nulos nos permiten hacer algo en caso de que aparezca un valor nulo.

- **NVL(valor, expr1)** - Si el valor es nulo, devuelve expr1, el cual debe ser del mismo tipo que el valor.

```
SELECT NVL(correo, 'No tiene correo') FROM USUARIOS;
```

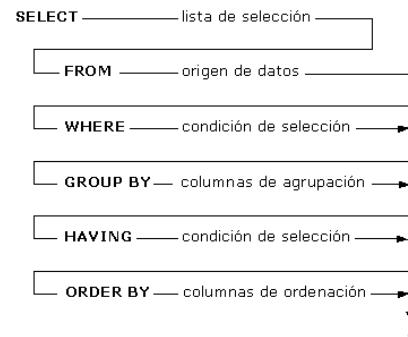
- **DECODE(expr1, cond1, valor1 [, cond2, valor2,...], default)** - Evalúa una expresión expr1 y si se cumple la primera condición cond1 devuelve el valor1, en caso contrario evalúa la siguiente condición hasta que una de ellas se cumpla, si no, devuelve el valor por defecto.

6. Consultas de resumen

La sentencia SELECT nos permite obtener resúmenes de datos de modo vertical con las cláusulas específicas GROUP BY y HAVING. Además tenemos unas funciones llamadas de agrupamiento o de agregado que indican qué cálculos queremos realizar sobre la columna, las cuales toman un grupo de datos de una columna y producen un único dato que resume al grupo

El resultado de estas consultas no corresponde a ningún valor de la tabla, sino a un total calculado sobre los datos de la tabla.

El simple hecho de utilizar una función de agregado en una consulta la convierte en consulta de resumen.



Las funciones tienen una estructura muy parecida: **FUNCIÓN ([ALL | DISTINCT]) Expresión:**

- **ALL** - toma todos los valores de la columna (valor por defecto).
- **DISTINCT** - Se consideran las repeticiones del mismo valor como uno solo.
- El grupo de valores sobre el que se actúa determina el resultado de la expresión con el nombre de una columna o una expresión basada en una o varias columnas. En la expresión no puede aparecer una función de agregado o una subconsulta.
- Todas las funciones aplicadas a filas del origen de datos se aplican una vez ejecutada la cláusula WHERE si existiera.
- Todas las funciones excepto COUNT ignoran valores NULL.
- No se pueden anidar funciones de este tipo. Podemos encontrarlas dentro de una lista de selección en cualquier sitio donde pueda aparecer el nombre de una columna.
- No se pueden mezclar funciones de columna con nombres de columna ordinarias.

6.1. Funciones de agregado: SUM y COUNT

Son funciones para sumar los datos de una columna o contar sus filas.

- **Función SUM**

SUM([ALL|DISTINCT] expresión) - Devuelve la suma de los valores de la expresión, se utiliza en columnas numéricas, el resultado es del mismo tipo aunque puede tener una precisión mayor.

```
SELECT SUM( credito) FROM Usuarios;
```

- **Función COUNT**

COUNT([ALL|DISTINCT] expresión) - Cuenta los elementos de un campo. Expresión es el nombre del campo que queremos contar. Los operandos de expresión pueden incluir el nombre del campo, una constante, una función o el carácter *.

Puede contar cualquier tipo de datos, ya que no tiene en cuenta los valores almacenados. No cuenta los valores NULL a menos que la expresión tenga el comodín asterisco.

```
SELECT COUNT(nombre) FROM Usuarios;
SELECT COUNT(*) FROM Usuarios;
```

6.2. Funciones de agregado: MIN y MAX

- **Función MIN**

MIN ([ALL| DISTINCT] expresión)

Devuelve el valor mínimo sin contar valores nulos. En expresión se incluye el nombre de una tabla, constante o función, pero no otras funciones.

```
SELECT MIN(credito) FROM Usuarios;
```

- **Función MAX:**

MAX ([ALL| DISTINCT] expresión)

Exactamente el mismo funcionamiento que MIN, pero devolviendo el valor máximo.

6.3. Funciones de agregado: AVG, VAR y STDEV

Obtienen datos estadísticos a partir de los datos de nuestra base de datos.

- **Función AVG:**

AVG ([ALL| DISTINCT] expresión)

Devuelve el promedio o media de los valores de un grupo, se omiten los valores null. La expresión puede ser nombre de columna o expresión basada en una o varias columnas de la tabla. Se aplica a campos numéricos y el resultado puede variar según necesidades del sistemas para representar el valor.

```
SELECT AVG(CREDITO) FROM USUARIOS;
```

- **Función VAR**

VAR ([ALL| DISTINCT] expresión)

Devuelve la varianza estadística de todos los valores de la expresión, solo admite columnas numéricas y omite valores null.

- **Función STDEV**

STDEV ([ALL| DISTINCT] expresión)

Devuelve la desviación típica estadística de todos los valores de la expresión. solo admite columnas numéricas y omite valores null.

7. Agrupamiento de registros

En muchas ocasiones, utilizaremos consultas de resumen agrupados según un determinado campo.

En estos casos en lugar de una única fila de resultados, necesitaremos una fila por cada agrupamiento.

Por ejemplo, de una tabla EMPLEADOS en la que se guarda sueldo y actividad, podemos obtener el valor medio del sueldo en función de la actividad realizada en la empresa.

Estos subtotales se obtienen con la cláusula GROUP BY y podemos poner condiciones a esos grupos con la cláusula HAVING.

```
SELECT columna1, columna2, ...
FROM tabla1, tabla2, ...
[WHERE condición1, condición2, ...]
[GROUP BY columna1, columna2, ...]
[HAVING condición ]]
[ORDER BY ordenación];
```

En la cláusula GROUP BY colocamos las columnas por las que vamos a agrupar y en la cláusula HAVING se especifica la condición que han de cumplir los grupos. El orden de las cláusulas siempre será:

1. WHERE - Filtramos filas a partir de las condiciones que pongamos.
2. GROUP BY - Crea una tabla de grupos nueva.
3. HAVING - Filtra los grupos.
4. ORDER BY - Ordena la salida.

Las columnas que aparecen en SELECT y no aparecen en GROUP BY deben tener una orden de agrupamiento para que no se produzca error.

```
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 Sevilla y Badajoz nos quedaría:

```
SELECT provincia, SUM(credito) FROM Usuarios
GROUP BY provincia
HAVING UPPER(provincia) = 'SEVILLA' OR UPPER(provincia)= 'BADAJOZ';
```

8. Consultas multitablas

Es frecuente consultar datos que se encuentren en distintas tablas en la misma sentencia SELECT, lo cual permite realizar distintas operaciones como son:

- La composición interna.
- La composición externa.

```
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)]
```

Por ejemplo: Disponemos de una tabla USUARIOS cuya clave principal es Login y está relacionada con la tabla PARTIDAS a través del campo Cod_Creador_Partida. Si queremos obtener el nombre de los usuarios y las horas de las partidas de cada jugador, necesitamos coger los datos de ambas tablas (las horas se guardan en la tabla partida), debiendo coger filas de una y otra tabla. También podríamos tener una tabla de usuarios en servidores distintos y quisiéramos unirlas.

8.1. Composiciones internas

Cuando combinamos dos o más tablas, el resultado es un producto cartesiano, dando como resultado la combinación de todas las filas de esas dos tablas, relaciona una fila de una tabla con todas y cada una de las filas de otra tabla, aunque no tengan relación ninguna.

Se obtiene poniendo en la cláusula FROM las tablas que queramos poner, separadas por comas.

Hay que tener en cuenta que el resultado es la suma de todas las combinaciones posibles y nos podemos encontrar con una operación muy costosa a medida que se aumenta el número de tablas o de filas.

Es necesario discriminar para que aparezcan filas de una tabla que estén relacionadas con la otra (asociar tablas - JOIN)

Lo importante en las composiciones internas es emparejar campos que han de tener valores iguales, con las siguientes reglas:

- Se combinan tantas tablas como se deseé.
- El criterio de combinación puede estar formado por más de una pareja de columnas.
- La cláusula SELECT puede citar columnas de ambas tablas, condicionen o no la combinación.
- Si hay columnas del mismo nombre en distintas tablas, se identifican especificando la tabla de procedencia seguida de un punto o utilizando alias.

Las columnas relacionadas de la cláusula WHERE se denominan columnas de join o emparejamiento. No tienen que estar incluidas en la lista de selección. Emparejamos tablas que estén relacionadas entre sí utilizando la

clave principal y la ajena.

NombreTabla1.Camporelacionado1 = NombreTabla2.Camporelacionado2.

También se puede combinar una tabla consigo misma, pero se debe crear un alias a uno de los nombres de la tabla que se va a repetir.

En el siguiente ejemplo obtenemos el historial laboral de los empleados, incluyendo nombre y apellidos, fecha en la que entraron a trabajar y fecha de fin de trabajo:

```
SELECT Nombre, Apellido1, Apellido2, Fecha_inicio, Fecha_fin  
FROM EMPLEADOS, HISTORIAL_LABORAL  
WHERE HISTORIAL_LABORAL.Empleado_DNI= EMPLEADOS.DNI;
```

Y en la siguiente obtenemos el historial con nombres de departamento, nombre y apellidos del empleado.

```
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;
```

8.2. Composiciones externas

Sirven para seleccionar filas de una tabla aunque no tengan correspondencia con las filas de otra.

Por ejemplo. En una base de datos donde tenemos empleados de la empresa (Cod_empleado, Nombre, Apellidos, salario y Cod_dpto) por otro lado los departamentos (Codigo_dep, Nombre), en un momento dado se incluyen varios departamentos más a los que todavía no se le han asignado empleados. Si tenemos que obtener un informe de los empleados por departamento y queremos que aparezcan los datos de esos departamentos aunque no tengan empleados usamos composiciones externas.

Para ello se añade un signo (+) entre paréntesis en la igualdad entre campos que ponemos en la cláusula WHERE. El carácter (+) irá detrás del nombre de tabla en la que deseamos aceptar valores nulos.

En el ejemplo sería: Cod_dpto (+)= Codigo_dep (en la tabla empleado es donde aparecerán valores nulos).

8.3. Composiciones en la versión SQL99

CROSS JOIN. Crea un producto cartesiano de las filas de ambas tablas, olvidándose de la cláusula WHERE

NATURAL JOIN: Detecta automáticamente las claves de unión, se requiere que el nombre de unión sea el mismo en cada tabla. Funciona aunque no estén definidas las claves primarias o ajenas.

JOIN USING: permite establecer relaciones indicando que campo o campos comunes se quieren utilizar, si no queremos que se relacionen todos.

JOIN ON: une tablas en las que los nombres de columna no coinciden en ambas tablas o se necesita establecer asociaciones más complicadas.

OUTER JOIN. Elimina el uso del signo (+) para composiciones externas.

LEFT OUTER JOIN. Composición externa izquierda, todas las filas de la tabla de la izquierda se devuelven, aunque no haya ninguna columna correspondiente en la tabla combinada.

RIGHT OUTER JOIN. Composición externa derecha.

FULL OUTER JOIN. Se devuelven todas las filas de los campos no relacionados de ambas tablas.

Por ejemplo, obtenemos el historial laboral de los empleados: nombre y apellidos, la fecha en la que entraron a trabajar y la fecha de fin de trabajo si no continúan en la empresa. Es una composición interna con 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);
```

En este otro ejemplo, obtenemos los distintos departamentos y sus jefes con sus datos personales, deben aparecer los departamentos aunque no tengan asignados jefes. Es una composición externa con 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);
```

9. Otras consultas multatablas: Unión, intersección y la diferencia de consultas

UNION: Combina filas de un primer SELECT con otro SELECT, desapareciendo las filas duplicadas. Ejemplo:
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';
```

INTERSECT: examina las filas de dos SELECT y devuelve aquellas que aparezcan en ambos conjuntos. Las filas duplicadas se eliminan. Ejemplo: Una academia de idiomas da clases de inglés, francés y portugués; almacena los datos de los alumnos en tres tablas distintas una llamada "ingles", en una tabla denominada "frances" y los que aprenden portugués en la tabla "portugues". La academia necesita el nombre y domicilio de todos los alumnos que cursan los tres idiomas para enviarles información sobre los exámenes.

```
SELECT nombre, domicilio FROM ingles INTERSECT  
SELECT nombre, domicilio FROM frances INTERSECT  
SELECT nombre, domicilio FROM portugues;
```

MINUS: devuelve aquellas filas que están en el primer SELECT pero no en el segundo. Las filas del primer SELECT se reducen antes de comenzar la comparación. Ejemplo: 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;
```

Se utilizan en los dos SELECT el mismo número y tipo de columnas y en el mismo orden.

10. Subconsultas

A veces se deben utilizar en consultas resultados de otras a las que llamamos subconsulta o consulta subordinada:

```
SELECT listaExpr  
FROM tabla  
WHERE expresión_o_columna OPERADOR  
(SELECT expresion_o_columna  
FROM tabla);
```

Puede ir dentro de WHERE, HAVING O FROM.

- >, <, >=, <=, !=, =, IN. Las subconsultas que utilizan estos operadores devuelven un único valor.

Ejemplo: Obtenemos el nombre de los empleados y el sueldo de aquellos que cobran menos que Ana, si hay más de un empleado llamado Ana, se devuelve error.

```
SELECT Nombre, salario
FROM EMPLEADOS
WHERE salario <
(SELECT salario FROM EMPLEADOS
WHERE Nombre= 'Ana');
```

Si necesitamos que la subconsulta devuelva más de un valor y queremos comparar el campo con todos esos valores se utilizan palabras reservadas:

- **ANY.** Compara con cualquier fila de la consulta. La instrucción es válida cuando un registro de la subconsulta permite que la comparación sea cierta.
- **ALL.** Compara con todas las filas de la consulta y será válida si es cierta la comparación con todas las filas.
- **IN.** No utiliza comparador, comprueba que el valor se encuentra en la subconsulta.
- **NOT IN.** Comprueba que un valor no se encuentre en la subconsulta.

Ejemplo. En la siguiente consulta obtenemos al empleado que menos cobra:

```
SELECT nombre, salario
FROM EMPLEADOS
WHERE salario <= ALL (SELECT salario FROM EMPLEADOS);
```

y en esa misma consulta se podría realizar utilizando función de agregado o colectiva MIN de la misma forma.

```
SELECT nombre, salario
FROM EMPLEADOS
WHERE salario = (SELECT MIN(salario) FROM EMPLEADOS);
```



5. Tratamiento de datos

Class	Bases de datos
Column	(X) Xerach Casanova
Last Edited time	@Feb 4, 2021 10:46 PM

- 1. Introducción
 - 2. Edición de la información mediante herramientas gráficas
 - 2.1. Inserción de registros
 - 2.2. Modificación de registros
 - 2.3. Borrado de registros
- 3. Edición de la información mediante sentencias SQL
 - 3.1. Inserción de registros
 - 3.2. Modificación de registros
 - 3.2. Borrado de registros
- 4. Integridad referencial
 - 4.1. Integridad en actualización y supresión de registros.
 - 4.2. Supresión en cascada
- 5. Subconsultas y composiciones en órdenes de edición
 - 5.1. Inserción de registros a partir de una consulta.
 - 5.2. Modificación de registros a partir de una consulta
 - 5.3. Supresión de registros a partir de una consulta
- 6. Transacciones
 - 6.1. Hacer cambios permanentes
 - 6.2. Deshacer cambios
 - 6.3. Deshacer cambios parcialmente
- 7. Políticas de boqueo
 - 7.1. Políticas de bloqueo
 - 7.2. Bloqueos compartidos y exclusivos
 - 7.3. Bloqueos automáticos
 - 7.4. Bloqueos manuales
 - 7.5. Interbloqueos

Mapa conceptual

1. Introducción

Las operaciones de tratamiento de datos son las acciones que permiten añadir, modificar o suprimir información de la base de datos.

Este tipo de operaciones debe seguir una serie de requisitos en las relaciones existentes entre las tablas que la componen. Todas las operaciones que se realicen deben asegurar que se cumplen las relaciones existentes entre ellas en todo momento.

Además, si una aplicación falla en un momento dado no debe impedir que la información almacenada sea incorrecta, así como la posibilidad de cancelar una determinada operación no debe suponer un problema para la fiabilidad de los datos almacenados.

2. Edición de la información mediante herramientas gráficas

Para realizar el tratamiento de datos por línea de comandos se requiere la utilización de lenguaje SQL, sin embargo, con las herramientas gráficas para manipulación de datos, se permite la introducción, edición y borrado de datos desde un entorno gráfico.

En Oracle podemos trabajar con SQLDeveloper y en otros, como en SGDB Mysql se encuentran herramientas gráficas similares, como phpmyadmin.

2.1. Inserción de registros

La inserción de registros o filas permite introducir nuevos datos en las tablas que componen la base de datos, en SQLDeveloper se realizan haciendo click en cualquier tabla, pulsando en la pestaña datos y haciendo click en el icono de añadir.

DPTO_COD	NOMBRE_DPTO	JEFE	PRESUPUESTO	PRES_ACTUAL
1	1 INFORMATICA	33333	80000	50000
2	99 RECURSOS HUMANOS	(null)	20000	140000
+3	NUEVO DEPARTAMENTO	(null)	(null)	(null)

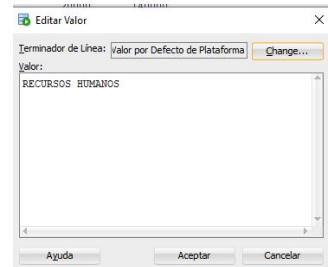
Cuando se finaliza la inserción de todas las filas se confirma la operación con el botón correspondiente y si todo va bien, se visualizará un mensaje indicando que la inserción se ha realizado de forma correcta.



Si se produce un error, se debe comprobar que mensaje muestra para solucionarlo, por ejemplo por intentar insertar datos alfanuméricos en una columna numérica.

2.2. Modificación de registros

Permite la modificación de datos ya existentes en las tablas. Para ello nos situamos directamente en la columna de la fila que se quiere modificar, hacemos click con el ratón y escribimos el nuevo contenido. También se puede realizar pulsando el icono del lápiz que aparece al lado de la columna e introduciendo el valor en la ventana emergente.



DPTO_COD	NOMBRE_DPTO	JEFE	PRESUPUESTO	PRES_ACTUAL
1	1 INFORMATICA	33333	80000	50000
2	99 RECURSOS HUMANOS	(null)	20000	140000
*3	98 MARKETING	<input type="text"/>	12345	222

Para realizar las modificaciones se confirma la operación en el botón correspondiente y también debemos analizar los errores en caso de que se hayan producido.

2.3. Borrado de registros

Permite la eliminación de datos existentes en las tablas. Para eliminar un registro o fila de una tabla, en la misma pestaña de datos, nos situamos en la fila que se quiera eliminar, en cualquier columna y pulsamos el icono en forma de de

aspas. Al hacerlo se colocará el signo - delante del número de cada fila.

DPTO_COD	NOMBRE_DPTO	JEFE	PRESUPUESTO	PRES_ACTUAL
1	INFORMATICA	33333	80000	50000
-2	RECURSOS HUMANOS	(null)	20000	140000
-3	MARKETING	12345	222	22

Se debe confirmar el borrado con el botón correspondiente, pero también podemos deshacer los borrados antes de confirmar en el icono que se muestra.

La eliminación de un registro no se puede realizar si un registro de otra tabla hace referencia a él y se mostrará un mensaje de error en pantalla. Si esto ocurre, para eliminar el registro se debe eliminar el registro que hace referencia a él o modificarlo para que haga referencia a otro registro.



3. Edición de la información mediante sentencias SQL

El lenguaje SQL dispone de una serie de sentencias para la edición de datos. Este conjunto de sentencias recibe el nombre de DML (Data Manipulation Language).

Estas se pueden ejecutar en SQLPlus o escribiéndolas en cualquier hoja de trabajo de SQLDeveloper.

EMPLEADO_DNI	UNIVERSIDAD	AGNO	GRADO	ESPECIALIDAD
1	12345	1	1992 MED	ADMINISTRATIVO
2	22222	1	1998 SUP	ING INFORMÁTICA
3	33333	2	1997 SUP	LIC INFORMÁTICA

Para trabajar con SQLDeveloper pulsamos en el botón SQL, del menú superior. Se solicitará la conexión con la que vamos a trabajar y seguidamente se abrirá la hoja de trabajo. para ejecutar las sentencias se utiliza el botón play.

También se puede acceder a todas las sentencias SQL que se han escrito aunque no se hayan guardado con el botón que se indica.



3.1. Inserción de registros

La sentencia INSERT permite insertar nuevas filas o registros. El formato más sencillo es:

```
INSERT INTO nombre_tabla (lista_campos) VALUES (lista_valores);
```

nombre_tabla es el nombre de la tabla donde se añaden los nuevos registros, lista_campos son los campos de dicha tabla en los que se van a insertar los valores de lista_valores. Es posible omitir la lista de campos si se van a insertar todos los valores de cada campo y en el orden en el que se encuentran en la tabla.

La lista de campos debe tener un valor válido en la posición correspondiente de lista_valores. Si no se recuerda bien se puede utilizar la sentencia DESCRIBE seguido del nombre de la tabla a consultar.

Con el siguiente script podemos hacer pruebas en las tablas de JuegosOnline, ejecutando el script en sqlplus.

