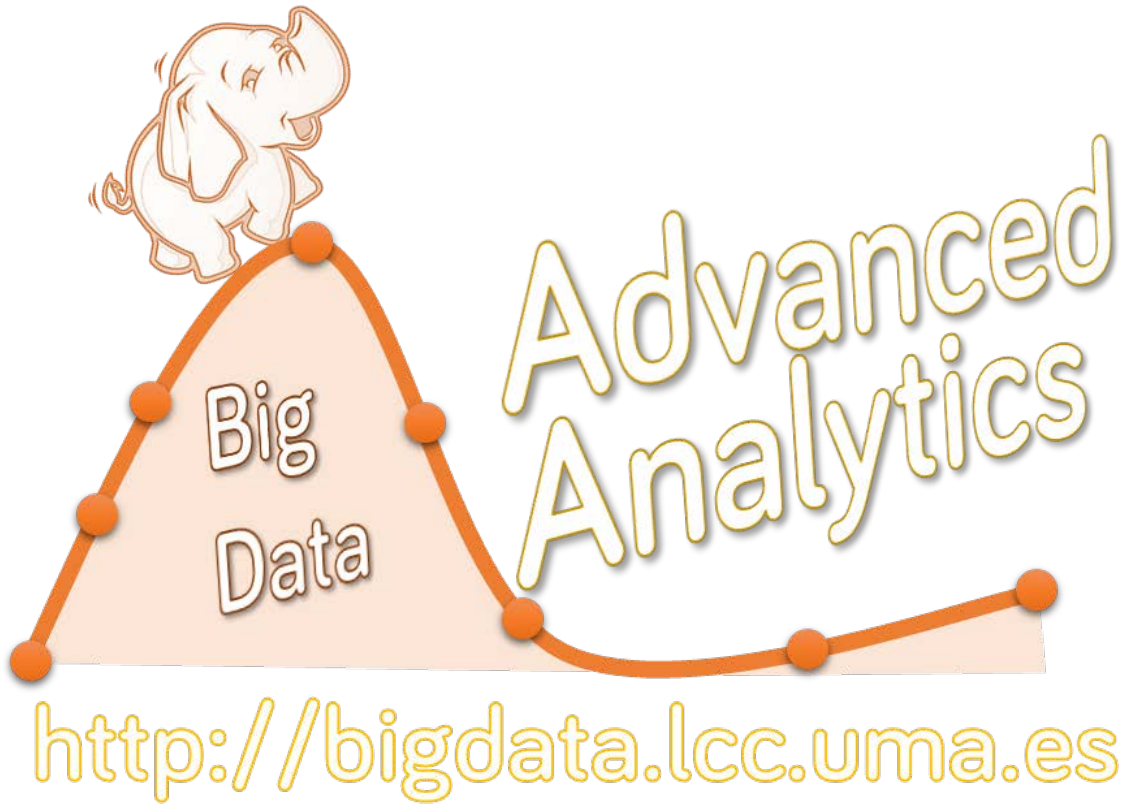
MÓDULO DE BASES DE DATOS



MÓDULO DE BASES DE DATOS

Índice de contenido

1 MOTIVACIÓN. UN PROBLEMA SENCILLO. 3

2 LA APROXIMACIÓN AL PROBLEMA 4

3 EL MODELADO CONCEPTUAL CON EL MODELO DE DATOS ENTIDAD-RELACIÓN 9

Un problema de diseño: límite entre atributo y entidad. 14
De vuelta a nuestro problema 16