```
-- Desde CONN SYS AS SYSDBA/ORACLE;
create user c##JUEGOS identified by juegos default tablespace users;
grant connect, resource,DBA to c##JUEGOS;
conn c##JUEGOS/juegos
CREATE TABLE USUARIOS (
    login      VARCHAR2(15) PRIMARY KEY NOT NULL,
    password   VARCHAR2(9) NOT NULL,
    nombre     VARCHAR2(25) NOT NULL,
    apellidos  VARCHAR2(30) NOT NULL,
    direccion  VARCHAR2(30) NOT NULL,
    cp         VARCHAR2(5) NOT NULL,
    localidad  VARCHAR2(25) NOT NULL,
    provincia  VARCHAR2(25) NOT NULL,
    pais       VARCHAR2(15) NOT NULL,
    f_nac      DATE,
    f_ing      DATE DEFAULT (sysdate),
    correo    VARCHAR2(25) NOT NULL,
    credito   NUMBER,
    sexo       VARCHAR2(1));
--TABLA JUEGOS
CREATE TABLE JUEGOS(
    codigo      VARCHAR2(15) PRIMARY KEY NOT NULL,
    nombre     VARCHAR2(15) NOT NULL,
    descripcion VARCHAR2(200) NOT NULL);
--TABLA PARTIDAS
CREATE TABLE PARTIDAS(
    codigo      VARCHAR2(15) PRIMARY KEY NOT NULL,
    nombre     VARCHAR2(25) NOT NULL,
    estado     VARCHAR2(1) NOT NULL,
    cod_juego  VARCHAR2(15) NOT NULL
        CONSTRAINT CA_cod_juego REFERENCES JUEGOS(codigo),
    fecha_inicio DATE,
    hora_inicio  TIMESTAMP,
    cod_creador  VARCHAR2(15)
        CONSTRAINT CA_cod_creador REFERENCES USUARIOS(login));
--TABLA UNEN
CREATE TABLE UNEN(
    codigo_partida VARCHAR2(15) NOT NULL
        CONSTRAINT CA_codigo_partida REFERENCES PARTIDAS(codigo),
    codigo_usuario VARCHAR2(15) NOT NULL
        CONSTRAINT CA_codigo_usuario REFERENCES USUARIOS(login),
    CONSTRAINT PK_UNEN primary key (codigo_partida, codigo_usuario));
ALTER SESSION SET NLS_DATE_FORMAT='MM/DD/YYYY';
INSERT INTO USUARIOS VALUES('anamat56','JD9U67','ANA M.','MATA VARGAS','GARCILASO DE LA VEGA','8924','SANTA COLOMA DE GRAMANET','BARCE
INSERT INTO USUARIOS VALUES('alecam89','5;5@PK','ALEJANDRO EMILIO','CAMINO LAZARO','PEDRO AGUADO BLEYE','34004','PALENCIA','PALENCIA',
INSERT INTO USUARIOS VALUES('verbad64','MP49HF','VERONICA','BADIOLA PICAZO','BARRANCO GUINIGUADA','35015','PALMAS GRAN CANARIA, LAS','P
INSERT INTO USUARIOS VALUES('conmar76','01<N9U','CONSUENO','MARTINEZ RODRIGUEZ','ROSA','4002','ALMERIA','ALMERIA','ESPAÑA','08/09/1978
INSERT INTO USUARIOS VALUES('encpay57','FYC3LS','ENCARNACIÓN','PAYO MORALES','MULLER,AVINGUDA','43007','TARRAGONA','TARRAGONA','ESPAÑA
INSERT INTO USUARIOS VALUES('mandia79','00JRIH','MANUELA','DIAZ COLAS','214 (GENOVA)','7015','PALMA DE MALLORCA','BALEARES','ESPAÑA','
INSERT INTO USUARIOS VALUES('alibar52','IERS8','ALICIA MARIA','BARRANCO CALLIZO','HECTOR VILLALOBOS','29014','MÁLAGA','MÁLAGA','ESPAÑA
INSERT INTO USUARIOS VALUES('adofid63',';82-MH','ADOLFO','FIDALGO DIEZ','FORCALL','12006','CASTELLÓN DE LA PLANA','CASTELLÓN','ESPAÑA'
INSERT INTO USUARIOS VALUES('jesdie98','X565ZS','JESUS','DIEZ GIL','TABAIBAL','35213','TELDE','PALMAS (LAS)','ESPAÑA','10/23/1981','09
INSERT INTO USUARIOS VALUES('pedsan70','T?5=J@','PEDRO','SANCHEZ GUIL','PINTOR ZULOAGA','3013','ALACANT ALICANTE','ALICANTE','ESPAÑA',
INSERT INTO USUARIOS VALUES('diahue96','LSQZMC','DIANA','HUERTA VALIOS','JOAQUIN SALAS','39011','SANTANDER','CANTABRIA','ESPAÑA','04/2
INSERT INTO USUARIOS VALUES('robrod74','<LQMLP','ROBERTO','RODRIGUEZ PARMO','CASTILLO HIDALGO','51002','CEUTA','CEUTA','ESPAÑA','06/28
INSERT INTO USUARIOS VALUES('milgar78','SF=UZB','MILAGROSA','GARCIA ELVIRA','PEDRALBA','28037','MADRID','MADRID','ESPAÑA','04/12/1983'
INSERT INTO USUARIOS VALUES('frabar93','19JZ7@','FRANCISCA','BARRANCO RODRIGUEZ','BALSAS, LAS','26006','LOGROÑO','RIOJA (LA)','ESPAÑA'
INSERT INTO USUARIOS VALUES('migarc93','AAFLTW','MIGUEL ANGEL','ARCOS ALONSO','ISAAC ALBENIZ','4008','ALMERIA','ALMERIA','ESPAÑA','03/
--TABLA JUEGOS
INSERT INTO JUEGOS VALUES('1','Parchís','El parchís es un juego de mesa derivado del pachisi y similar al ludo y al parcheesi');
INSERT INTO JUEGOS VALUES('2','Oca','El juego de la oca es un juego de mesa para dos o más jugadores');
INSERT INTO JUEGOS VALUES('3','Ajedrez','El ajedrez es un juego entre dos personas, cada una de las cuales dispone de 16 piezas móviles
INSERT INTO JUEGOS VALUES('4','Damas','Las damas es un juego de mesa para dos contrincantes');
INSERT INTO JUEGOS VALUES('5','Poker','El póker es un juego de cartas de los llamados de "apuestas"');
INSERT INTO JUEGOS VALUES('6','Chinchón','El chinchón es un juego de naipes de 2 a 8 jugadores');
```

```

INSERT INTO JUEGOS VALUES('7','Mus','El mus es un juego de naipes, originario de Navarra, que en la actualidad se encuentra muy extendido');
INSERT INTO JUEGOS VALUES('8','Canasta','La canasta o rummy-canasta es un juego de naipes, variante del rummy');
INSERT INTO JUEGOS VALUES('9','Dominó','El dominó es un juego de mesa en el que se emplean unas fichas rectangulares');
INSERT INTO JUEGOS VALUES('10','Pocha','La pocha es un juego de cartas que se juega con la baraja española');
INSERT INTO JUEGOS VALUES('11','Backgammon','Cada jugador tiene quince fichas que va moviendo entre veinticuatro triángulos (puntos) situados en una tabla de doble fila');
INSERT INTO JUEGOS VALUES('12','Billar','El billar es un deporte de precisión que se practica impulsando con un taco un número variable de bolas');

-- PARTIDAS

INSERT INTO PARTIDAS VALUES('1','Billar_migarc93_18/7','1','12','07/18/2011',TO_TIMESTAMP ('00:47:40','HH24:MI:SS'),'migarc93');
INSERT INTO PARTIDAS VALUES('2','Chinchón_mandia79_2/10','1','6','10/02/2011',TO_TIMESTAMP ('01:47:40','HH24:MI:SS'),'mandia79');
INSERT INTO PARTIDAS VALUES('3','Canasta_alibar52_26/2','0','8','02/26/2011',TO_TIMESTAMP ('08:57:33','HH24:MI:SS'),'alibar52');
INSERT INTO PARTIDAS VALUES('4','Damas_verbad64_16/3','1','4','03/16/2011',TO_TIMESTAMP ('00:53:00','HH24:MI:SS'),'verbad64');
INSERT INTO PARTIDAS VALUES('5','Chinchón_alibar52_9/9','1','6','09/09/2011',TO_TIMESTAMP ('09:10:22','HH24:MI:SS'),'alibar52');
INSERT INTO PARTIDAS VALUES('6','Oca_pedisan70_21/12','0','2','12/21/2011',TO_TIMESTAMP ('18:53:17','HH24:MI:SS'),'pedsan70');
INSERT INTO PARTIDAS VALUES('7','Canasta_encipay57_18/2','0','8','02/18/2011',TO_TIMESTAMP ('09:41:02','HH24:MI:SS'),'encipay57');
INSERT INTO PARTIDAS VALUES('8','Pocha_adofid63_26/10','1','10','10/26/2011',TO_TIMESTAMP ('02:23:43','HH24:MI:SS'),'adofid63');
INSERT INTO PARTIDAS VALUES('9','Damas_diahue96_25/6','1','4','06/25/2011',TO_TIMESTAMP ('18:11:14','HH24:MI:SS'),'diahue96');
INSERT INTO PARTIDAS VALUES('10','Parchis_encipay57_31/7','1','1','07/31/2011',TO_TIMESTAMP ('21:21:36','HH24:MI:SS'),'encipay57');

-- TABLA UNEN

INSERT INTO UNEN VALUES('4','anamat56');
INSERT INTO UNEN VALUES('3','alecam89');
INSERT INTO UNEN VALUES('6','alecam89');
INSERT INTO UNEN VALUES('2','comnar76');
INSERT INTO UNEN VALUES('2','encipay57');
INSERT INTO UNEN VALUES('2','mandia79');
INSERT INTO UNEN VALUES('4','alibar52');
INSERT INTO UNEN VALUES('3','adofid63');
INSERT INTO UNEN VALUES('5','jesdie98');
INSERT INTO UNEN VALUES('8','pedsan70');
INSERT INTO UNEN VALUES('6','diahue96');
INSERT INTO UNEN VALUES('4','robrod74');
INSERT INTO UNEN VALUES('5','milgar78');
INSERT INTO UNEN VALUES('4','frabar93');
INSERT INTO UNEN VALUES('5','encipay57');

```

Antes de ejecutar las siguientes sentencias debemos ejecutar la siguiente, para que se tome la fecha en el formato que vamos a insertar registros.

```
ALTER SESSION SET NLS_DATE_FORMAT='DD/MM/YYYY';
```

En este ejemplo hacemos una inserción de registros indicando toda la lista de campos:

```
INSERT INTO USUARIOS (Login, Password, Nombre, Apellidos, Direccion, CP, Localidad, Provincia, Pais, F_Nac, F_Ing, 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');
```

Y en este otro, se inserta un registro pero sin disponer de todos los datos.

```
INSERT INTO USUARIOS (Login, Password, Nombre, Apellidos, direccion, cp, localidad, provincia, pais, Correo) VALUES ('natsan63', 'VBROMI', 'NATALIA', 'SANCHEZ GARCIA', 'C/Blanca', '28003', 'Madrid', 'Madrid', 'Spain', 'natsan63@hotmail.com');
```

En aquellos campos que no se especifiquen valores, se obtiene NULL. Si la lista de campos indicados no se corresponde con la lista de valores, o no se proporcionan valores para campos que no admiten valor NULL se obtiene un error en ejecución

```
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

3.2. Modificación de registros

La sentencia UPDATE permite modificar una serie de valores de determinados registros de las tablas:

```
UPDATE nombre_tabla SET nombre_campo = valor [, nombre_campo = valor]...
[ WHERE condición ];
```

nombre_tabla es el nombre de la tabla en la que modificaremos datos. Se indican en cada campo el nuevo valor utilizando el signo =, cada emparejamiento campo=valor, debe ir separado por comas.

La cláusula Where seguida de la condición es opcional, si se indica, la modificación afecta solo a los registros que la cumplen y si no, afectará a todos los registros.

Añadimos 200 al crédito de todos los usuarios:

```
UPDATE USUARIOS SET Credito = 200;
```

Ponemos a cero el crédito y ponemos en nulo el valor del campo f_nac de todos los usuarios:

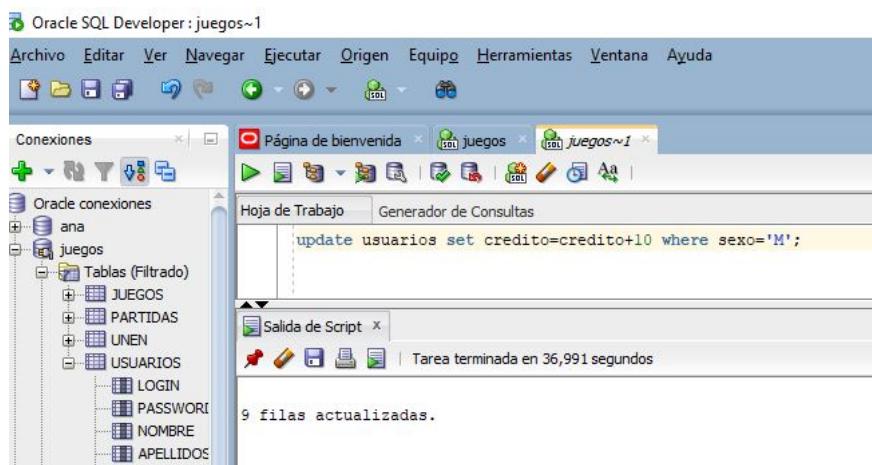
```
UPDATE USUARIOS SET Credito = 0, f_nac = NULL;
```

Damos 300 de crédito a los registros que correspondan a mujeres:

```
UPDATE USUARIOS SET Credito = 300 WHERE Sexo = 'M';
```

Cuando termina la ejecución de una sentencia UPDATE se muestra la cantidad de registros actualizados o se obtiene un error si algo ha ido mal.

También podemos escribir la sentencia en la hoja de trabajo de SQLDeveloper. Por ejemplo incrementar en 10 el crédito de las mujeres



3.2. Borrado de registros

La sentencia DELETE permite borrar registros de una tabla:

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

nombre_tabla hace referencia a la tabla donde se va a realizar la operación, la cláusula where es opcional y si no se usa se borrará el contenido de toda la tabla aunque la estructura seguirá intacta.

Elimina todos los usuarios:

```
DELETE FROM USUARIOS;
```

Borramos los usuarios que tengan crédito cero:

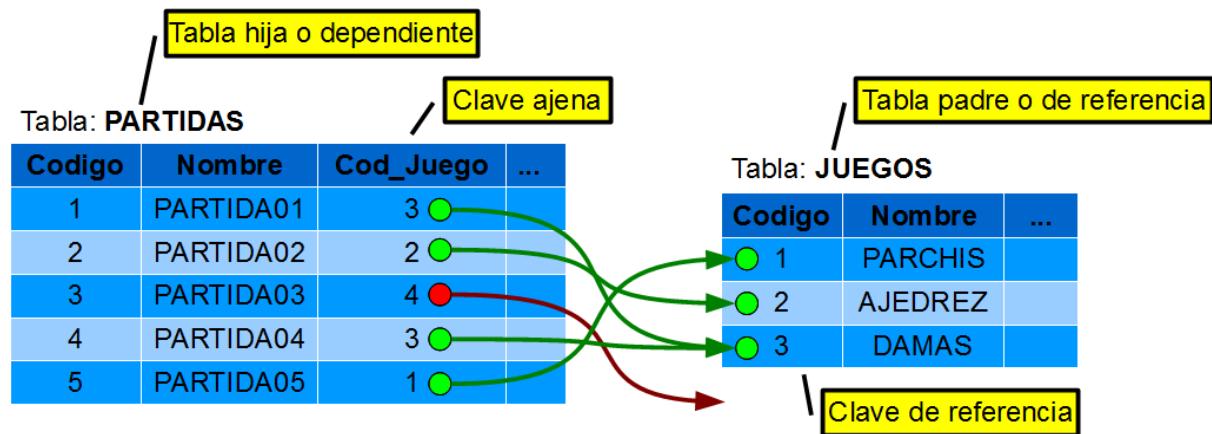
```
DELETE FROM USUARIOS WHERE Credito = 0;
```

Se obtendrá un mensaje de error si se produce algún problema o el número de filas eliminadas.

4. Integridad referencial

Dos tablas pueden estar relacionadas entre ellas por uno o más campos que tengan en común. Cada valor del campo que forma parte de la integridad referencial definida, debe corresponderse en otra tabla con otro registro que contenga el mismo valor en el campo referenciado.

En el ejemplo de juegos online, podemos tener una tabla PARTIDAS en la que existe una referencia al tipo de juego que le corresponde, mediante su código. No puede existir ninguna partida que no tenga un código que corresponda con alguno de la tabla JUEGOS. Por tanto, en la siguiente imagen, el código de juego 4 no cumple la integridad referencial, pues no existe un juego con código 4.



Cuando se habla de integridad referencial se utilizan los siguientes términos:

- **Clave ajena (o foránea)**, es el campo o conjunto de campos incluidos en la definición de la restricción que deben hacer referencia a una clave de referencia. (Cod_Juego en la tabla PARTIDAS)
- **Clave de referencia**: clave única o primaria en la tabla a la que se hace referencia desde la clave ajena (Código en la tabla JUEGOS)
- **Tabla hija o dependiente**. Tabla que incluye la clave ajena y por tanto depende de los valores existentes en la clave de referencia (tabla PARTIDAS)
- **Tabla padre o de referencia**: Corresponde a la tabla que es referenciada por la clave ajena en la tabla hija. Esta determina las inserciones o actualizaciones que son permitidas en la tabla hija, en función de dicha clave. (tabla JUEGOS)

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 valores.

Si modificamos 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. Del mismo modo, no podemos modificar el valor de la clave principal en un registro de la tabla padre, si una clave ajena hace referencia a ese registro.

En los borrados de registros no se pueden suprimir registros referenciados con una clave ajena desde otra tabla.

En el registro de la partida con nombre PARTIDA01, no podemos modificar el valor de Cod_Juego a 4, debido a que no existe en la tabla JUEGOS un registro con esa clave primaria.

El código de juego DAMAS no se puede cambiar, ya que hay registros en la tabla PARTIDAS que hacen referencia a dicho juego a través del campo Cod_Juego.

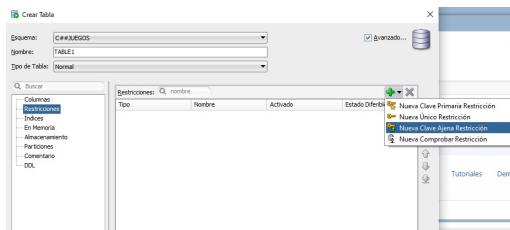


Para borrar o modificar registros en una tabla de referencia, se puede configurar la clave ajena de distintas maneras para conservar la integridad referencial:

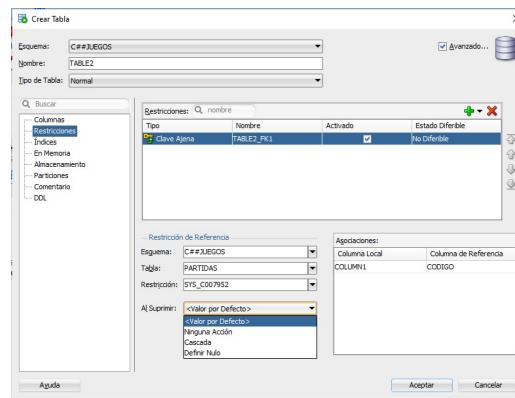
- No permitir supresión ni modificación.** Es la opción por defecto, en caso de borrar o modificar en la tabla de referencia un registros que está siendo referenciado desde otra tabla, se produce error.
- Supresión a modificación en Cascada (ON DELETE/UPDATE CASCADE).** Al suprimir o modificar registros en la tabla de referencia, los registros de la tabla hija también son borrados o modificados.
- Asignación de Nulo (ON DELETE/UPDATE SET NULL).** Los valores de la clave ajena que hacían referencia a los registros borrados o modificados se cambian a NULL.
- Valor por defecto (ON DELETE/UPDATE DEFAULT).** Los valores de la clave ajena que hacían referencia a los registros borrados o modificados se cambian a su valor por defecto.

4.2. Supresión en cascada

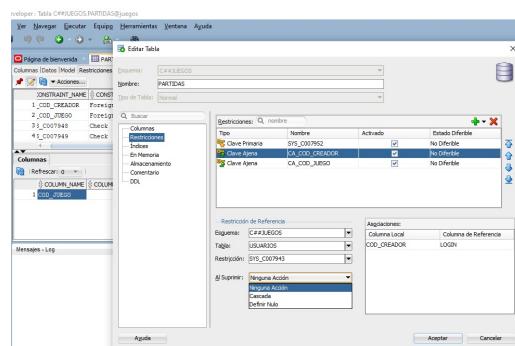
Las opciones de supresión en Cascada o Definir Nulo en suprimir se pueden establecer desde que creamos las tablas, desde SQLDeveloper marcando en la casilla avanzado, nos situamos en la columna que es clave ajena y seleccionamos en el árbol de restricciones, en la lista desplegable, del botón Añadir, seleccionamos nueva clave ajena Restricción como se muestra en imagen.



Seguidamente se selecciona "en cascada" en el desplegable "al suprimir"



Si la tabla ya estaba creada y se desea establecer una restricción de clave ajena con una opción concreta, se puede realizar desde la opción "editar tabla" siguiendo los mismos pasos.



Para realizar este tipo de operaciones, se dispone de las siguientes opciones durante la declaración de la clave ajena en la tabla (también podemos usar ON DELETE SET NULL):

```
CONSTRAINT JUEGOS_CON FOREIGN KEY (Cod_Juego) REFERENCES JUEGO (Codigo) ON DELETE CASCADE;
```

Si queremos añadirlo después de haber sido creado, podemos usar la estructura ALTER TABLE.

5. Subconsultas y composiciones en órdenes de edición

Podemos utilizar sentencias de una forma más avanzada insertando consultas dentro de esas mismas operaciones de tratamiento de datos.

Una tabla se puede ver afectada por los resultados de las operaciones en otras tablas. Con una misma instrucción, podemos añadir más de un registro a una tabla, o actualizar o eliminar varios registros basados en otras consultas.

Los valores que se añadan o modifiquen pueden ser obtenidos como resultado de una consulta.

También podemos usar como condiciones en las sentencias, otras consultas.

5.1. Inserción de registros a partir de una consulta.

El siguiente INSERT

```
INSERT INTO USUARIOS (Login, Password, Nombre, Apellidos, direccion, cp, localidad, provincia, pais, Correo) VALUES ('natsan63', 'VBROMI', 'NATALIA', 'SANCHEZ GARCIA', 'C/Blanca', '28003', 'Madrid', 'Madrid', 'Spain', 'natsan63@hotmail.com');
```

también se puede realizar así:

```
INSERT INTO (SELECT Login, Password, Nombre, Apellidos, direccion, cp, localidad, provincia, pais, Correo FROM USUARIOS) VALUES ('natsan63', 'VBROMI', 'NATALIA', 'SANCHEZ GARCIA', 'C/Blanca', '28003', 'Madrid', 'Madrid', 'Spain', 'natsan63@hotmail.com');
```

Sustituyendo el nombre de la tabla, junto con sus campos, por una consulta equivalente.

También es posible insertar una tabla de valores que se obtienen directamente del resultado de una consulta.

Supongamos que tenemos una tabla USUARIOS_SIN_CREDITO con la misma estructura que la tabla USUARIOS ya creada, 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;
```

En este caso, no se utiliza la palabra values, ya que no se está especificando una lista de valores.

También podemos crear una tabla e insertar datos a partir de una consulta:

```
CREATE TABLE USUARIOS_CON_CREDITO AS SELECT * FROM USUARIOS WHERE CREDITO >0 ;
```

Si queremos crear una tabla con la misma estructura pero sin contenido, podemos crear una condición que no se cumpla nunca para que no se copie ningún registro.

```
CREATE TABLE USUARIAS AS SELECT * FROM USUARIOS WHERE 1 <0;
```

Después podremos insertar en esta nueva tabla:

```
INSERT INTO USUARIAS  
SELECT * FROM USUARIOS WHERE UPPER(SEXO)='M';
```

5.2. Modificación de registros a partir de una consulta

También podemos utilizar consultas para realizar modificaciones complejas.

Las consultas pueden formar parte de cualquiera de los elementos de la sentencia UPDATE.

En el siguiente ejemplo, la sentencia modifica el crédito de los usuarios que tienen una partida creada y cuyo estado es 1 (activada). El valor que se le 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_Creador FROM PARTIDAS WHERE Estado=1);
```

5.3. Supresión de registros a partir de una consulta

Se puede realizar borrado de registros utilizando consultas como parte de las tablas donde se hará la eliminación, o como parte de la condición que delimita la operación.

```
DELETE FROM (SELECT LOGIN FROM USUARIOS, UNEN WHERE CODIGO_USUARIO=LOGIN AND PROVINCIA='PALENCIA');
```

Se eliminarán determinados registros de la tabla USUARIOS y UNEN, en concreto aquellos que en la tabla UNEN registran asociados a algún usuario de Palencia.

En este caso no ha hecho falta utilizar ninguna condición WHERE en la sentencia. Otra manera, pero utilizando la cláusula WHERE:

```
DELETE FROM (SELECT LOGIN, PROVINCIA FROM USUARIOS, UNEN WHERE CODIGO_USUARIO=LOGIN) WHERE PROVINCIA='PALENCIA';
```

6. Transacciones

Una transacción es una unidad atómica de trabajo que puede contener una o más sentencias SQL. Las transacciones agrupan sentencias SQL de tal manera que a todas ellas se les aplica una operación COMMIT (confirmadas, aplicadas o guardadas en la BBDD), o bien a todas ellas se les aplica la acción ROLLBACK (deshacer operaciones que deberían hacer sobre la BBDD).

Un ejemplo sería el proceso de transferencia entre cuentas bancarias, donde Juan hace 100 euros de transferencia a María:

- Actualizar la cuenta de Juan restando 100.
- Registrar el movimiento de decremento de 100 en los movimientos de cuenta de Juan.
- Actualizar la cuenta de María incrementando 100.
- Registrar el movimiento de incremento en los movimientos de María.

Si hubiese un corte o caída en el sistema en el punto 2, los 100 euros no los tendría ni Juan, ni María. Para evitar estas situaciones se utilizan las transacciones, consideradas como una única operación. Hasta que las 4 operaciones no estén realizadas no se confirma (COMMIT) la operación y, si hay problemas en el proceso, se deshacen (ROLLBACK) las operaciones que dejaron a medias el proceso.

Mientras no se haga un COMMIT sobre una transacción, los resultados se pueden deshacer. Así, una sentencia del lenguaje de manipulación de datos (DML) no es permanente hasta que se realiza un COMMIT sobre la transacción en la que esté incluida o hasta que se ejecuta una operación DDL.

Las transacciones de Oracle cumplen con las siguientes propiedades:

- **Atomicidad.** O se realizan todas las tareas o no se realiza ninguna. No hay una transacción parcial.
- **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 operaciones hasta que finaliza.
- **Durabilidad:** los cambios efectuados por las transacciones que han volcado sus modificaciones son permanentes.

Las sentencias de control de transacciones gestionan los cambios que realizan las sentencias DML y las agrupa en transacciones. Estas permiten:

- **COMMIT** (hacer permanentes cambios producidos por una transacción).
- **ROLLBACK** (deshacer cambios de una transacción desde que fue iniciada o desde un punto de restauración (**ROLLBACK TO SAVEPOINT**). ROLLBACK termina la transacción, pero ROLLBACK TO SAVEPOINT no.

- Establecer un punto intermedio (**SAVEPOINT**) a partir del cual se puede deshacer la transacción.
- Indicar propiedades de una transacción (**SET TRANSACTION**). Por ejemplo, elegir **READ ONLY** no permite modificaciones.
- 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**). Se puede elegir **IMMEDIATE** o **DEFERRED**.

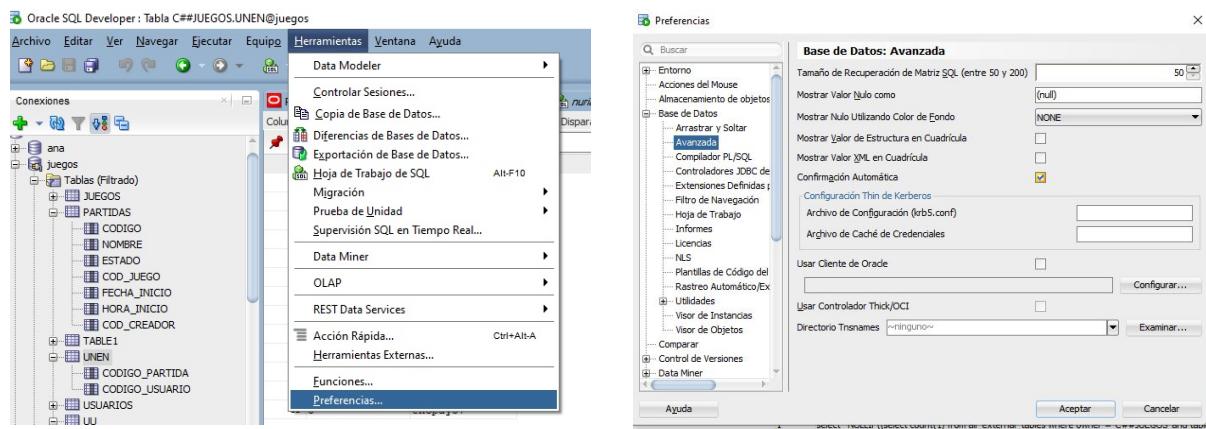
6.1. Hacer cambios permanentes

Una transacción comienza con la primera sentencia SQL ejecutable, para que los cambios producidos se hagan permanente se dispone de:

- Utilizar la sentencia **COMMIT**
- Ejecutar una sentencia **DDL** (como CREATE, DROP, RENAME O ALTER)
- Si el usuario cierra adecuadamente las aplicaciones de gestión de bases de datos de Oracle.

Para permitir **ROLLBACK** es necesario desactivar la variable de confirmación **AUTOCOMMIT**.

Para modificar el valor de la variable se puede hacer desde Herramientas/Preferencias, abriendo la opción Avanzada de Base de Datos y marcando o desmarcando la casilla de confirmación automática.



También se puede hacer desde SQLPlus con el comando **show autocommit** para mostrar el valor de la sesión actual y cambiar el valor con el comando **SET**.

6.2. Deshacer cambios

La sentencia **ROLLBACK** permite deshacer cambios efectuados y la da por finalizada. Se recomienda explícitamente finalizar las transacciones en las aplicaciones usando las sentencias **COMMIT** o **ROLLBACK**.

Si la transacción termina de forma anormal, los cambios que hasta el momento se hubiesen realizado serán deshechos automáticamente.

6.3. Deshacer cambios parcialmente

Un punto de restauración (**SAVEPOINT**) es un marcador intermedio declarado por el usuario en el contexto de una transacción. Estos dividen la transacción grande en pequeñas partes.

Si se usan puntos de restauración en una transacción larga, se tiene la opción de deshacer los cambios efectuados por la transacción a justo después del punto de restauración establecido. Si se produce un error no es necesario rehacer todas las sentencias, sino aquellas posteriores al punto de restauración.

```
SAVEPOINT nombre_punto_restauración;
```

```
ROLLBACK TO SAVEPOINT nombre_punto_restauración;
```

7. Políticas de boqueo

Un dato no puede ser modificado sin tener en cuenta que otros usuarios están modificando el dato al mismo tiempo. Las transacciones ejecutadas simultáneamente deben generar resultados consistentes. 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 visión consistente de los datos, incluyendo cambios visibles por transacciones del mismo usuario y las finalizadas de otros usuarios.

En una base de datos monousuario no es necesario bloquear, ya que solo modifica datos un usuario, pero en bases de datos multiusuario se debe proveer de un mecanismo para prevenir la modificación concurrente del mismo dato.

Los bloqueos permiten:

- **Consistencia.** Los datos consultados o modificados no pueden ser cambiados por otro hasta que el usuario haya finalizado la operación completa.
- **Integridad:** los datos y su estructura deben reflejar todos los cambios efectuados sobre ellos en el orden correcto.

7.1. Políticas de bloqueo

Los bloqueos afectan a la interacción de lectores y escritores. Un lector es quien consulta sobre un recurso y un escritor es quien realiza una modificación. Oracle resume las reglas del comportamiento sobre lectores y escritores de la siguiente manera.

- Un registro se bloquea solo cuando es modificado por un escritor, cuando una sentencia actualiza un registro, la transacción obtiene un bloqueo solo para ese registro.
- Un escritor de un registro bloquea a otro escritor concurrente del mismo registro.
- Un lector no bloquea a un escritor. La única excepción es SELECT... FOR UPDATE (una sentencia especial que bloquea el registro que está siendo consultado).
- Un escritor no bloquea a un lector. Cuando un registro está siendo modificado, la base de datos proporciona al lector una vista del registro si los cambios que se están realizando.

Hay dos mecanismos para el bloqueo en una base de datos:

- **Bloqueo pesimista.** Se bloquea un registro o una tabla inmediatamente, en cuanto se solicita el bloqueo. Con el bloqueo pesimista se garantiza que el registro será siempre actualizado.
- **Bloqueo optimista.** El acceso al registro o tabla solo se cierra en el momento en que los cambios realizados a ese registro se realizan en el disco. Esta situación solo es apropiada cuando hay menos posibilidad de que alguien necesite acceder al registro mientras está bloqueado.

7.2. Bloqueos compartidos y exclusivos

La base de datos utiliza dos tipos de bloqueos.

- **Bloqueo exclusivo.** 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 que puede modificar el recurso hasta que es liberado. Cualquier otra transacción debe esperar en cola.

```
LOCK TABLE nombreTabla IN EXCLUSIVE MODE
```

- **Bloqueo compartido.** Permite que el recurso asociado sea compartido, dependiendo de la operación en la que se encuentra involucrado. Varios usuarios que lean datos pueden compartir datos realizando bloqueos compartidos.

Por ejemplo, usando la sentencia SELECT... FOR UODATE, se obtiene un bloqueo exclusivo del registro y un bloqueo compartido de la tabla. El bloqueo del registro permite a otras sesiones modificar cualquier otro registro distinto al bloqueado. El bloqueo de la tabla previene que otras sesiones modifiquen la estructura de la tabla.

7.3. Bloqueos automáticos

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.

Los bloqueos que realiza Oracle se dividen en:



- **Bloqueos DML:** Protegen los datos, garantizando la integridad de los datos accedidos de forma concurrente por varios usuarios. Las sentencias DML: INSERT, UPDATE o DELETE realizan bloqueo exclusivo por las filas afectadas en su cláusula WHERE.
- **Bloqueos DDL:** Protegen la definición del esquema de un objeto mientras una operación DDL actúa sobre él. Se realizan de manera automática en cualquier transacción DDL. Los usuarios no pueden solicitar bloqueo explícito DDL.
- **Bloqueos del sistema:** Oracle usa varios tipos de bloqueo del sistema para proteger la bases de datos interna y estructuras de memoria.