4 EL MODELO DE DATOS RELACIONAL 20

5 ALGORITMO DE TRADUCCIÓN DE ENTIDAD RELACIÓN EXTENDIDO A MODELO RELACIONAL 24

De vuelta con nuestro ejemplo. 29

6 SQL: EL ESTÁNDAR DE LAS BASES DE DATOS RELACIONALES 36

6.1 DEFINICIÓN DE DATOS Y RESTRICCIONES EN SQL 36
Creación y gestión de tablas 37
Restricciones 38
6.2 SENTENCIAS INSERT, DELETE, Y UPDATE EN SQL 39
6.3 CONSULTAS BÁSICAS EN SQL 41
Filtrado y Ordenación 41
Ejemplo de utilización de la cláusula WHERE 42
Ejemplo de uso de operadores aritméticos en la cláusula WHERE 42
Ejemplo de uso de operadores de comparación complejos 42
Ejemplo de uso de operadores lógicos 43
Ejemplo de uso de ORDER BY 43
6.4 CONSULTAS SQL MÁS COMPLEJAS 44
Tipos de Composiciones 44
6.5 SUBCONSULTAS 46
Tipos de Subconsultas 46
Subconsultas correlacionadas 47
6.6 FUNCIONES DE AGREGACIÓN EN SQL 48
6.7 VISTAS (TABLAS VIRTUALES) EN SQL 49

7 SQL INMERSO. PROGRAMACIÓN EN PL/SQL 51

7.1 ¿QUÉ ES PL/SQL? 51
7.2 ¿PARA QUÉ SIRVE PL/SQL? 51
7.3 CREACIÓN DE PROGRAMAS PL/SQL 52
7.3.1 Estructura Básica: el bloque. 52
7.3.2 Anidación de bloques 53
7.3.3 Estructura de un bloque 53
Sección de declaraciones 54
Variables 55
Constantes 56
Variables especiales (%TYPE y %ROWTYPE) 56
Tablas y Registros 57

Registros 58
Tablas 58
Subprogramas locales 61
Objetos 61
Sección de ejecución 61
Instrucción de asignación 61
Estructura de condición. 63
Estructura de repetición. 64
Sección de excepciones 65
7.3.4 Comentarios 66
7.3.5 Funciones y procedimientos. 66
7.3.6 Uso de etiquetas 69
7.3.7 Acceso a datos 69
7.3.8 Manipulación de datos (sentencias DML) 69
7.3.9 Ejecución de sentencias DDL 70
7.3.10 Depuración de programas, entrada y salida 70
7.3.11 Como crearlos y ejecutarlos 71
7.4 EJEMPLOS 71
7.5 ¿QUÉ ES UN CURSOR? 75
7.6 ¿PARA QUÉ SIRVE UN CURSOR? 76
7.7 CREACIÓN Y UTILIZACIÓN DE CURSORES 77
7.7.1 Cursores actualizables 80
7.8 EJEMPLOS 81
7.9 ¿QUÉ ES UNA EXCEPCIÓN? 84
7.10 ¿PARA QUÉ SIRVE UNA EXCEPCIÓN? 85
7.11 PROGRAMACIÓN DEL MANEJO DE EXCEPCIONES 85
7.12 EJEMPLOS 88
7.13 ¿QUÉ ES UN PAQUETE? 90
7.14 ¿PARA QUE SIRVEN LOS PAQUETES? 90
7.15 CREACIÓN Y UTILIZACIÓN DE PAQUETES. 90
7.16 EJEMPLOS 93
7.17 ¿QUÉ ES UN OBJETO? 94
7.18 CREACIÓN DE OBJETOS 94
7.18.1 Métodos asociados a tipos de objetos 94
7.19 ¿QUÉ SON LOS DISPARADORES? 98
7.20 ¿PARA QUE SIRVEN LOS DISPARADORES? 98
7.21 CREACIÓN DE DISPARADORES. 99
7.22 EJEMPLOS 103
7.23 ENTORNOS DE DESARROLLO DE PL/SQL 108
8 FINALIZANDO EL EJEMPLO. 110
9 BIBLIOGRAFÍA 115
APÉNDICE A: SOBRE LOS SISTEMAS DE BASES DE DATOS. UN POCO DE HISTORIA. 117
APÉNDICE B: BASE DE DATOS Y SISTEMA DE GESTIÓN DE LA BASE DE DATOS 122
Arquitectura de Referencia: Independencia en los Datos 124