7.4. Bloqueos manuales

Se pueden omitir los mecanismo de bloqueo por defecto de Oracle, usando bloqueos manuales:

- En aplicaciones que requieren consistencia en la consulta de datos a nivel transacciones o en lecturas repetitivas.
- En aplicaciones que requieren que una transacción tenga acceso exclusivo a un recurso con el fin de que no tenga que esperar a que otras terminen.

La cancelación de bloqueos se realiza a nivel de sesión o de transacción. A nivel sesión se realiza con ALTER SESSION, a nivel transacción las transacciones que incluyan las siguientes sentencias omiten el bloqueo por defecto:

- SET TRANSACTION ISOLATION LEVEL
- LOCK TABLE
- SELECT... FOR UPDATE.

Estos bloqueos terminan cuando termina la transacción.

7.5. Interbloqueos

Ejemplo de interbloqueo entre dos tablas con cancelación de bloqueo automático a nivel transacción.

```
CREATE TABLE Gallina(
    idGallina number(3) PRIMARY KEY,
    idHuevo number(3) REFERENCES Huevo(idHuevo));

CREATE TABLE Huevo(
    idHuevo number(3) PRIMARY KEY,
    idGallina number(3) REFERENCES Gallina(idGallina));
```

Si intentamos crear dos tablas con referencias cruzadas, dará error, porque cualquiera de las dos tablas necesita que la otra esté creada para poder crearse. Para solucionar este problema consideramos la transacción como DEFERRED, la comprobación de restricciones se hará en diferido.

Podemos crear ambas tablas sin especificar la FK y a continuación ALTER table para añadir la referencia a otra tabla:

```
CREATE TABLE Gallina(
    idGallina number(3) PRIMARY KEY,
    idHuevo number(3) );

CREATE TABLE Huevo
```

```

idHuevo number(3) PRIMARY KEY,
idGallina number(3);

ALTER TABLE Gallina ADD CONSTRAINT fkRefHuevo
FOREIGN KEY(idHuevo) REFERENCES Huevo(idHuevo);

ALTER TABLE Huevo ADD CONSTRAINT fkRefGallina
FOREIGN KEY(idGallina) REFERENCES Gallina(idGallina);

```

Si intentamos insertar un registro en cada una de las tablas dará error, ya que al insertar en la primera tabla aún no tenemos el dato de la segunda y viceversa.

Para solucionarlo podemos añadir INITIALLY DEFERRED después de REFERENCES en la orden ALTER TABLE. La comprobación se hará después de commit y aunque aún no existan datos en la segunda tabla nos permite insertar registro en la primera.

```

ALTER TABLE Gallina ADD CONSTRAINT fkRefHuevo
FOREIGN KEY(idHuevo) REFERENCES Huevo(idHuevo) INITIALLY DEFERRED;

ALTER TABLE Huevo ADD CONSTRAINT fkRefGallina
FOREIGN KEY(idGallina) REFERENCES Gallina(idGallina) INITIALLY DEFERRED;

INSERT INTO Gallina VALUES(4, 5);

INSERT INTO Huevo VALUES(5, 4);

COMMIT;

```

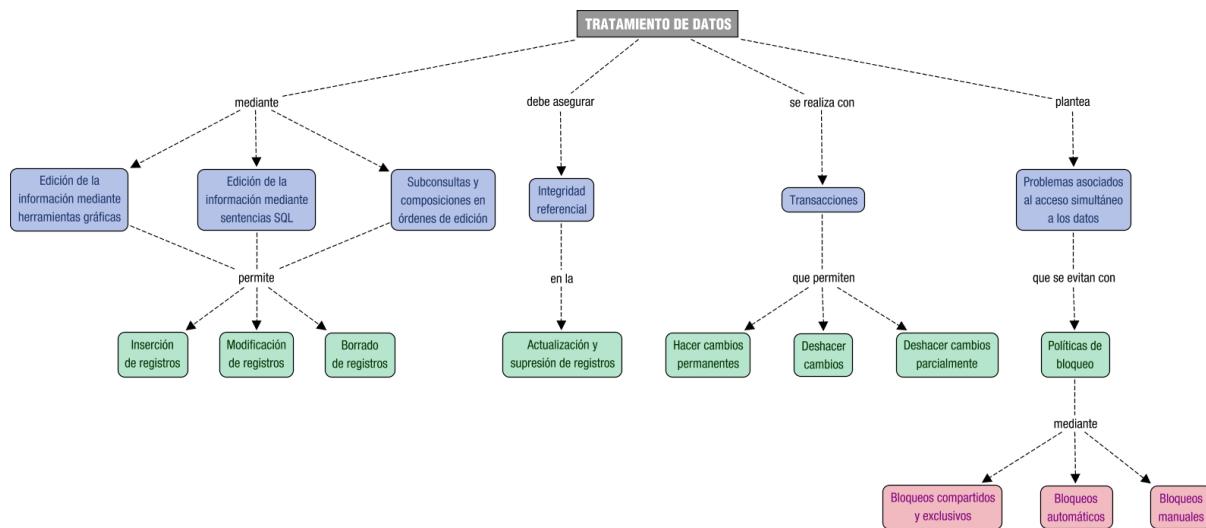
Por último, sentencias para borrar las constraints y las tablas tras el ejemplo:

```

ALTER TABLE Huevo DROP CONSTRAINT fkRefGallina;
ALTER TABLE Gallina DROP CONSTRAINT fkRefHuevo;
DROP TABLE Huevo;

```

Mapa conceptual





6. Programación de bases de datos

Class	Bases de datos
Column	(X) Xerach Casanova
Last Edited time	@Mar 29, 2021 9:25 PM

- 1. Introducción
- 2. Conceptos básicos
 - 2.1. Unidades léxicas (I)
 - 2.1.1. Unidades léxicas (II)
 - 2.2. Tipos de datos simples, variables y constantes
 - 2.2.1. Subtipos
 - 2.2.2. Variables y constantes
 - 2.3. El bloque PL/SQL
 - 2.4. Estructuras de control (I)
 - 2.4.1 Estructuras de control (II). Bucles
 - 2.5. Manejo de errores (I)
 - 2.5.1. Manejo de errores (II)
 - 2.5.2. Manejo de errores (III)
 - 2.5.3. Manejo de errores (IV)
 - 2.6. Sentencias SQL en programas PL/SQL
- 3. Tipos de datos compuestos
 - 3.1. Registros
 - 3.2. Colecciones. Arrays de longitud variable
 - 3.2.1 Colecciones. Tablas anidadas
 - 3.3. Cursores
 - 3.3.1. Cursores explícitos
 - 3.3.2. Cursores variables
- 4. Abstracción en PL/SQL
 - 4.1. Subprogramas
 - 4.1.1. Almacenar subprogramas en la base de datos
 - 4.1.2. Parámetros de los subprogramas
 - 4.1.3. Sobrecarga de subprogramas y recursividad
 - 4.2. Paquetes
 - 4.2.1. Ejemplos de utilización del paquete DBMS_OUTPUT
 - 4.3. Objetos
 - 4.3.1. Objetos. Funciones mapa y funciones de orden
- 5. Disparadores
 - 5.1. Definición de disparadores de tablas
 - 5.2. Ejemplos de disparadores
- 6. Interfaces de programación de aplicaciones para lenguajes externos
- Anexo 1. Caso de estudio.
- Anexo II - Excepciones predefinidas en Oracle
- Anexo III - Evaluación de los atributos de un cursor explícito
- Anexo IV - Paso de parámetros a subprogramas
- Anexo V - Sobrecarga de subprogramas
- Anexo VI. Ejemplo de recursividad
- Anexo VII. Ejemplo de paquete
- Mapa conceptual

1. Introducción

PL/SQL es un lenguaje procedimental estructurado en bloques, que amplía la funcionalidad de SQL, con él podemos usar sentencias SQL para manipular datos y sentencias de control de flujo para procesar los datos, combinando la potencia de SQL con la potencia de lenguajes procedimentales para procesar datos.

Fue creado por Oracle, pero lo utilizan todos los gestores de bases de datos.

Se pueden definir variables, constantes, funciones, procedimientos, capturar errores, anidar cualquier número de bloques, etc. También podemos usar disparadores.

El motor de PL/SQL acepta como entrada bloques PL/SQL o subprogramas, ejecuta las sentencias procedimentales y envía sentencias SQL al servidor de base de datos.

La gran ventaja es el mejor rendimiento en entornos de red cliente-servidor, ya que se envían bloques de PL/SQL desde el cliente al servidor y no se envían las sentencias SQL una a una.

2. Conceptos básicos

2.1. Unidades léxicas (I)

PL/SQL no es sensible a las mayúsculas, podemos escribir en mayúsculas y minúsculas excepto cuando hablamos de literales de tipo cadena o de tipo carácter.

Cada unidad léxica se puede separar por espacios (debe estar separada por espacios si se trata de 2 identificadores), por saltos de línea o por tabuladores, para aumentar la legibilidad.

Ejemplos equivalentes:

```
IF A=CLAVE THEN ENCONTRADO:=TRUE;ELSE ENCONTRADO:=FALSE;END IF;
```

```
if a=clave then encontrado:=true;else encontrado:=false;end if;
```

```
IF a = clave THEN
    encontrado := TRUE;
ELSE
    encontrado := FALSE;
END IF;
```

Las unidades léxicas se clasifican en:

Delimitadores

Se utilizan para representar operaciones entre tipos de datos, delimitar comentarios, etc.

Delimitadores Simples.		Delimitadores Compuestos.	
Símbolo.	Significado.	Símbolo.	Significado.
+	Suma.	**	Exponenciación.
%	Indicador de atributo.	<>	Distinto.
.	Selector.	j≠	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.		

2.1.1. Unidades léxicas (II)

Identificadores

Se utilizan para nombrar elementos y debemos tener los siguientes aspectos:

- Es una letra seguida opcionalmente de letras, números, \$, _, #.
- No se puede utilizar como identificador una palabra reservada.
- Podemos definir los identificadores acotados, en los que se puede utilizar cualquier carácter, con una longitud máxima de 30 y deben estar delimitados por ". Ejemplo: "X*Y".
- Existen algunos identificadores predefinidos con un significado especial para dar sentido sintáctico a nuestros programas. Son palabras reservadas: IF, THEN, ELSE...
- Algunas palabras reservadas para PL/SQL no lo son para SQL, por lo que podemos tener una tabla con una columna llamada 'type' que nos dará error de compilación, para solucionarlo se acota: "type".

Literales

Se utilizan en las comparaciones de valores para asignar valores concretos a los identificadores, que actúan como variables o constantes.

- Los literales numéricos se expresan en notación decimal o exponente: 234, +341, 2e3, -2E-3, 7.45, 8.1e3
- Los literales tipo carácter se delimitan con comillas simples.
- Los literales lógicos son TRUE y FALSE.
- El literal NULL indica que la variable no tiene valor.

Comentarios

No tienen efectos sobre el código pero ayudan a los programadores a recordar lo que se está haciendo. Existen dos tipos

- Comentarios de una línea, con delimitador — (doble guión).
- Comentario de varias líneas con /**/.

2.2. Tipos de datos simples, variables y constantes

En PL/SQL contamos con todos los tipos de datos simples utilizados en SQL y algunos más. Los más utilizados son:

Numéricos

- **BINARY_INTEGER**: Tipo numérico de rango entre -2147483647 y 2147483647, Además se definen algunos subtipos: NATURAL, NATURALN, POSITIVE, POSITIVEN, SIGNTYPE.
- **NUMBER**: Tipo numérico que almacena números racionales. Se puede especificar su escala (-84 a 127) y su precisión (1 a 38), La escala indica cuándo se redondea y hacia dónde. Ejemplo: escala = 2: 8.234 → 8.23 / escala = -3: 7689 → 8000. Si definimos un NUMBER(6,2) indicamos que son 6 dígitos numéricos de los cuales 2 son decimales. También se definen algunos subtipos como: DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, INT, NUMERIC, REAL SMALLINT.
- **PLS_INTEGER**: Tipo numérico, con rango igual al BINARY_INTEGER. Su representación es distinta y las operaciones llevadas a cabo con ellos serán más eficientes.

Alfanumérico

- **CHAR(n)**: Array de n caracteres, máximo 2000 bytes, la longitud por defecto es 1, ocupa en memoria n caracteres.
- **LONG**: Array de caracteres con un máximo de 32760 bytes.
- **RAW**: Array de bytes con número máximo de 2000.
- **LONGRAW**: Array de bytes con un máximo de 32760.
- **VARCHAR2(n)**: Almacena cadenas de longitud variable con un máximo de 32760. Ocupa en memoria n caracteres.

Grandes objetos

- **BFILE**: Puntero a un fichero del S.O.
- **BLOB**: Objeto binario con capacidad de 4GB.

- GLOB: Objeto carácter con capacidad de 2GB.

Otros

- BOOLEAN: TRUE/FALSE
- DATE: almacena valores de día y hora, desde el 1 de enero de 4712 a.c. hasta el 31 de diciembre de 4712 d.c.

Atributo %TYPE

Si utilizamos el atributo %TYPE nos aseguramos cuando declaramos la variable que tiene el mismo tipo de datos que la variable especificada. Normalmente se relaciona con una columna de la BD indicando el nombre de la tabla de la bd y el de la columna. Si se hace referencia a un variable declarada anteriormente, se indica el nombre de la variable previamente declarada a la variable por declarar.

El uso de este atributo ayuda a que si se cambia la definición de una columna no tendremos que cambiar la declaración de la variable.

Ejemplos:

```
nombre_oficina oficinas.nombre%type; -- la variable nombre_oficina tomará el mismo tipo de dato que la columna nombre de la tabla ofi
vx number(5,2); -- Variable numérica de 5 digitos, dos de los cuales son decimales
vy vx%type; -- la variable vy tomará el mismo tipo de dato que tenga la variable vx
```

2.2.1. Subtipos

Los subtipos nos permiten definir subtipos de tipos de datos para darles un nombre distinto y aumentar legibilidad de nuestros programas. Las operaciones aplicables a estos subtipos son las mismas que con los que los preceden.

```
SUBTYPE subtipo IS tipo_base;
```

subtipo es el nombre que le damos a nuestro subtipo y tipo_base es cualquier tipo de dato que le damos a nuestro tipo de dato PL/SQL

Cuando especificamos el tipo base, se puede utilizar el modificador %TYPE para indicar el tipo de dato de una variable o columna de la bd y %ROWTYPE para especificar el tipo de un cursor o tabla de una base de datos:

```
SUBTYPE id_familia IS familias.identificador%TYPE; -- Permite nombrar a la columna identificador de la tabla familias con el nombre id_
SUBTYPE agente IS agentes%ROWTYPE; -- Permite nombrar a las filas de la tabla agentes con el nombre agente
```

No se pueden restringir los subtipos, pero podemos seguir el mismo efecto por medio de una variable auxiliar:

```
BTYPE apodo IS varchar2(20);          --illegal
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;

```

2.2.2. Variables y constantes

Para declarar variables o constantes se pone el nombre de la variable, seguido del tipo de dato y opcionalmente se le asigna un valor con el operador `:=`, si es una constante anteponemos la palabra `CONSTANT` al tipo de dato. Se puede sustituir el operador de asignación por la palabra reservada `DEFAULT`, o podemos forzar que no sea nula con `NOT NULL` después del tipo y antes de la asignación, la cual es obligatoria al declararla o se lanza una excepción `VALUE_ERROR`.

```

id SMALLINT;
hoy DATE := sysdate;
p CONSTANT REAL:= 3.1415;
id SMALLINT NOT NULL; --illegal, no está inicializada
id SMALLINT NOT NULL := 9999; --legal
nombre varchar2(30):= ;'Alejandro Magno';
num INTEGER DEFAULT 4; -- también se puede utilizar DEFAULT para inicializar una variable

```

Conversión de tipos

Aunque existe la conversión implícita de tipos para tipos parecidos siempre es aconsejable utilizar la conversión explícita por medio de funciones de conversión.

Precedencia de operadores

Se utilizan para realizar operaciones aritméticas, si dos operadores tienen la misma precedencia se evalúa de izquierda a derecha. Es aconsejable utilizar paréntesis para alterar la precedencia de los mismos.

Operador.	Operación.
<code>**, NOT</code>	Exponenciación, negación lógica.
<code>+, -</code>	Identidad, negación.
<code>*, /</code>	Multiplicación, división.
<code>+, -, </code>	Suma, resta y concatenación.
<code>=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN</code>	Comparaciones.
<code>AND</code>	Conjunción lógica
<code>OR</code>	Disyunción lógica.

2.3. El bloque PL/SQL

La unidad básica en PL/SQL es el bloque, que consta de tres zonas:

- **Declaraciones:** definición de variables, constantes, cursosres y excepciones.
- **Procesos:** zona donde se realiza el proceso en sí, conteniendo sentencias ejecutables.
- **Excepciones:** zona de manejo de errores en tiempo de ejecución.

La sintaxis es:

```

[DECLARE
  [Declaración de variables, constantes, cursores y excepciones]]
BEGIN
  [Sentencias ejecutables]
[EXCEPTION
  Manejadores de excepciones]
END;

```

Los bloques se pueden anidar a cualquier nivel, el ámbito y la visibilidad de las variables es la misma que un lenguaje procedural. En el siguiente ejemplo, aux es variable global, pero también está declarada como variable local y la visibilidad dominante es la de la variable local, por tanto, vale 5

```

DECLARE
  aux number := 10;-- Variable global

```

```

BEGIN
    DECLARE
        aux number := 5;      -- Variable local al bloque donde es definida
    BEGIN
        ...
        IF aux = 10 THEN    --evalúa a FALSE, no entraría
        ...
    END;
END;

```

2.4. Estructuras de control (I)

Como en todos los lenguajes de programación existen dos: condicionales e iterativas.

Control condicional. sentencia if

Sus variantes son

- **Sentencia IF -THEN.** Si la evaluación de la condición es TRUE se ejecuta la sentencia entre el then y el final de la sentencia.

```

IF condicion THEN
secuencia_de_sentencias;
END IF;

```

Ejemplo:

```

SET SERVEROUTPUT ON
DECLARE a integer:=10;
B integer:=7;
BEGIN
IF a>b
THEN dbms_output.put_line(a || ' es mayor'); -- Como la función put_line solo imprime un valor utilizamos la concatenación || para qu
END IF;
END;

```

- **Sentencia IF-THEN-ELSE.** Si la evaluación es TRUE se ejecuta la primera secuencia de sentencias y si no, la segunda.

```

IF condicion
THEN Secuencia_de_sentencias1;
ELSE Secuencia_de_sentencias2;
END IF;

```

```

DECLARE
a integer:=10;
b integer:=17;
BEGIN
IF a>b THEN
    dbms_output.put_line(a || ' es mayor');
ELSE
    dbms_output.put_line(b || ' es mayor o iguales');
END IF;
END;
/

```

- **Sentencia IF-THEN-ELSIF:** se trata de una condición múltiple, si la condición, podemos poner cuantos ELSIF queramos, se ejecutará la sentencia de la condición que sea TRUE

```

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; -- Lanza un error de ejecución
END IF;
/

```

Sentencia CASE

Representa n sentencias if anidadas y es más fácil de interpretar cuando se compara con varios valores, permitiendo sustituir a las sentencias if encadenadas.

- Utilizando un selector y un manejador WHEN para cada posible valor de selector

```

CASE selector
    WHEN expression1 THEN
        ordenes;
    [ WHEN expression2 THEN
        ordenes;
    ...
    [WHEN expression THEN
        ordenes;]
    [ ELSE
        ordenes;]
END CASE ;

```

- Utilizando condiciones de búsqueda:

```

CASE
WHEN condición1 THEN
ordenes;
[ WHEN condición2 THEN
ordenes;
...
WHEN condiciónN THEN
ordenes;]
[ ELSE
ordenes;]
END CASE ;

```

Antes de terminar la sentencia CASE se puede especificar ELSE por si no se ha cumplido ninguna de las condiciones anteriores. Todas las condiciones se analizan en el orden listado. Desde que se encuentra una condición verdadera se deja de analizar el resto.

```

DECLARE
    nota INTEGER:=8; -- Se podria especificar nota INTEGER:=&nota
BEGIN
    CASE
        WHEN nota in(1,2) THEN
            DBMS_OUTPUT.PUT_LINE('Muy deficiente');
        WHEN nota in (3,4) THEN
            DBMS_OUTPUT.PUT_LINE('Insuficiente');
        WHEN nota = 5 THEN
            DBMS_OUTPUT.PUT_LINE('Suficiente');
        WHEN nota=6 THEN
            DBMS_OUTPUT.PUT_LINE('Bien');
        WHEN nota in(7,8) THEN
            DBMS_OUTPUT.PUT_LINE('Notable');
        WHEN nota in (9,10) THEN
            DBMS_OUTPUT.PUT_LINE('Sobresaliente');
        ELSE
            DBMS_OUTPUT.PUT_LINE('Error, no es una nota');
    END CASE;
END;
/

```

Para pedir datos por teclado se utilizan variables de sustitución escribiendo el carácter especial '&' y a continuación un identificador, si el dato a introducir es alfanumérico se escribe entre comillas simples:

```
DECLARE
    cadena varchar2(25) :='&cad';
BEGIN
    DBMS_OUTPUT.PUT_LINE(cadena);
END;/
```

2.4.1 Estructuras de control (II). Bucles

Control iterativo

Ejecuta sentencias un determinado número de veces:

- **LOOP:** La forma simple es el bucle infinito:

```
LOOP
    secuencia_de_sentencias;
END LOOP;
```

- **EXIT.** Con ella forzamos a terminar el bucle. No fuerza nunca la salida de un bloque, solo del bucle.

```
LOOP
    ...
    IF encontrado = TRUE THEN
        EXIT;
    END IF;
END LOOP;
```

```
DECLARE
    a integer :=1;
BEGIN
    LOOP
        dbms_output.put_line(a);
        IF a>9 THEN
            EXIT;
        END IF;
        a:=a+1;
    END LOOP;
END;
/
```

- **EXIT WHEN condición.** Fuerza a salir del bucle si se cumple una determinada condición:

```
LOOP
    ...
    EXIT WHEN encontrado;
END LOOP;
```

```
DECLARE
    a integer :=1;
BEGIN
    LOOP
        dbms_output.put_line(a);
        EXIT WHEN a>9;
        a:=a+1;
    END LOOP;
END;
/
```

- **WHILE LOOP:** ejecuta la secuencia de sentencias mientras la condición sea cierta.

```
WHILE condicion LOOP
    Secuencia_de_sentencias;
END LOOP;
```

```

DECLARE
    a integer :=1;
BEGIN
    WHILE a<10 LOOP
        dbms_output.put_line(a);
        a:=a+1;
    END LOOP;
END;/
```

FOR-LOOP. El bucle itera mientras el contador se encuentre en el rango definido.

```

FOR contador IN [REVERSE] limite_inferior..limite_superior LOOP
    Secuencia_de_sentencias;
END LOOP;
```

```

BEGIN
    FOR a IN 1..10 LOOP -- ascendente de uno en uno
        dbms_output.put_line(a);
    END LOOP;
    FOR a IN REVERSE 1..10 LOOP -- descendente de uno en uno
        dbms_output.put_line(a);
    END LOOP;
END;
/
```

2.5. Manejo de errores (I)

Cualquier situación de error se llama excepción, cuando se detecta se lanza una excepción, la ejecución normal se para y el control se transfiere a la parte de manejo de excepciones, la cual está etiquetada como EXCEPTION y cuenta con sentencias llamadas manejadores de excepciones.

Manejadores de excepciones

Sintaxis:

```

WHEN nombre_excepcion THEN
    <sentencias para su manejo>
    ...
WHEN OTHERS THEN
    <sentencias para su manejo>
```

Ejemplo:

```

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;
/
```

Las excepciones las puede definir el usuario o están definidas internamente, las predefinidas se lanzan automáticamente asociadas a un error Oracle, las definidas por el usuario se deben definir y lanzar explícitamente.

Para definir nuestras excepciones, lo hacemos en la parte DECLARE de cualquier bloque. Las podemos lanzar explícitamente on la sentencia RAISE nombre_excepción.

Excepciones definidas por el usuario

Sintaxis:

```

DECLARE
    nombre_excepcion EXCEPTION;
```

```

BEGIN
  ...
  RAISE nombre_excepcion;
  ...
END;

```

Ejemplo

```

DECLARE
    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;

```

2.5.1. Manejo de errores (II)

Detalles de uso de las excepciones:

- Al igual que en las variables, las excepciones locales redefinidas (que ya eran globales para el bloque), prevalece la definición local y no podremos capturar esa excepción a menos que el bloque donde estaba fuese un bloque nombrado, con lo cual podremos capturarla usando: nombre_bloque.nombre_excepcion
- Las excepciones predefinidas están definidas globalmente, ni necesitamos, ni debemos redefinirlas.

```

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 se puede continuar por la siguiente sentencia que se 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
    --explicativamente en el manejador
END;

```

Pero podemos encerrar la sentencia de dentro de un bloque y así capturar las posibles excepciones para continuar con otras 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);
  END;

```

```

    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;

```

2.5.2. Manejo de errores (III)

- En vez de encapsular las excepciones en bloque para manejar cada una de ellas individualmente, podemos utilizar una variable localizadora para saber que sentencia la lanzó, pero de esta manera no podremos continuar con la siguiente sentencia a la excepción lanzada.

```

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 capturada por un manejador de excepción apropiado, ésta es tratada y después el control se devolverá al entorno. Se puede dar que la excepción sea manejada en un bloque superior a falta de manejadores en bloques internos, propagándose al bloque superior, y así sucesivamente hasta que sea manejada o que no queden bloques superiores y el control se devuelva al entorno.

2.5.3. Manejo de errores (IV)

Oracle también permite que lancemos nuestros propios mensajes de error a las aplicaciones y asociarlos a un código de error que Oracle reserva.

```
RAISE_APPLICATION_ERROR(error_number, message [, (TRUE|FALSE)]);
```

error_number es un número entero comprendido entre -20000 y -20099 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). A este procedimiento podemos llamarlo desde un subprograma.

No hay excepciones predefinidas asociadas a todos los posibles errores, pero podemos asociar excepciones definidas propias a errores Oracle:

```
PRAGMA_INIT( nombre_excepcion, error_Oracle )
```

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. Una sentencia ejecutada correctamente devuelve 0 en SQLCODE, en caso contrario devuelve 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;

```

2.6. Sentencias SQL en programas PL/SQL

Para trabajar con SQL dentro de un programa trabajaremos con SQL embebido incrustado, los datos devueltos de SQL los guardamos en variables y estructuras definidas para utilizarlas como convenga, añadiendo alguna cláusula en el SELECT.

PL/SQL soporta DML y comandos de control de transacciones. No soporta directamente DDL (CREATE TABLE, ALTER TABLE, etc.), ya que estas instrucciones son sentencias dinámicas en SQL. Tampoco soporta DCL (GRANT O REVOKE). Se puede utilizar SQL dinámico para ejecutarlas.

Si queremos recuperar datos de la BD utilizamos select:

```

SELECT lista_campos INTO {nombre_variable[, nombre_variable]...| nombre_registro}
FROM tabla
[WHERE condición];

```

- La lista_campos contiene al menos una columna y puede incluir expresiones SQL, funciones de fila o de grupo.
- La cláusula INTO es obligatoria y se especifica entre SELECT Y FROM
- nombre_variable, donde se guarda el valor recuperado y se especifican tantas variables como campos indicados.
- nombre_registro, un dato compuesto de PL/SQL donde se guardan todos los valores recuperados.

Las instrucciones SELECT deben devolver una sola fila, una consulta que devuelve más de una fila o ninguna genera un error de tipo (TOO_MANY_ROWS o NO_DATA_FOUND). Para recuperar más de una fila usamos estructura de datos llamada cursor.

INSERT, UPDATE y DELETE se ejecutan de la misma manera que en SQL.

3. Tipos de datos compuestos

3.1. Registros

Un registro es un grupo de elementos asociados, referenciados por un único nombre y almacenados en campos, cada uno de ellos tiene su propio nombre y dato.

Hacen que la información sea más fácil de organizar. Por ejemplo. una dirección podría ser un registro con campos como calle, número, piso, puerta...

```

TYPE nombre_tipo IS RECORD (decl_campo[, decl_campo] ...);

```

donde

```
decl_campo := nombre tipo [[NOT NULL] {:=|DEFAULT} expresion]
```

El tipo de campo es cualquier tipo válido en PL/SQL excepto REF CURSOR.

```
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 utilizamos el operador punto.

```
...
mi_direccion.calle := 'Ramirez Arellano';
mi_direccion.numero := 15;
...
```

Para asignar un registro a otro, deben ser del mismo tipo, no basta que tengan el mismo número de campos y se emparejen uno a uno. Tampoco podemos comparar registros aunque sean del mismo tipo, ni comprobar si éstos son nulos.

```
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; -- tendrá los mismos campos que tenga la tabla familias
  mi_fam familia;
  mi_fam_aux familia_aux;
  mi_fam_fila familia_fila;
BEGIN
  ...
  mi_fam := mi_fam_aux;           --illegal
  mi_fam := mi_fam_fila;          --legal
  IF mi_fam IS NULL THEN ...     --illegal
  IF mi_fam = mi_fam_fila THEN ... --illegal
  SELECT * INTO mi_fam FROM familias ... --legal
  INSERT INTO familias VALUES (mi_fam_fila); --illegal
  ...
END;/
```

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 ella.

En PL/SQL las colecciones solo tienen una dimensión y ofrecen 2 clases: arrays de longitud variable y tablas anidadas.

Arrays de longitud variable

Son los elementos de tipo VARRAY. A la hora de declararlos se indica su tamaño máximo y el array puede ir creciendo dinámicamente hasta alcanzar su tamaño máximo.

```
TYPE nombre IS {VARRAY | VARYING} (tamaño_máximo) OF tipo_elementos [NOT NULL];
```

tamaño_máximo es 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 es nulo y debemos inicializarlo para empezar a usarlo. Para ello se usa 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, nu
```

Podemos usar constructores vacíos:

```
familias_hijas2 familias_hijas := familias_hijas();
```

Para referencias elementos en un VARRAY utilizamos sintaxis nombre_colección(subíndice). Si una función devuelve un VARRAY podemos utilizar la sintaxis: nombre_funcion(listaparametros)(subindice)

```
IF familias_hijas1(i).identificador = 100 THEN ...
IF dame_familias_hijas(10)(i).identificador = 100 THEN ...
```

Un VARRAY se puede asignar 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;         --illegal
    ...
END;
```

Para extender un VARRAY utilizamos el método EXTEND. Sin parámetros extendemos en 1 elemento nulo. EXTEND(n) añade n elementos nulos al VARRAY y EXTEND(n,i) añade n copias del i-ésimo elemento.

COUNT nos dice 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 devuelven el antecesor y sucesor del elemento.

Al trabajar con VARRAY podemos hacer que salte alguna de las siguientes excepciones por mal uso:
COLLECTION_IS_NULL, SUBSCRIPT_BEYOND_COUNT, SUBSCRIPT_OUTSIDE_KLIMIT Y VALUE_ERROR.

Ejemplos de uso:

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;
        mi_tab(i) := calcular_elemento(i);
    END LOOP;
    ...
END;
```

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
        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;

```

Possibles 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
    v_numeros(-1) := 5; --lanzaría SUBSCRIPT_OUTSIDE_LIMIT v_numeros('A') := 25; --
    lanzaría VALUE_ERROR
    ...

```

3.2.1 Colecciones. Tablas anidadas

Las tablas anidadas son colecciones de elementos que no tienen límite superior fijo, pueden aumentar dinámicamente su tamaño y además podemos borrar elementos individuales.

```
TYPE nombre IS TABLE OF tipo_elementos [NOT NULL];
```

En tipo_elementos existen las mismas restricciones que en los VARRAY.

También son nulas al declararlas y se deben inicializar antes de usarlas.

```
TYPE hijos IS TABLE OF agente;
hijos_fam hijos := hijos( agente(...) ...);
```

También se pueden usar constructores nulos y para extenderlas también se hace igual que en los VARRAY.

COUNT nos dice el número de elementos, el cual no tiene por qué coincidir con LAST.

LIMIT no tiene sentido y devuelve NULL. EXISTS(n) devuelve TRUE si existe y FALSE si ha sido borrado.

FIRST devuelve el primer elemento, que no tiene que ser 1, ya que se puede borrar cualquier elemento del principio.

PRIOR y NEXT nos dicen el antecesor y el sucesor ignorando los 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, 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 devuelve FALSE.

Al trabajar con tablas anidadas podemos recibir las siguientes excepciones: COLLECTION_IS_NULL, NO_DATA_FOUND, SUBSCRIPT_BEYOND_COUNT y VALUE_ERROR

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;

```

Possibles excepciones de 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;

```

3.3. Cursores

El cursor es una estructura que almacena el conjunto devuelto por una consulta a la base de datos.

Oracle usa áreas de trabajo para ejecutar consultas SQL y almacenar la información procesada. Hay 2 clases de cursores: implícitos y explícitos. PS/SQL declara implícitamente un cursor para todas las DML, incluyendo las que devuelven una fila. Para todas las que devuelven más de una, se debe declarar explícitamente un cursor para procesar las filas individualmente.

Cursors implícitos.

Con un cursor implícito no podemos usar las sentencias OPEN, FETCH Y CLOSE para controlarlo, pero podemos usar los atributos del cursor para obtener información sobre las sentencias SQL recientemente ejecutadas.

Atributos de un cursor

Cada cursor tiene 4 atributos, pueden ser usados por PL/SQL pero no en SQL.

- %FOUND. Después de que el cursor esté abierto y antes del primer FETCH %FOUND devuelve NULL. Después de %FOUND devuelve TRUE y si el último FETCH ha devuelto una fila y FALSE en caso contrario. Para cursos 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 fila, en caso contrario devuelve FALSE.
- %NOTFOUND. Es el caso contrario a %FOUND.
- %ISOPEN. Devuelve TRUE si el cursor está abierto y FALSE si está cerrado, para cursos implícitos Oracle cierra automáticamente, así que siempre es FALSE.
- %ROWCOUNT. Para un cursor abierto y antes del primer FETCH %ROWCOUNT devuelve 0. Después de cada FETCH %ROWCOUNT es incrementado y evalúa al número de filas que hemos procesado. Para cursos implícitos %ROWCOUNT evalúa el número de filas afectadas por un INSERT, UODATE, O DELENTE, o el número de filas devueltas por SELECT... INTO.

3.3.1. Cursors explícitos

Si una consulta devuelve múltiples filas debemos declarar explícitamente un cursor para procesarlas. Para ello se les da un nombre y se asocian a una consulta:

```
CURSOR nombre_cursor [(parametro [, parametro] ...)] [RETURN tipo_devuelto] IS sentencia_select;
```

tipo_devuelto debe representar un registro o una fila de una tabla de la bd 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 ...
```

Un cursor puede tomar parámetros que pueden aparecer en la consulta asociada como constantes. Los parámetros son de entrega, ya que un cursor no puede devolver valores en los parámetros actuales. A un parámetro de cursor no se puede imponer la restricción NOT NULL

```
CURSOR c1 (cat INTEGER DEFAULT 0) IS SELECT * FROM agentes WHERE categoria = cat;
```

Al abrir un cursor, se ejecuta la consulta asociada y se identifica el conjunto resultado, que serán todas las filas emparejadas al criterio de búsqueda de la consulta.

Un cursor se abre así:

```
OPEN nombre_cursor [(parametro [, parametro] ...)];
```

```
OPEN cAgentes;
OPEN c1(1);
OPEN c1;
```

FETCH devuelve una fila del conjunto resultado. Después de cada FETCH el cursor avanza a la siguiente fila.

```
FETCH cFamilias INTO mi_id, mi_nom, mi_fam, mi_ofi;
```

Por cada valor de columna en la consulta select del cursor, debe haber una variable que corresponda en la lista de variables después de INTO.

Se procesan los cursores por medio de bucles.

```
BEGIN
...
OPEN cFamilias;
LOOP
    FETCH cFamilias INTO mi_id, mi_nom, mi_fam, mi_ofi;
    EXIT WHEN cFamilias%NOTFOUND;
    ...
END LOOP;
CLOSE cFamilias;
...
END;
```

Cuando cerramos el cursor se puede reabrir, pero no se puede realizar ninguna operación con el cursor cerrado, ya que se lanza la excepción INVALID_CURSOR.

Otro bucle consiste en declarar una variable índice definida como %ROWTYPE para el cursor, se abre el cursor, se extraen los valores de cada fila del cursor, se almacena en la variable índice y se cierra 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;
```

3.3.2. Cursos variables

También podemos definir cursos variables, que son como punteros a cursos, y podemos usarlos para referirnos a cualquier tipo de consulta. Los cursos son estáticos y los cursos variables son dinámicos.

Para declarar un cursor variable:

- Definimos un tipo REF cursor y declaramos una variable de ese tipo.

```
TYPE tipo_cursor IS REF CURSOR RETURN agentes%ROWTYPE;
cAgentes tipo_cursor;
```

- Una vez definido, debemos asociarlo a una consulta, esto se hace en la parte de ejecución y no en la declarativa, lo hacemos con la sentencia OPEN-FOR utilizando la siguiente sintaxis:

```
TYPE tipo_cursor IS REF CURSOR RETURN agentes%ROWTYPE;
cAgentes tipo_cursor;
```

Los cursos variables no pueden tomar parámetros.

También podemos usar varios OPEN-FOR Para abrir el mismo cursor variable con diferentes consultas y no necesitamos cerrarlo antes de reabrirlo. Cuando reabrimos un cursor variable para una consulta diferente, la consulta previa se pierde.

Una vez abierto, su manejo es idéntico al de un cursor.