1 MOTIVACIÓN. UN PROBLEMA SENCILLO.

Uno de los temas que más preocupa a la comunidad científica en la actualidad es la aparición nuevas cepas de microorganismos con una mayor resistencia a los antibióticos. Se sospecha que debido al uso indebido de determinados antibióticos en algunos casos (por automedicación, por retrasos en la identificación de los microorganismos así como la determinación de sus patrones de resistencia, etc.), y al contagio producido dentro de los mismos hospitales en otros, se está creando un caldo de cultivo para la aparición de nuevas variantes, potencialmente letales, de microorganismos multiresistentes.
Se desea controlar en cada momento qué microorganismo es aislado en una muestra concreta. El microorganismo se determina siguiendo la nomenclatura convencional de familia, género y especie pero determinando además la cepa concreta del aislado. Cada muestra lleva además información relativa al paciente (un código de identificación, nombre, apellidos, fecha de nacimiento y sexo), tipo de muestra, fecha y centro donde se extrae la muestra. De cada centro se almacena el nombre que lo identifica, un número de teléfono de contacto, la dirección, el código postal, la provincia y el país.
Relativo a cada especie se almacena una descripción de la misma, así como las múltiples cepas conocidas de dicha especie; y en caso de conocerse, la información relativa a la cepa que da origen por mutación a una nueva variante.
De cada cepa tendremos almacenado la concentración mínima inhibitoria (CMI) que presenta ante determinados antibióticos (de los cuales almacenamos su nombre y el grupo al que pertenece). Así mismo para cada especie tendremos almacenado igualmente un patrón general de resistencia propia de la especie (estos valores están derivados de las normas NCCLS que especifican tres valores [sensible, intermedio y resistente], nosotros por simplicidad utilizaremos únicamente uno de ellos, el intermedio por debajo del cual se considera sensible). Para cada género se almacena la lista de antibióticos para los cuales éste presenta una resistencia primaria. Y deberemos tener constancia de las reglas NCCLS que nos indican para cada familia y cada antibiótico el grupo al que pertenecen (todas las normas NCCLS incluyen el grupo, incluidas las relativas a especies, aunque nosotros sólo utilizamos esta información para familias).

También se nos informa que con cierta frecuencia puede ocurrir que dos muestras de una misma cepa sean identificadas inicialmente por error como distintas. Así que debemos añadir un procedimiento que dada dos cepas las fusione en una (dada dos cepas la primera indicada es la correcta y la segunda la errónea).

Igualmente se desea trazar la evolución geográfica del contagio de una cepa en concreto. Para ello será necesario que la base de datos visualice cronológicamente los primeros casos detectados en cada provincia (con su respectivo país) para una cepa dada. También es de vital importancia localizar los microorganismos multiresistentes que se define (para nuestro ejemplo sencillo) como todas aquellas cepas que han dado siempre como resultado un antibiograma resistente. Para ello se nos pide que se muestre el género, la especie y el código de cepa ordenados alfabéticamente.

Finalmente se desea la detección automática de toda nueva resistencia de una especie. Para ello el sistema debe detectar la aparición de una nueva resistencia de una especie sobre un antibiótico, indicando la fecha, centro, provincia y país donde ocurre.

2 LA APROXIMACIÓN AL PROBLEMA

La primera cuestión que se nos plantea para resolver el problema anterior es cómo abordar una solución válida al mismo. Existen diferentes aspectos que deben ser analizados en profundidad para abordar una situación como la anterior, no obstante uno de los principales va a ser el abordar las estructuras de la información que se ven implicadas (su representación, su gestión, su manipulación, etc.). Si nos fijamos en una situación como la anterior veremos que el volumen de información que va a ser manipulado puede ser bastante considerable. No sólo por la variedad de estructuras que van a participar sino también por el gran número de datos con el que pretendemos operar.
Ante situaciones de este tipo se hace necesario acudir a sistemas especializados en la gestión y manipulación de grandes volúmenes de información. Tradicionalmente éstos son los sistemas de bases de datos.