```
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;
```

También se puede utilizar el bucle FOR con cursos variables definidos dentro del mismo bucle

```
for va_cursor in (select ...) loop
  ...
end loop;
```

4. Abstracción en PL/SQL

PL/SQL permite definir funciones y procedimientos, además podemos agrupar todas las que tengan relación en paquetes. También permite utilización de objetos.

4.1. Subprogramas

Son bloques de código PL/SQL, referenciados bajo un nombre y crean una acción determinada. Le podemos pasar parámetros y los podemos invocar.

Suelen estar almacenados en la base de datos o encerrados en otros bloques. Si el programa está en la bd, podemos invocarlo con los permisos suficientes, y si está encerrado en el bloque podremos invocarlo si tenemos visibilidad sobre el mismo.

Existen dos subprogramas: funciones (devuelven valor) y procedimientos (no devuelven valor)

Sintaxis de funciones:

```
FUNCTION nombre [(parametro [, parametro] ...)]
    RETURN tipo_dato IS
    [declaraciones_locales]
BEGIN
    sentencias_ejecutables
[EXCEPTION
    manejadores_de_excepciones]
END [nombre];
```

Sintaxis de procedimientos

```
PROCEDURE nombre [( parametro [, parametro] ... )] IS
    [declaraciones_locales]
BEGIN
    sentencias_ejecutables
[EXCEPTION manejadores_de_excepciones]
END [nombre];
```

```
parametro := nombre_parametro [IN|OUT|IN OUT] tipo_dato [{:=|DEFAULT} expresion]
```

- No podemos imponer restricción NOT NULL a un parámetro.
- No podemos especificar una restricción del tipo.
- Una función debe acabar con la sentencia RETURN

En Oracle, cualquier identificador se debe declarar antes de usarse y eso también pasa con los subprogramas:

```
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 queremos definir subprogramas en orden alfabético o lógico, o necesitamos definir subprogramas mutuamente recursivos debemos 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
    ...

```

4.1.1. Almacenar subprogramas en la base de datos

Para almacenar subprogramas en la base de datos utilizamos la misma sintaxis para declararlo, anteponiendo CREATE [OR REPLACE] a PROCEDURE o FUNCTION.

y finalizamos el subprograma con una línea que contendrá el carácter / para indicar que termina ahí. Con REPLACE indicamos que si el subprograma ya existe, sea reemplazado.

```

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 se almacenan subprogramas en la bd, no se pueden utilizar declaraciones hacia delante. Cualquier subprograma almacenado debe conocer todos los subprogramas que utiliza. Invocamos un subprograma con la siguiente sintaxis:

```

nombre_procedimiento [(parametro [,parametro] ...)];
variable := nombre_funcion [(parametro [, parametro] ...)];
BEGIN
...
hijos := hijos_familia(10);
...
END;

```

Si el subprograma está almacenado en la bd y queremos invocarlo desde SQL Plus usamos la sintaxis:

```

EXECUTE nombre_procedimiento [(parametros)];
EXECUTE :variable_sql := nombre_funcion [(parametros)];

```

Los subprogramas de las bd son compilados antes, si hay algún error se informa de los mismos y se deben corregir creándolo de nuevo por medio de la cláusula OR REPLACE antes de que sea utilizado.

Hay varias vistas del diccionario de datos que nos ayudan a llevar control de subprogramas, tanto para ver su código como para los errores de compilación. También hay comandos que nos ayudan a hacer lo mismo de forma menos engorrosa. El comando show errors tras compilar muestra los errores que hay.

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.

4.1.2. Parámetros de los subprogramas

Las variables pasadas como parámetros a un subprograma son llamadas parámetros actuales. Las 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 nombrada. La asociación entre parámetros actuales y formales las hacemos por posición o por nombre.

En la notación posicional, el primer parámetro actual se asocia con el primer formal, y así con el resto. En la notación nombrada usamos el operador \Rightarrow para asociarlos. También podemos usar notación mixta.

Los parámetros pueden ser de entrada al subprograma, de salida, o de entrada y salida. Por defecto serán de entrada, si es de salida o de entrada y salida el parámetro actual debe ser una variable.

Un parámetro de entrada permite que le 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 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 recibirlas, por lo que un parámetro formal que actúe como parámetro actual siempre debe 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 medios de otros, debemos usar notación nombrada, o si los parámetros restantes también tienen valor por defecto, omitirlos todos.

4.1.3. Sobrecarga de subprogramas y recursividad

PL/SQL nos ofrece la posibilidad de sobrecargar funciones o procedimientos: llamar con el mismo nombre subprogramas que realizan el mismo cometido aceptando distinto número y/o tipo de parámetros. No se pueden sobrecargar subprogramas con el mismo número de parámetros aunque sus tipos sean diferentes pero de la misma familia o subtipos de la misma familia.

4.2. Paquetes

Un paquete es un objeto que agrupa tipos, elementos y subprogramas. Tienen dos partes. Especificación y cuerpo, aunque el cuerpo puede ser no necesario.

En la especificación se declara la interfaz del paquete con nuestra aplicación y en el cuerpo implementamos esa interfaz.

```
CREATE [OR REPLACE] PACKAGE nombre AS
    [declaraciones públicas y especificación subprogramas]
END [nombre]
CREATE [OR REPLACE] PACKAGE BODY nombre AS
    [declaraciones privadas y cuerpo subprogramas especificados]
[BEGIN
    sentencias inicialización]
END [nombre];
```

4.2.1. Ejemplos de utilización del paquete DBMS_OUTPUT

Oracle suministra un paquete público con el que podemos enviar mensajes desde subprogramas almacenados, paquetes y disparadores, colocarlos en un buffer y leerlos desde subprogramas almacenados, paquetes o disparadores.

SQL Plus permite visualizar los mensajes del buffer por medio de SET SERVEROUTPUT ON. Es fundamental su utilización para depurar nuestros subprogramas.

Los subprogramas que nos suministra este paquete son:

- 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 subprogramas y purga el buffer. Como con ENABLE, no es necesario si estamos usando la opción SERVEROUTPUT.

```
DISABLE
DISABLE();
```

Coloca elementos en el buffer, para convertirlos 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 devuelve 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 de tipo CHARARR, tipo definido en el paquete DBMS_OUTPUT como una tabla de VARCHAR2(255).

En el siguiente ejemplo creamos un procedimiento que visualice todos los agentes, su nombre, nombre de la familia y/o oficina a la que pertenece.

```
En el siguiente ejemplo creamos un procedimiento que visualice todos los agentes, su nombre, nombre de la familia y/o oficina a la que
```

Para ejecutarlo en SQL Plus se deben ejecutar las siguientes sentencias:

```
SQL>SET SERVEROUTPUT ON;  
SQL>EXEC lista_agentes;
```

4.3. Objetos

En PL/SQL las variables son los atributos y los subprogramas son métodos de los objetos. Podemos pensar en un tipo de objeto como una entidad que posee unos atributos y un comportamiento.

- Cuando creamos un tipo de objeto, creamos una entidad abstracta que especifica los atributos que tendrán los objetos de ese tipo y define su comportamiento.
- Cuando lo instanciamos, particularizamos la entidad abstracta en una particular, con los atributos del objeto con sus propios valores y comportamiento.

Los objetos tienen 2 partes. Especificación y cuerpo. La especificación declara primero los atributos y después los métodos. El cuerpo implementa la parte de especificación.

Todos los atributos son públicos (visibles). No podemos declarar atributos en el cuerpo, pero podemos declarar subprogramas locales que serán visibles en el cuerpo del objeto y que nos ayudan a implementar nuestros métodos.

Los atributos pueden ser de cualquier tipo 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 y tampoco asignarle la restricción NOT NULL.

Un método es un subprograma declarado en la parte de especificación de un tipo de objeto por medio de MEMBER. No puede llamarse igual que el tipo de objeto o que cualquier atributo, para cada método en la especificación, debe haber un método implementado en el cuerpo con la misma cabecera.

Todos los métodos en un tipo de objeto aceptan de primer parámetro una instancia de su tipo. Este parámetro es SELF y siempre es accesible a un método. Se declara explícitamente, por defecto será IN para las funciones e IN OUT para los procedimientos.

Los métodos se pueden sobrecargar, no podemos sobrecargarlos si los parámetros formales solo difieren en el modo o pertenecen a la misma familia. Tampoco se puede sobrecargar una función miembro si solo difiere del tipo devuelto.

Una vez creado el objeto, se puede utilizar en cualquier declaración. Un objeto declarado sigue las mismas normas que cualquier variable.

Un objeto declarado es NULL, dejará de serlo cuando se inicie por medio de su constructor o le asignemos otro. Intentar acceder a atributos de un objeto NULL lanzará una excepción ACCES_INTO_NULL.

Todos los objetos tienen constructores por defecto con el mismo nombre del 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, debemos hacerlo nosotros:

```
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.

Ejemplo:

```
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

Los tipos de objetos no tienen orden predefinido, no pueden ser comparados ni ordenados. Se puede definir el orden que seguirá un tipo de objeto por medio de funciones mapa y funciones orden.

Una función miembro mapa es una función sin parámetros que devuelve un tipo de dato DATE, NUMBER o VARCHAR2, siendo similar a una función hash. Se definen anteponiendo la palabra clave MAP. Solo puede haber una para cada 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 devuelve un número negativo si el objeto pasado es menor, cero si son iguales y uno 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;

```

5. Disparadores

PL/SQL ofrece para programar nuestra base de datos y mantener la integridad y seguridad, los disparadores o triggers.

Un disparador es un procedimiento que es ejecutado cuando se realiza alguna sentencia sobre la BD, bajo unas circunstancias a la hora de definirlo.

Pueden ser de tres tipos: de tablas, de sustitución o de sistema.

Puede ser lanzado antes (BEFORE) o después (AFTER) de realizar la operación que lo lanza.

Disparadores de tablas

Previenen transacciones erróneas y permiten implementar restricciones de integridad o seguridad, o automatizar procesos. Son los más utilizados.

Se ejecutan cuando se realiza alguna sentencia DML sobre una tabla, se puede lanzar al insertar, actualizar o borrar de una tabla. Podemos tener disparadores INSERT, UPDATE, DELETE o mezclados.

Puede ser lanzado una vez por sentencia (FOR STATEMENT) o una por cada fila a la que afecta (FOR EACH ROW).

De sustitución

No se ejecutan ni antes ni después, sino en lugar de (INSTEAD OF), solo se pueden asociar a vistas o a nivel de fila.

De Sistema

Se ejecuta cuando se produce una determinada operación sobre la BD, crear una tabla, conexión de usuario, etc. Se pueden detectar eventos como por ejemplo: LOGON, LOGOFF, CREATE, DROP, etc... y puede ser tanto BEFORE como AFTER.

Un disparador es simplemente un procedimiento que se puede utilizar para

- Llevar auditorías sobre la historia de los datos de nuestra BD.
- Garantizar complejas reglas de integridad.
- Automatizar la generación de valores derivados de consultas.
- 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.

5.1. Definición de disparadores de tablas

Para definir un disparador debemos indicar si se lanza antes o después de la ejecución de la sentencia que lo lanza. Si se lanza una vez por sentencia o una vez por fila y si será lanzado cuando se inserta, actualiza o borra.

Sintaxis:

```
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;
```

el **nombre** es el nombre del disparador, **momento** es BEFORE O AFTER acontecimiento es la acción: INSERT, UPDATE, DELETE. REFERENCING nos permite asignar alias a los valores NEW y/o OLD de las filas afectadas y WHEN nos permite indicar al disparador que solo sea lanzado cuando sea true cierta condición para cada fila (ambos solo pueden realizarse con disparadores para filas).

Un disparador de fila puede acceder a valores antiguos y nuevos de la fila afectada, referenciados como :old y :new, asignándole un alias a cada uno.

INSERT. Tiene sentido solo el valor nuevo.

UPDATE. El valor antiguo contiene la fila antes de actualizar y el valor nuevo contiene la fila a actualizar.

DELETE. Para un disparador lanzado al borrar solo tiene sentido el valor antiguo.

En el cuerpo de un disparador también podemos acceder a unos predicados que nos dicen el tipo que se está llevando a cabo: INSERTING, UPDATING, DELETING.

Un disparador de fila no puede acceder a la tabla asociada. Se dice que la tabla está mutando, su un disparador es lanzado en cascada por otro, 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 disparadores de una misma tabla el orden de ejecución es:

- 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 disparadores: 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

Ejemplo 1:

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 desde el DDL.

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 -- Si los dos valores son nulos
        RAISE_APPLICATION_ERROR(-20201, 'Un agente no puede ser huérfano');-- Lanza una excepción
    ELSIF (:new.familia IS NOT NULL and :new.oficina IS NOT NULL) THEN -- Si los dos tienen valores
        RAISE_APPLICATION_ERROR(-20202, 'Un agente no puede tener dos padres');
    END IF;
END;
/
```

Ejemplo 2:

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.

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.nombre, familia, oficina);
        COMMIT;
    ELSE
        UPDATE histagentes SET fecha_hasta = ahora WHERE identificador = :old.identificador and fecha_hasta IS NULL;
        COMMIT;
    END IF;
END;
/
```

Ejemplo 3

Queremos realizar un disparador que no nos permita llevar a cabo operaciones con familias si no estamos en la jornada laboral.

```

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

para acceder a bases de datos desde un lenguaje de programación externo se utilizan APIs disponibles para distintos lenguajes de programación. Podemos encontrar en estos enlaces más información:

<https://www.oracle.com/java/technologies/javase/javase-tech-database.html>

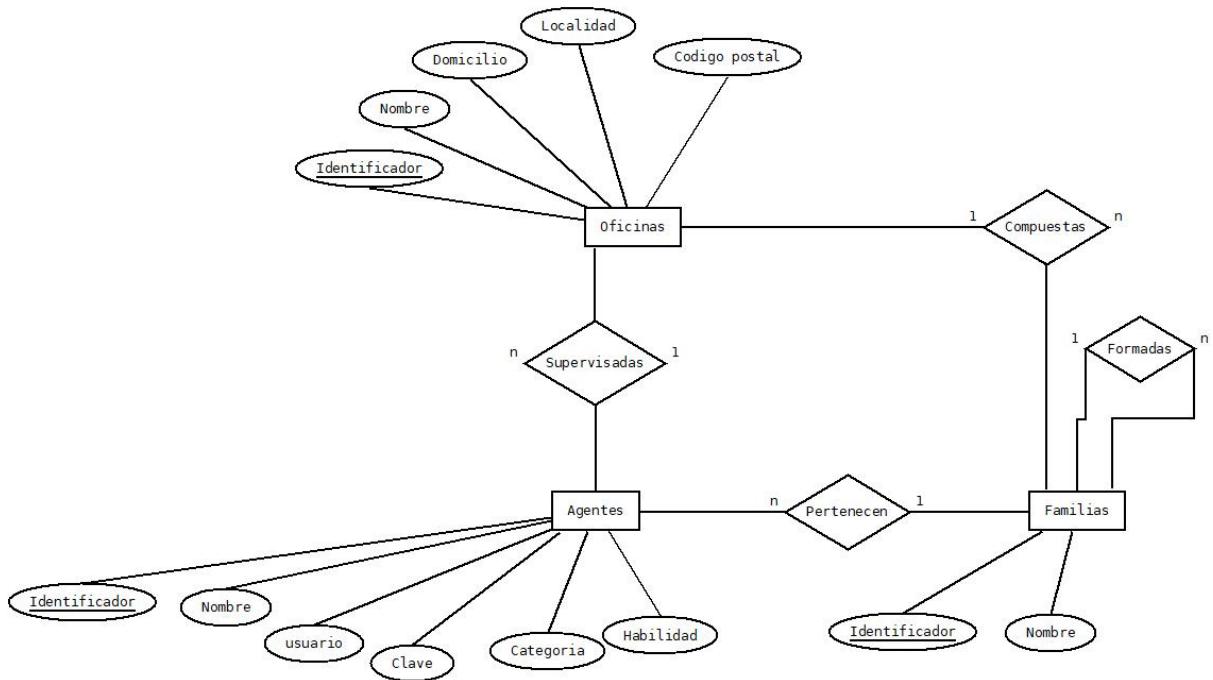
https://en.wikipedia.org/wiki/Open_Database_Connectivity

Anexo 1. 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.

Modelo entidad relación:



El **Modelo Relacional** resultante sería:

OFICINAS (identificador, nombre, domicilio, localidad, codigo_postal)

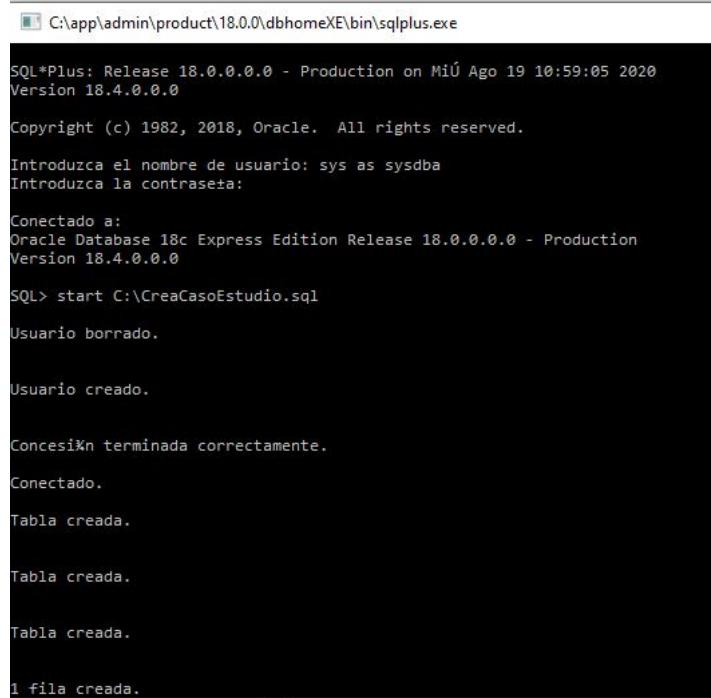
FAMILIAS (identificador, nombre, familia (fk), oficina (fk))

AGENTES (identificador, nombre, usuario, clave, habilidad, categoría, familia (fk), oficina (fk))

De este modelo de datos surgen tres tablas, que puedes crear en Oracle con el script del siguiente enlace:

El script crea un usuario llamado c##agencia con clave agencia. Para que puedas ejecutarlo y comenzar de cero, cuantas veces quieras, el script elimina el usuario c##agencia y sus tablas antes de volver a crearlo.

Conecta como administrador con SYS as SYSDBA y ejecútalo anteponiendo el símbolo @ o la palabra start antes del nombre del script.



C:\app\admin\product\18.0.0\dbhomeXE\bin\sqlplus.exe

```
SQL*Plus: Release 18.0.0.0 - Production on Mi  Ago 19 10:59:05 2020
Version 18.4.0.0

Copyright (c) 1982, 2018, Oracle. All rights reserved.

Introduzca el nombre de usuario: sys as sysdba
Introduzca la contraseña:

Conectado a:
Oracle Database 18c Express Edition Release 18.0.0.0 - Production
Version 18.4.0.0

SQL> start C:\CreaCasoEstudio.sql

Usuario borrado.

Usuario creado.

Concesi n terminada correctamente.

Conectado.

Tabla creada.

Tabla creada.

Tabla creada.

Tabla creada.

1 fila creada.
```

Habilitando la Salida/OUTPUT en Bloques PL/SQL.