Para ver una introducción sobre la evolución histórica de los sistemas de bases de datos véase el Apéndice A.

Definir una Base de Datos es una tarea difícil, aunque no obstante una posible definición de que intente abarcar sus principales características podría ser la siguiente:

> "Colección de Datos estructurados según un modelo que refleje las entidades, relaciones y restricciones existentes en una parcela del mundo real. Los datos que han de poder ser compartidos y presentados en la mejor forma posible ante diferentes usuarios y aplicaciones que harán distintos usos de ellos. Estos datos deben mantenerse independientes de las aplicaciones; su definición y descripción han de ser únicas estando almacenadas junto con los mismos datos; y deberán estar almacenados sin redundancia perjudiciales o innecesarias. Los tratamientos habrán de conservar la integridad y privacidad de la información y además deberán realizarse de forma que permitan la recuperabilidad de la calidad de la información si esta se deteriora por cualquier causa".

Si desea entrar más en profundidad dentro de la definición de una base de datos y sus características fundamentales véase el Apéndice B.

Pare realizar el sencillo ejercicio de diseño y desarrollo de base de datos que se nos ha propuesto en el apartado anterior, debemos acudir a lo que normalmente se denomina proceso de diseño de una base de datos.

Usualmente, un Sistema de Bases de Datos (SBD) forma parte de un sistema de información mayor. Vamos a examinar cómo encaja el ciclo de vida de un SBD dentro del ciclo de vida típico de un sistema de información. Este consta de las siguientes fases:

- Análisis de factibilidad
- Análisis y recolección de requisitos
- Diseño

- Implantación
- Validación y realización de pruebas de aceptación
- Operación

Relacionándolo con este ciclo de vida clásico, nos vamos a centrar en el "micro" ciclo de vida que nos concierne, el del SBD:

- Definición del SBD, de su ámbito, sus usuarios y sus aplicaciones
- Diseño completo tanto físico como lógico
- Implementación de los esquemas conceptual, interno y externo. Creación de ficheros e implementación de las aplicaciones software
- Carga o conversión de Datos en los ficheros de la BD
- Conversión de aplicaciones previamente existentes
- Prueba y validación del sistema
- Operación del Sistema de Base de Datos y de las aplicaciones
- Monitorización y mantenimiento durante la fase operacional del sistema. Expansión, modificación y reorganización del sistema.

Este "micro" ciclo de vida del SBD se encuentra inmerso dentro del "macro" ciclo de vida del sistema de información general.

Si nos centramos en el segundo paso del ciclo de vida del SBD. Esta fase aborda el problema del diseño de la BD:

> " Diseñan la estructura física y lógica de una o mas Bases de Datos para resolver las necesidades de información de los diversos usuarios de una organización con relación a un conjunto de aplicaciones previamente definido."

El problema del diseño de la BD se ve agravado por el hecho de que se parte de unos requisitos informales y pobremente definidos y tras el proceso se obtiene un esquema de la BD, rígidamente definido y que no puede ser modificado fácilmente una vez que se ha implementado.

Es posible identificar seis fases en el proceso de diseño de la BD:

- Recolección y análisis de requisitos
- Diseño conceptual de la BD
- Elección del Sistema Gestor de la BD
- Correspondencia de Modelos de Datos (ó diseño lógico de la BD)
- Diseño físico de la BD
- Implementación de la BD

Como ya se expresó este proceso de diseño se realiza dentro del marco de un sistema de información y consta de dos actividades paralelas estrechamente interrelacionados:

- el diseño de los datos contenidos en la BD y su estructura
- el diseño del procesamiento de estos datos y las aplicaciones software

Tradicionalmente una de estas dos actividades ha sido considerada el centro del diseño llegándose a diseños de la BD guiados por datos o guiados por procesos.