PL/SQL no proporciona funcionalidad de entrada o salida directamente siendo necesario utilizar paquetes predefinidos de **Oracle** para tales fines. Para generar una salida debes realizar lo siguiente :

1. Ejecutar el siguiente comando tanto en **SQL*Plus** como en cualquier otro entorno:

```
SET SERVEROUTPUT ON
```

2. En el bloque **PL/SQL**, hay que utilizar el procedimiento PUT LINE del paquete DMBS_OUTPUT para mostrar la salida. El valor a mostrar en pantalla se pasará como argumento del procedimiento.

Ejecuta el siguiente código desde SQLPlus y desde SQLDeveloper para comprobar su funcionamiento.

```
SET SERVEROUTPUT ON
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hola mundo');
END;
```

Anexo II - 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 cursos.
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.

Anexo III - Evaluación de los atributos de un cursor explícito

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.

Anexo IV - Paso de parámetros a subprogramas

Notación mixta

```

DECLARE
    PROCEDURE prueba( formal1 NUMBER, formal2 VARCHAR2) IS
    BEGIN
        ...
    END;
    actual1 NUMBER;
    actual2 VARCHAR2;
BEGIN
    ...
    prueba(actual1, actual2);          --posicional
    prueba(formal1=>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; --illegal, 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
            ROLLBACK;
            RETURN -1;
        WHEN OTHERS THEN
            ROLLBACK;
            RETURN SQLCODE;
    END inserta_familia;
BEGIN
    ...
    resultado := inserta_familia( familia1 );
    ...
    resultado := inserta_familia( familia2, hijos_fam );
    ...
END;

```

Anexo V - Sobrecarga de subprogramas

```
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_familia.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).ha
            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;

    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;
```

Anexo VI. Ejemplo de recursividad

```
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;
```

Anexo VII. Ejemplo de paquete

```
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(reference_no_encontrada, -2291);  
    PRAGMA EXCEPTION_INIT(reference_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.habilidades);
    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_ofi
    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;

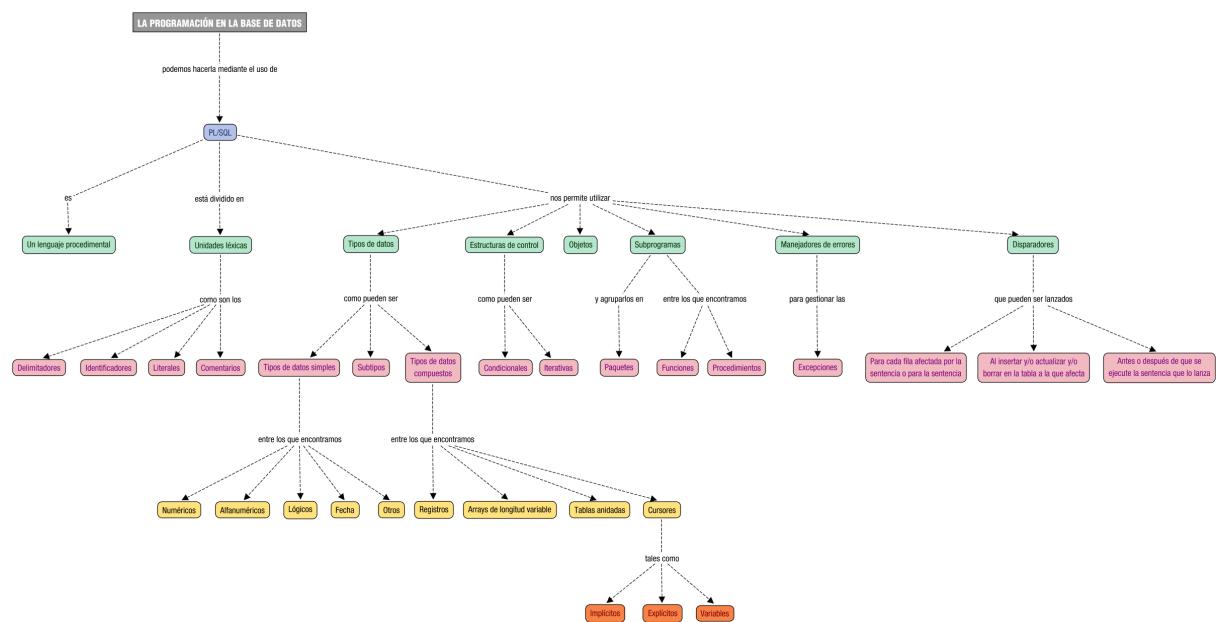
```

```

END IF;
DELETE agentes WHERE identificador = id_agente;
COMMIT;
RETURN todo_bien;
EXCEPTION
WHEN OTHERS THEN
ROLLBACK;
RETURN SQLCODE;
END borra_agente;
END call_center;
/

```

Mapa conceptual





7. Uso de bases de datos objeto - relacionales

Class	Bases de datos
Column	Xerach Casanova
Last Edited time	@Mar 15, 2021 1:06 PM

- 1. Características de las bases de datos objeto-relacionales
- 2. Tipos de dato objeto
- 3. Definición de tipos de objeto
 - 3.1. Declaración de atributos
 - 3.2. Definición de métodos
 - 3.3. Parámetro SELF
 - 3.4. Sobrecarga
 - 3.5. Métodos constructores
- 4. Utilización de objetos
 - 4.1. Declaración de objetos
 - 4.2. Inicialización de objetos
 - 4.3. Acceso a los atributos de objetos
 - 4.4. Llamada a los métodos de los objetos
 - 4.5. Herencia
- 5. Métodos MAP y ORDER
 - 5.1. Métodos ORDER
- 6. Tipos de datos colección
 - 6.2. Declaración y uso de colecciones
- 7. Tablas de objetos
 - 7.1. Tablas con columnas tipo objeto
 - 7.2. Uso de la sentencia SELECT
 - 7.3. Inserción de objetos
 - 7.4. Modificación de objetos
 - 7.5. Borrado de objetos
 - 7.6. Consultas con función VALUE
 - 7.7. Referencias a objetos
 - 7.8. Navegación a través de referencias.

Mapa conceptual

1. Características de las bases de datos objeto-relacionales

Son aquellas que han evolucionado desde el modelo relacional tradicional a un modelo híbrido que utiliza tecnología orientada a objetos. Clases, objetos y herencia son soportados en los esquemas de la base de datos y en el lenguaje de consulta y manipulación de datos. Además, da soporte a una extensión del modelo de datos con creación personalizada de tipos de datos y métodos.

Oracle implementa el modelo orientado a objetos como una extensión del modelo relacional.

El modelo objeto-relacional ofrece las ventajas de la reutilización de los objetos y a la vez mantiene la capacidad de concurrencia y rendimiento de bd relacionales.

Los tipos de objetos y las características orientadas a objetos permiten organizar los datos y acceder a ellos a alto nivel. Por debajo de la capa de objetos, los datos seguirán siendo almacenados en columnas y tablas, pero se trabaja con ellos de manera más parecida a entidades de la vida real, dando más significado a los datos.

El modelo orientado a objetos es similar a lenguajes como C++ o Java. La reutilización de objetos permite desarrollar aplicaciones de bases de datos más rápidamente y de manera más eficiente. Se permite además a los desarrolladores de aplicaciones acceder a las mismas estructuras de datos creadas en bases de datos.

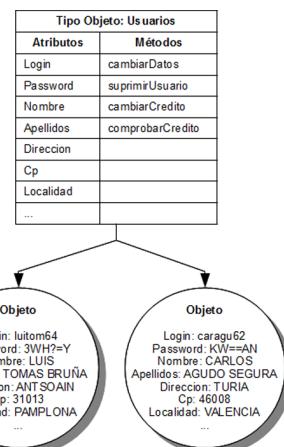
La programación orientada a objetos está enfocada a la construcción de componentes reutilizables. En PL/SQL la programación orientada a objetos está basada en tipos de objetos. Estos tipos permiten modelar objetos de la vida real, separar detalles de interfaces de usuarios e implementación y almacenar los datos de forma permanente en la base de datos. Son especialmente útiles cuando se realizan programas interconectados con Java u otros lenguajes POO.

Las tablas de bases de datos relacionales solo contienen datos, en cambio los objetos pueden incluir acciones sobre ellos. Un objeto compra puede incluir el método para calcular el importe de todos los elementos comprados. Un objeto cliente puede tener métodos para acceder al historial de compras.

2. Tipos de dato objeto

Un tipo de dato objeto es un tipo de dato compuesto definido por el usuario. Representa una estructura de datos, junto con funciones y procedimientos para manipularlos. Las colecciones permiten almacenar varios datos en una misma variable siendo todos del mismo tipo. Los tipos de datos objetos nos permiten almacenar datos de distinto tipo y asociar código a dichos datos.

Las variables que forman la estructura de datos de un tipo de dato objeto reciben el nombre de atributos y se corresponde con sus propiedades. Las funciones o procedimiento del tipo de dato objeto se denominan métodos y son sus acciones.



Al definir un tipo de dato objeto se crea una plantilla abstracta de un objeto real. La plantilla especifica atributos y comportamiento en el entorno de la aplicación. Dependiendo de la aplicación a desarrollar se utilizan unos determinados atributos y comportamientos y se descartan los que no son necesarios.

Los atributos son públicos y visibles desde otros programas cliente, pero los programas deben manipular los datos únicamente a través de los métodos (funciones o procedimientos) en vez de asignar u obtener sus valores directamente para mantener el estado apropiado de los atributos.

Durante la ejecución, una aplicación crea instancias de un tipo de objeto, con valores asignados en sus atributos.

3. Definición de tipos de objeto

La definición o declaración de un tipo de objeto está dividida en una especificación y un cuerpo. La especificación define la interfaz de programación, se declaran los atributos y operaciones o métodos para manipular los datos. El cuerpo se implementa en el código fuente de los métodos.

Toda la información que un programa necesita para usar los métodos se encuentra en la especificación, pudiendo modificar el cuerpo sin que afecte a los programas cliente.

En la especificación se declaran los atributos antes que los métodos. Si el objeto solo tiene atributos no es necesario declarar cuerpo. En el cuerpo no se declaran atributos. Todas las declaraciones realizadas en la especificación del tipo de objeto son públicas y por lo tanto visibles.

Un tipo objeto encapsula datos y operaciones. Se pueden declarar atributos y métodos en la especificación, pero no constantes (constants), excepciones (exceptions), cursores (cursors) o tipos (types). Se debe tener un atributo declarado como mínimo y un máximo de 1000. Los métodos son opcionales.

Para crear objetos en Oracle:

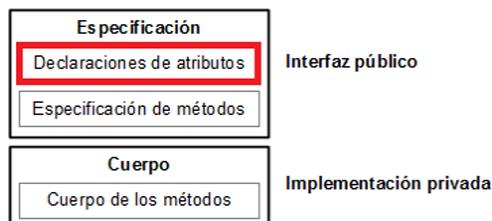
```
CREATE [OR REPLACE] TYPE nombre_tipo AS OBJECT (
  Declaración_atributos
  Declaración_métodos
);
```

nombre_tipo es el nombre del objeto y debe ser único. Si se quiere reemplazar se utiliza OR REPLACE.

Para eliminar objetos:

```
DROP TYPE nombre_tipo;
```

3.1. Declaración de atributos



Se puede realizar de forma similar a las variables. Se utiliza un nombre y un tipo de dato, el nombre debe ser único dentro del tipo de objeto y puede ser reutilizado en otros tipos de objeto. El tipo de dato en Oracle puede ser cualquiera excepto:

- Long y Longraw
- RowID y URowID
- Los 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.

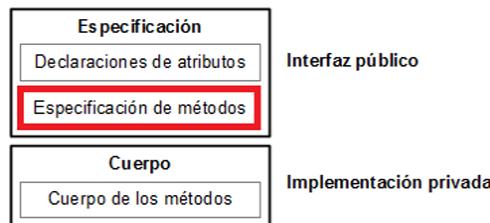
No se deben inicializar atributos usando el operador de asignación, ni cláusula default ni restricción not null. El tipo de dato puede ser otro tipo de objeto.

```
CREATE OR REPLACE TYPE Usuario AS OBJECT (
  login VARCHAR2(10),
  nombre VARCHAR2(30),
  f_ingreso DATE,
  credito NUMBER
);/
/
```

Para crear o modificar atributos después de haber sido creado el tipo de objeto se utiliza ALTER TYPE y ADD, MODIFY o DROP ATTRIBUTE

```
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);
```

3.2. Definición de métodos



Un método es un subprograma que declaras en la especificación de un tipo de objeto usando las palabras MEMBER o STATIC. El nombre del método no puede ser del mismo que el tipo de objeto ni de algún atributo.

Los métodos tienen dos partes: especificación y cuerpo.

La especificación asigna el nombre al método con sus parámetros opcionales, en el caso de funciones, un tipo de dato de retorno.

```
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 se indica el código que debe ejecutar para realizar una determinada tarea cuando es invocado.

```
CREATE OR REPLACE TYPE BODY Usuario AS
  MEMBER PROCEDURE incrementoCredito(inc NUMBER) IS
    BEGIN
      credito := credito + inc;
    END incrementoCredito;
  END;
/
```

Por cada especificación que se indique debe existir su correspondiente cuerpo del método, o bien, declararse como NOT INSTANTIABLE, para indicar que el cuerpo del método se encontrará en un subtipo de ese tipo de objeto. El nombre de las cabeceras deben coincidir en especificación y cuerpo.

Los parámetros formales se declaran con nombre y tipo de dato pero sin restricciones de tamaño, el tipo de dato puede ser de los mismos tipos que para los atributos y se aplican las mismas restricciones para los tipos de retorno en las funciones.

El código fuente no solo puede escribirse en PL/SQL, sino también en otros lenguajes de programación.

También se puede usar ALTAR TYPE para añadir, modificar o eliminar métodos de manera similar.

3.3. Parámetro SELF

El parámetro SELF se utiliza en los métodos MEMBER, hace referencia a una instancia del mismo tipo de objeto. Es como el this de java. En funciones member si no se declara SELF, su modo por defecto es IN, en procedimientos MEMBER es IN OUT. Los métodos static no pueden utilizar SELF y tampoco se puede especificar el modo OUT.

```
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;
```

3.4. Sobrecarga

Los métodos se pueden sobrecargar (utilizar el mismo nombre siempre que sus parámetros sean distintos en cantidad o tipo de dato).

Al hacer una llamada a método se comparan sus parámetros actuales con los formales de los métodos declarados y se ejecuta aquel que coincida.

no es válida una sobrecarga de dos métodos con parámetros formales que se diferencian solamente en su modo y tampoco funciones que se diferencien solo en el valor de retorno.

```
MEMBER PROCEDURE setNombre(Nombre VARCHAR2)
MEMBER PROCEDURE setNombre(Nombre VARCHAR2, Apellidos VARCHAR2)
```

3.5. Métodos constructores

Todos los objetos tienen un método constructor, que es una función con el mismo nombre que el tipo de objeto e inicializa los atributos, retornando una nueva instancia de ese objeto.

Oracle crea un método constructor por defecto para cada objeto declarado. Sus parámetros formales coinciden en orden, nombres y tipos con los atributos del objeto.

También se pueden declarar constructores propios, reescribiendo el declarado por el sistema o definiendo otro con otros parámetros.

La ventaja de crear un constructor personalizado es el poder verificar que los datos que se van a asignar son correctos.

Si se desea reemplazar el constructor por defecto simplemente se debe utilizar la sentencia CONSTRUCTOR FUNCTION seguida del nombre del tipo de objeto en el que se encuentra. A continuación se indican los parámetros de la manera habitual y por último se indica el valor de retorno, que en este caso es el propio objeto: RETURN SELF AS RESULT.

Se pueden crear varios métodos constructores siguiendo las normas de sobrecarga de métodos.

```
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;

```

4. Utilización de objetos

4.1. Declaración de objetos

Una vez definido el objeto se puede utilizar para declarar variables de objetos 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.

Por ejemplo:

```
u1 Usuario;
```

En la declaración de cualquier procedimiento o función, se puede utilizar el tipo de dato objeto definido para pasarlo como parámetro.

```
PROCEDURE setUsuario(u IN Usuario)
```

La llamada a ese método se realiza utilizando el objeto como parámetro:

```
setUsuario(u1);
```

Para que una función retorne objetos:

```
FUNCTION getUsuario(codigo INTEGER) RETURN Usuario
```

Los objetos se crean durante ejecución 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 en PL/SQL. Es decir, se crean o instancian en dicho bloque y se destruyen cuando se sale de ellos. En un paquete, los objetos se instancian cuando se hace referencia al paquete y dejan de existir cuando se finaliza sesión en la bd.

4.2. Inicialización de objetos

Para crear o instanciar objetos de un determinado tipo de objeto se hace llamada a su método constructor con la instrucción NEW seguido del nombre del tipo de objeto como llamada a una función, indicando sus parámetros iniciales.

El orden de parámetros debe coincidir con el orden en que están declarados los atributos.

```
variable_objeto := NEW Nombre_Tipo_Objeto (valor_atributo1, valor_atributo2, ...);
```

Por ejemplo:

```
u1 := NEW Usuario('luitom64', 'LUIS ', 'TOMAS BRUÑA', '24/10/07', 100);
```

hasta que no se inicializa un objeto llamando al constructor, el objeto tiene valor null. Es habitual declarar e inicializar a la vez:

```
u1 Usuario := NEW Usuario('luitom64', 'LUIS ', 'TOMAS BRUÑA', '24/10/07', 100);
```

La llamada al constructor se puede realizar en cualquier lugar en el que se puede hacer una llamada a función de manera habitual. Puede ser, por ejemplo 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 se está creando. Si la llamada es al constructor por defecto de Oracle, se debe indicar un parámetro para cada atributo en el orden declarado, teniendo en cuenta que los atributos no pueden tener valores por defecto asignados en la declaración.

Existe posibilidad de utilizar nombres de los parámetros formales en la llamada al constructor, en lugar de utilizar el modelo posicional de los parámetros y así no hace falta respetar el orden en que se encuentran los parámetros reales respecto a los formales.

```
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;
/
```

4.3. Acceso a los atributos de objetos

Para hacer referencia a un atributo de objeto se debe utilizar el nombre del atributo utilizando un punto para acceder al valor que contiene o para modificarlo, precedido del nombre del objeto.