Estas dos actividades son complementarias y es muy importante que se realicen en paralelo sin que ninguna de ellas enmascare la otra.

Las fases anteriores no tienen porque realizarse en secuencia y frecuentemente se realizaran realimentaciones entre las distintas fases.

La fase primera hace referencia a la recolección y análisis de requisitos, al ámbito de aplicación de la BD. La fase sexta trata sobre su implementación. Por ello, muchas veces estas fases no se consideran en sí mismas como parte del proceso de diseño de la Base de Datos sino como parte del ciclo de vida, más general, del sistema de información.

El núcleo del proceso de diseño de la BD son los pasos segundo, cuarto y quinto:

- **diseño conceptual**: en esta fase lo que se intenta es obtener un esquema conceptual, una descripción global de los datos, que sea independiente del SGBD específico que se use posteriormente. Se emplean modelos de Datos de alto nivel por su mayor capacidad expresiva. El más extendido es el modelo Entidad-Relación de Chen [Chen 76], sin embargo, este modelo es criticado por su pobre o nula expresividad operacional. En esta fase de diseño se deben de tener en cuenta las diferentes aplicaciones o transacciones de los usuarios. Estas aplicaciones deberán especificarse usando una notación independiente del SGBD concreto.
- **correspondencia entre Modelos de Datos o diseño lógico de la BD**: el esquema conceptual global obtenido en la fase anterior, mediante un Modelo de Datos de alto nivel, está muy lejos de poder ser directamente desarrollado en los Sistemas Gestores de Base de Datos basados en los modelos de Datos jerárquico, red y relacional. En esta fase se transforma el esquema conceptual obtenido en la fase anterior. Para ello se establece una correspondencia entre los conceptos usados en el Modelo de Datos de alto nivel en el que se ha expresado el esquema conceptual y los conceptos manejados por el Modelo de Datos asumido por el SGBD que se vaya a utilizar en la implementación. En términos de la arquitectura ANSI/SPARC el resultado de esta fase es el esquema conceptual de la BD expresado en términos del Modelo de Datos elegido. En esta fase deben diseñarse además los esquemas externos (vistas) de cada aplicación. Hay que tener en cuenta que no toda la expresividad de los modelos de Datos de alto nivel es directamente traducible a los modelos de Datos de los cuales existen actualmente implementaciones disponibles. Por esto, será necesario incluir parte de la semántica del Modelo de Datos de alto nivel en los programas de aplicación o generar ciertos programas de control que suplan estas deficiencias.
- **diseño físico**: esta fase consiste en el diseño de las especificaciones del esquema interno de la BD en tres etapas: (1) estructuras físicas de almacenamiento, (2)

localización y agrupación de registros físicos y (3) caminos de acceso a los datos. Es en esta fase cuando se tienen en cuenta las restricciones del hardware y del SGBD usado. En la parte de diseño del procesamiento de los datos, se deberán considerar las frecuencias de realización de las transacciones y de acceso a los datos así como las restricciones de rendimiento (tiempos de respuestas límite, métodos de acceso disponibles, etc.).

Hasta ahora el diseño de Bases de Datos ha sido un proceso prácticamente manual, pero se esta desarrollando una activa investigación para generar herramientas de ayuda a las diferentes fases del proceso de diseño. Las técnicas CASE se han introducido ya con éxito en algunas fases del diseño (automatización de las correspondencia entre modelos de Datos, ayuda a la concepción del diseño y optimización del rendimiento) y se están haciendo prometedores avances en otras de ellas.

Por otro lado existe una activa investigación en nuevos modelos de Datos de mucho más alto nivel que los actuales como son el modelo Entidad Relación o el Modelo de Datos orientado a objetos. Actualmente existen prototipos de investigación e incluso implementaciones comerciales parciales de Sistemas Gestores de Base de Datos que asumen estos nuevos modelos de Datos y estos nuevos sistemas no tardarán en empezar a imponerse (lentamente) en el mercado.

Para aprender más sobre Modelos de Datos acúdase al Apéndice C.

3 EL MODELADO CONCEPTUAL CON EL MODELO DE DATOS ENTIDAD-RELACIÓN

El Modelado Conceptual en una base de datos se realiza típicamente utilizando el Modelo de Datos Entidad-Relación (ER). Este modelo discrimina una serie de conceptos básicos que serán la base para representar las estructuras de nuestro problema.

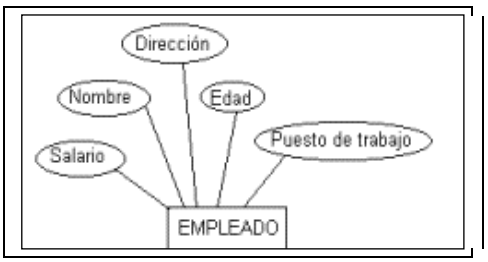
Como su propio nombre indica el Modelo de Datos Entidad-Relación define dos elementos principales:

- **Entidad**: Cualquier elemento del mundo real ya se concreto o abstracto susceptible de ser representado como ente con existencia propia e independiente. Así podemos representar como entidad a un libro, una institución, un trabajador, un puesto de trabajo e incluso un estado de ánimo.
- **Relación**: Cualquier tipo de asociación o conexión existente entre entidades susceptible de ser representada. Dada una entidad Persona y una entidad Empresa podría representarse como relaciones las siguientes: trabaja para, es cliente de, es director de, es gerente de,…

El modelo ER no se limita a estos dos conceptos básicos, sino que define una serie de elementos secundarios que lo enriquecen. Así toda entidad puede tener una colección de características propias que nos ayudan a describirla, a estas características denominaremos **atributos**. Así por ejemplo para una entidad Persona podríamos considerar atributos tales como nombre, apellidos, NIF, número de la seguridad social (NSS), fecha de nacimiento, sexo, una fotografía,…

El modelo ER establece un sistema gráfico basado en diagramas de representar sus conceptos. Aunque el sistema de representación original se sigue utilizando, hoy por hoy existe cierta tendencia a abandonarlo en pro de versiones más compactas gráficamente del mismo. Nosotros utilizaremos la representación original.

La entidad es representada con un cuadrado que contiene en su interior el nombre de la misma, y sus atributos como óvalos conectados a su entidad correspondiente con el nombre del atributo en el interior.



En el ejercicio propuesto al principio podemos comenzar ya a reconocer diferentes entidades y sus posibles atributos:

Uno de los temas que más preocupa a la comunidad científica en la actualidad es la aparición nuevas cepas de microorganismos con una mayor resistencia a los antibióticos. Se sospecha que debido al uso indebido de determinados antibióticos en algunos casos (por automedicación, por retrasos en la identificación de los microorganismos así como la determinación de sus patrones de resistencia, etc.), y al contagio producido dentro de los mismos hospitales en otros, se está creando un caldo de cultivo para la aparición de nuevas variantes, potencialmente letales, de microorganismos multiresistentes.
Se desea controlar en cada momento qué microorganismo es aislado en una muestra concreta. El microorganismo se determina siguiendo la nomenclatura convencional de **FAMILIA**, **GÉNERO** y **ESPECIE** pero determinando además la **CEPA** concreta del aislado. Cada muestra lleva además información relativa al **PACIENTE** (un código de identificación, nombre, apellidos, fecha de nacimiento y sexo), **TIPO** de muestra, fecha y centro donde se extrae la **MUESTRA**. De cada **CENTRO** se almacena el nombre que lo identifica, un número de teléfono de contacto, la dirección, el código postal, la **PROVINCIA** y el **PAÍS**.
Relativo a cada especie se almacena una descripción de la misma, así como las múltiples cepas conocidas de dicha especie; y en caso de conocerse, la información relativa a la cepa que da origen por mutación a una nueva variante.
De cada cepa tendremos almacenado la concentración mínima inhibitoria (CMI) que presenta ante determinados **ANTIBIÓTICOS** (de los cuales almacenamos su nombre y el **GRUPO** al que pertenece). Así mismo para cada especie tendremos almacenado igualmente un patrón general de resistencia propia de la especie (estos valores están derivados de las normas NCCLS que especifican tres valores [sensible, intermedio y resistente], nosotros por simplicidad utilizaremos únicamente uno de ellos, el intermedio por debajo del cual se considera sensible). Para cada género se almacena la lista de antibióticos para los cuales éste presenta una resistencia primaria. Y deberemos tener constancia de las reglas NCCLS que nos indican para cada familia y cada antibiótico el grupo al que pertenecen (todas las normas NCCLS incluyen el grupo, incluidas las relativas a especies, aunque nosotros sólo utilizamos esta información para familias).