```
nombre_objeto.nombre_atributo
```

por ejemplo:

```
unNombre := usuario1.nombre;
dbms_output.put_line(usuario1.nombre);
```

La modificación del valor puede ser:

```
usuario1.nombre:= 'Nuevo Nombre';
```

Los nombres de los atributos se pueden encadenar, de manera que se permite acceso a atributos de tipos de objetos anidados:

```
sitio1.usuario1.nombre
```

Si se utiliza una expresión con acceso a un atributo de objeto no inicializado, se evalúa NULL, si se intenta asignar valores a un objeto no inicializado se lanza una excepción ACCESS_INTO_NULL. Se puede comprobar si un objeto es null con IS NULL.

Al intentar una llamada a un método de objeto no 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 se lanza una excepción al intentar modificar el valor de los atributos.

4.4. Llamada a los métodos de los objetos

Se puede invocar a los métodos utilizando también un punto entre el objeto y el método, los parámetros reales que separan por comas, entre paréntesis.

```
usuario1.setNombreCompleto('Juan', 'García Fernández');
```

Si el método no tiene parámetros se dejan los paréntesis vacíos o se omiten los paréntesis.

```
credito := usuario1.getCredito();
```

La llamada a métodos puede encadenarse, el orden de ejecución de los métodos se hace de derecha a izquierda y se debe tener en cuenta que el método de la izquierda debe retornar un objeto del tipo correspondiente al de la derecha.

```
sitio1.getUsuario.setNombreCompleto('Juan', 'García Fernández');
```

Los métodos MEMBER son invocados utilizando una instancia del tipo objeto:

```
nombre_objeto.metodo()
```

Los métodos estáticos se invocan usando el tipo de objeto:

```
nombre_tipo_objeto.metodo()
```

4.5. Herencia

PL/SQL admite herencia simple, con lo cual los subtipos o tipos heredados contienen los atributos y métodos del tipo padre, además de los suyos adicionales o incluso sobreescribir los métodos del tipo padre.

Para indicar que es un tipo de objeto heredado se utiliza la palabra reservada UNDER y además tener en cuenta que el tipo de objeto del que hereda debe tener la propiedad NOT FINAL, ya que por defecto los tipos de objeto son FINAL y no se puede heredar de ellos.

Si no se indica lo contrario, se pueden crear objetos de los tipos de objeto declarado, pero usando la opción NOT INSTANTIABLE se puede declarar tipos de objetos que no se pueden instanciar y deben 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;
/

```

5. Métodos MAP y ORDER

Las instancias de un tipo de objeto no tienen orden predefinido, para establecer uno, con el fin de hacer ordenación o comparación se crea un método map.

Si se hace una comparación entre dos objetos y deseas saber que uno es mayor que otro hay que establecer con un método MAP cual va a ser el valor que se va a utilizar para comparar (nombre, apellidos...).

Para crear un método MAP se declara un método que retorne el valor que se va a utilizar para hacer la comparación. Este método debe empezar 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 utilizaremos para realizar comparaciones entre instancias:

```

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 métodos MAP para evaluar expresiones lógicas que resultan valores booleanos como `objeto1 > objeto2` y para hacer las comparaciones implícitas de las cláusulas DISTINCT, GROUP BY y ORDER BY

Cada tipo de objeto solo puede tener un método MAP declarado y solo puede retornar: DATE, NUMBER, VARCHAR2, CHARACTER O REAL.

Ejemplo:

Partimos de un tipo direccion_t definido para guardar datos de una dirección:

```
CREATE or replace TYPE direccion_t AS OBJECT (
    calle VARCHAR2(200),
    ciudad VARCHAR2(200),
    prov CHAR(2),
    codpos VARCHAR2(20)
);
```

Sea cliente_t un tipo definido para guardar información de un cliente, que contiene dos métodos, uno para calcular la edad de un cliente y otro sería el que contiene el MAP:

```
create or replace TYPE cliente_t AS OBJECT (
    clinum NUMBER,
    clinomb VARCHAR2(200),
    direccion direccion_t,
    telefono VARCHAR2(20),
    fecha_nac DATE,
    MEMBER FUNCTION edad RETURN NUMBER,
    MAP MEMBER FUNCTION ret_value RETURN NUMBER
);
```

Escribe el código correspondiente a los métodos edad y ret_value que ordenará por el campo clinum. Es decir cuando se comparan dos objetos del tipo cliente_t se compararán por su atributo clinum.

```
create or replace TYPE BODY cliente_t AS
    MEMBER FUNCTION edad RETURN NUMBER IS
        a integer;
        d DATE;
    BEGIN
        select sysdate into d from dual;
        a:=months_between(sysdate,fecha_nac); -- Obtiene el número de meses que hay entre las dos fechas
        DBMS_OUTPUT.PUT_LINE(a);
        a:=trunc(a/12); -- Trunca el resultado de la división quitando la parte decimal
        RETURN a;
    END;
    MAP MEMBER FUNCTION ret_value RETURN NUMBER
    IS
    BEGIN
        RETURN clinum;
    END;
END;
```

5.1. Métodos ORDER

De forma similar a MAP, se puede declarar en cualquier tipo de objeto un método ORDER que establece un orden entre dos objetos instanciados de dicho tipo.

Cada tipo de objeto solo puede contener un método ORDER y debe devolver un valor numérico que permite establecer el orden entre los objetos. Si deseas que un objeto sea menor que otro, se puede retornar el -1, si van a ser iguales el 0 y si es mayor el 1.

Para declarar que método va a realizar esta operación debes indicar la palabra ORDER delante de la declaración y debe devolver un INTEGER. Necesita además un parámetro del mismo tipo de objeto, el cual será el que se compare con el objeto que utilice el método.

```
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;

```

El método devolverá un -1, 0 o 1, dependiendo de si el objeto que utiliza el método (SELF) es menor, igual o mayor que el objeto pasado por parámetro.

Se puede declarar un método map o un método order, pero no los dos.

Con un número alto de objetos es preferible usar MAP ya que ORDER es menos eficiente.

Ejemplo:

Siguiendo el ejemplo del apartado anterior, tipo cliente_t quedaría:

```

CREATE or replace TYPE cliente_t AS OBJECT (
    clinum NUMBER,
    clinomb VARCHAR2(200),
    direccion direccion_t,
    telefono VARCHAR2(20),
    fecha_nac DATE,
    ORDER MEMBER FUNCTION cli_ordenados (x cliente_t) RETURN INTEGER,
    MEMBER FUNCTION edad RETURN NUMBER);

```

Implementa el método cli_ordenados.

```

ORDER MEMBER FUNCTION cli_ordenados (x IN cliente_t)
RETURN INTEGER IS
BEGIN
    RETURN clinum - x.clinum; /*la resta de los dos números clinum*/
END;
END;

```

6. Tipos de datos colección

Son como vectores o matrices, pero en este caso solo pueden ser de una dimensión y los elementos se indexan mediante un valor de tipo numérico o cadenas de caracteres.

Oracle proporciona VARRAY y NESTED TABLE (tabla anidada) como tipos de datos colección.

- **VARRAY** es una colección de elementos a la que se le establece un máximo número de elementos al declararse. La longitud es fija y la eliminación de elementos no ahorra espacio en memoria.
- **NESTED TABLE** almacena cualquier número de elementos y tiene un tamaño dinámico, no teniendo por qué existir valores para todas las posiciones de la colección.
- Una variación son los arrays asociativos, que utilizan valores arbitrarios para sus índices y no son necesariamente consecutivos.

Para almacenar un número fijo de elementos o hacer un recorrido entre elementos de forma ordenada, o necesitas obtener y manipular toda la colección como un valor se usa VARRAY

Si se necesita ejecutar consultas sobre una colección de manera eficiente o manipular elementos de manera arbitraria, hacer operaciones de inserción, actualización o borrado masivo se usa NESTED TABLE.

Se pueden declarar como instrucción SQL o en el bloque de declaraciones de un programa PL/SQL. El tipo de dato en PL/SQL puede ser cualquiera excepto REF CURSOR. En SQL no pueden ser: BINARY_INTEGER, PLS_INTEGER, BOOLEAN, LONG, LONG RAW, NATURAL, NATURALN, POSITIVE, POSITIVEN, REF CURSOR, SIGNTYPE Y STRING.

Una tabla de una base de datos puede contener columnas que sean colecciones. Sobre una tabla que contiene colecciones se pueden realizar operaciones de consulta y manipulación de datos de la misma

manera que con tablas habituales.

6.2. Declaración y uso de colecciones

```
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_indice;
```

nompre_tipo es el nombre de la colección, tamaño_max es el número máximo en caso de VARRAY y tipo_elemento es el tipo de elementos que forman la colección, que también pueden ser objetos.

En caso de que la declaración sea en SQL se declara así:

```
CREATE [OR REPLACE] TYPE.
CREATE TYPE nombre_tipo IS ...
```

tipo_índice representa el tipo de dato para el índice. Puede ser PLS_INTEGER, BINARY_INTEGER o VARCHAR2. Este último lleva entre paréntesis el tamaño.

Hasta que no sea iniciada, la colección es NULL, para inicializar una colección se usa el constructor y se le pasan como parámetros los valores iniciales de la colección:

```
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 directamente en el bloque de declaraciones:

```
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 se debe indicar el nombre de la colección y entre paréntesis el índice que ocupa. Tanto en VARRAY como en NESTED TABLE el primer elemento es 1.

```
dbms_output.put_line(misColores(2));
```

```
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;/
```

Al definir una tabla que contiene un atributo que es de tipo tabla, es necesario darle un nombre de almacenamiento a NESTED TABLE mediante la cláusula STORE AS que es un nombre interno y no se puede utilizar para acceder directamente a la tabla anidada.

7. Tablas de objetos

También se pueden almacenar objetos en tablas de igual manera que los tipos de datos habituales.

Los tipos de dato objeto se pueden usar para formar una tabla exclusivamente formado por elementos de ese tipo, o también para usarse como un tipo de columna más entre otras columnas de otros tipos de datos.

Para crear una tabla formada por un determinado tipo de dato objeto o tabla de objetos:

```
CREATE TABLE NombreTabla OF TipoObjeto;
```

Siendo nombreTabla el nombre de la tabla que almacenará los objetos de tipoObjeto:

```
CREATE TABLE UsuariosObj OF Usuario;
```

Si una tabla hace uso de un tipo objeto, no se puede modificar la estructura ni eliminar dicho tipo de objeto, Desde que un tipo objeto es utilizado en una tabla no se puede volver a definir.

Al crear una tabla de esta manera podemos almacenar objetos de tipo usuario en una tabla, quedando sus datos persistentes mientras no sean eliminados de la tabla. Hasta ahora, al guardar los objetos en variables, al terminar la ejecución, la información que contienen desaparece.

Cuando se instancia un objeto con el fin de almacenarlo en una tabla, este objeto no tiene identidad fuera de la tabla de la bd, pero el tipo de objeto existe independiente de cualquier tabla, para crear objetos en cualquier modo.

Las tablas que contienen solo filas con objetos se llaman tablas de objetos. Los atributos se muestran como columnas de la tabla.

SQL> select * from usuariosobj;					
LOGIN	NOMBRE	APELLIDOS	F_INGRES	CREDITO	
luitom64	LUIS	TOMAS BRUNA	24/10/07	50	
caragu72	CARLOS	AGUDO SEGURA	06/07/07	100	

7.1. Tablas con columnas tipo objeto

Se puede usar cualquier tipo de objeto declarado previamente para usarlo como tipo de dato de una columna de una tabla de la bd. Una vez creada la tabla se puede utilizar cualquier sentencia SQL para insertar un objeto, seleccionar sus atributos y actualizar sus datos.

Para crear una tabla en la que alguna de sus columnas sea un tipo de objeto, simplemente se especifica que el tipo de dato de esa columna es del tipo de objeto.

```
CREATE TABLE Gente (
    dni VARCHAR2(10),
    unUsuario Usuario,
    partidasJugadas SMALLINT
);
```

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.

```
SQL> select * from gente;
DNI
UNUSUARIO<LOGIN, NOMBRE, APELLIDOS, F_INGRESO, CREDITO>
PARTIDASJUGADAS
22900970P
USUARIO<'luitom64', 'LUIS', 'TOMAS BRUNA', '24/10/07', 50>
54
62603088D
USUARIO<'caragu72', 'CARLOS', 'AGUDO SEGURA', '06/07/07', 100>
21
```

7.2. Uso de la sentencia SELECT

Se puede utilizar la sentencia SELECT para obtener datos de filas almacenadas en tablas de objetos o tablas con columnas de tipos de objetos.

```
SELECT * FROM NombreTabla;
```

```
SQL> select * from usuariosobj;
LOGIN      NOMBRE      APELLIDOS          F_INGRES   CREDITO
luitom64   LUIS        TOMAS BRUNA       24/10/07   50
caragu72   CARLOS      AGUDO SEGURA     06/07/07   100
SQL> select * from gente;
DNI
UNUSUARIO<LOGIN, NOMBRE, APELLIDOS, F_INGRESO, CREDITO>
PARTIDASJUGADAS
22900970P
USUARIO<'luitom64', 'LUIS', 'TOMAS BRUNA', '24/10/07', 50>
54
62603088D
USUARIO<'caragu72', 'CARLOS', 'AGUDO SEGURA', '06/07/07', 100>
21
```

La tabla que forma parte de la consulta puede ser una tabla de objetos o una tablas que contiene columnas de tipos objetos.

En las sentencias SELECT con objetos se pueden incluir cláusulas y funciones de agrupamiento de las sentencias SELECT: SUM, MAX, WHERE, ORDER, JOIN...

Se usan alias para hacer referencias al nombre de la tabla. por ejemplo, en la siguiente consulta se obtiene el nombre y los apellidos de los usuarios con 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;
```

7.3. Inserción de objetos

Se usa la misma que para introducir datos de manera habitual. Usando insert de sql.

Para hacer un INSERT de un determinado tipo de objetos o si posee un campo de un determinado tipo de objeto, se suministra a la sentencia INSERT un objeto instanciado de su tipo de objeto correspondiente.

```
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;
```

También se puede crear el objeto dentro de la misma sentencia insert, sin necesidad de guardarla en una variable.

```
INSERT INTO UsuariosObj VALUES (Usuario('luitom64', 'LUIS', 'TOMAS BRUNA', '24/10/2007', 50));
```

y luego comprobar los resultados haciendo un select habitual

```
SELECT * FROM UsuariosObj;
```

De la misma manera, podemos realizar una inserción de filas en tablas con columnas de tipo objeto.

```
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

La diferencia entre los UPDATE habituales es la forma de especificar los nombres de los campos afectados, según sea una tabla de objetos o un atabla con alguna columna de tipo objeto.

Para tabla de objetos se hace referencia a los atributos de los objetos justo después del nombre asignado a la tabla.

```
UPDATE NombreTabla
SET NombreTabla.atributoModificado = nuevoValor
WHERE NombreTabla.atributoBusqueda = valorBusqueda;
```

```
UPDATE UsuariosObj
SET UsuariosObj.credito = 0
WHERE UsuariosObj.login = 'luitom64';
```

Es habitual abreviar con alias:

```
UPDATE UsuariosObj u
SET u.credito = 0
WHERE u.login = 'luitom64';
```

También se puede cambiar un objeto por otro:

```
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 columna de tipo objeto se hace referencia al nombre de la columna que contiene los objetos.

```
UPDATE NombreTabla
SET NombreTabla.colObjeto.atributoModificado = nuevoValor
WHERE NombreTabla.colObjeto.atributoBusqueda = valorBusqueda;
```

```
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

```
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

El uso del DELETE sobre objetos almacenados en tablas es:

```
DELETE FROM NombreTablaObjetos WHERE condición;
```

Recordando siempre que si no se utiliza WHERE, se eliminan todos los objetos de la tabla.

```
DELETE FROM UsuariosObj u WHERE u.credito = 0;
```

De manera similar se usa el borrado de filas en tablas en la que alguna de sus columnas son objetos.

```
DELETE FROM Gente g WHERE g.unUsuario.credito = 0;
```

También se pueden 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.

7.6. Consultas con función VALUE

Cuando se tiene la necesidad de hacer referencia a un objeto en lugar de alguno de sus atributos, se utiliza la función VALUE junto al nombre de la tabla de objetos o sus alias, dentro de un SELECT.

```
INSERT INTO Favoritos SELECT VALUE(u) FROM UsuariosObj u WHERE u.credito >= 100;
```

Esa misma función se puede usar para hacer comparaciones de igualdad entre objetos.

```
SELECT u.login FROM UsuariosObj u JOIN Favoritos f ON VALUE(u)=VALUE(f);
```

Si llevamos la comparación a una columna de tipo objetos, en el caso de la referencia a la columna, se obtiene directamente el objeto sin VALUE:

```
SELECT g.dni FROM Gente g JOIN Favoritos f ON g.unUsuario=VALUE(f);
```

Con la cláusula INTO se puede guardar en variables el objeto obtenido en las consultas usando la función VALUE.

Una vez asignado el objeto a la variable se puede hacer uso de ella de la manera habitual en manipulación de objetos:

```
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;
```

7.7. Referencias a objetos

El paso de objetos a un método es ineficiente si el objeto es grande, lo mejor es 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 pasarlo por completo. Ese puntero se llama REF.

Al compartir un objeto mediante referencia, los datos no se duplican y si se hacen cambios en un atributo se producen en un único lugar.

Cada objeto almacenado en una tabla tiene un identificador de objetos y sirve como referencia a dicho objeto.

Las referencias se crean con el modificador REF delante del tipo objeto y se pueden 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),
    usuarioCreator 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;
/
```

Las referencias a tipos de objetos solo se pueden usar si han sido declarados previamente. En el ejemplo anterior, no se puede declarar el tipo partida antes que el tipo usuario.

Si surgen dos tipos que utilizan referencias mutuas, se soluciona haciendo una declaración de tipo anticipada. Se realiza indicando únicamente el nombre del tipo 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.

No se puede acceder directamente a los atributos de un objeto referenciado dentro de una tabla, para ellos se usa la función DEREF.

Esta función toma una referencia a un objeto y retorna el valor de ese objeto.

```
u_ref REF Usuario;
u1 Usuario;
```

Si u_ref hacer referencia a un objeto de tipo usuario de la tabla UsuariosObj, para obtener info de sus atributos usamos la función DEREF. Se usa como parte de una consulta SELECT, utilizando una tabla tras la cláusula FROM.

La BD de oracle ofrece una tabla DUAL para estas operaciones, es creada automáticamente por la bd y es accesible por todos los usuarios, solo tiene un campo y un registro, es una tabla comodín.

```
SELECT DEREF(u_ref) INTO u1 FROM Dual;
dbms_output.put_line(u1.nombre);
```

Para obtener el objeto referenciado por una variable REF, se utiliza una consulta sobre cualquier tabla, independientemente de la tabla en la que se encuentre el objeto referenciado, solo existe la condición de que se obtenga una sola fila como resultado. El resultado será un objeto almacenado en la tabla UsuariosObj

```
DECLARE
US USUARIO;
U_REF REF USUARIO;
BEGIN
SELECT REF(U) INTO U_REF FROM UsuObj u WHERE u.login = 'luitom64';
SELECT DEREF(u_ref) INTO us FROM dual;
dbms_output.put_line(us.nombre);
END;
```

Mapa conceptual