También se nos informa que con cierta frecuencia puede ocurrir que dos muestras de una misma cepa sean identificadas inicialmente por error como distintas. Así que debemos añadir un procedimiento que dada dos cepas las fusione en una (dada dos cepas la primera indicada es la correcta y la segunda la errónea).

Igualmente se desea trazar la evolución geográfica del contagio de una cepa en concreto. Para ello será necesario que la base de datos visualice cronológicamente los primeros casos detectados en cada provincia (con su respectivo país) para una cepa dada. También es de vital importancia localizar los microorganismos multiresistentes que se define (para nuestro ejemplo sencillo) como todas aquellas cepas que han dado siempre como resultado un antibiograma resistente. Para ello se nos pide que se muestre el género, la especie y el código de cepa ordenados alfabéticamente.

Finalmente se desea la detección automática de toda nueva resistencia de una especie. Para ello el sistema debe detectar la aparición de una nueva resistencia de una especie sobre un antibiótico, indicando la fecha, centro, provincia y país donde ocurre.

Todo atributo puede ser:

- Simple o compuesto: dependiendo de si consta de un único atributo o de varios. Los atributos compuestos se denotan como:

- Monovaluado o multivaluado: según pueda tomar un único valor o varios para un mismo individuo. Los atributos multivaluados se representan con un doble óvalo.

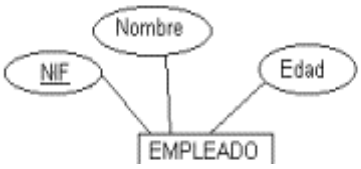
- Almacenado o derivado: según esté almacenado físicamente su valor o éste se calcule en función a otros valores cuando se necesite. Los atributos derivados se representan con un óvalo discontinuo.

Bajo algunas condiciones el valor para un atributo ser desconocido o simplemente no aplicable debido a la falta de dicha característica en un individuo en concreto. Por ejemplo, podría ocurrir que en un momento dado no se conociese la fecha de nacimiento de un individuo o simplemente que un niño o turista extranjero no posea un NIF. Es en estos casos cuando el valor de un atributo puede tomar lo que se denomina **valor nulo** o simplemente NULL. Esta constante está definida en todos los tipos permitidos en una base de datos e indica el desconocimiento o "no aplicabilidad" del valor concreto. En una base de datos podemos especificar si un atributo puede o no tomar el valor nulo.

Otra restricción muy importante que puede definirse para un atributo o un conjunto de atributo es lo que se denomina restricción de **clave** o de **unicidad**. Esta restricción afecta a un atributo o conjunto de ellos e indica a la base de datos que no se permiten repeticiones en los valores de los atributos para los que se definen. Así, por ejemplo, puedo tener un atributo NSS (número de la seguridad social) para un trabajador que debería ser único para dicha entidad, esto es, no pueden existir dos trabajadores con el mismo NSS. Se dice entonces que NSS es una clave o que es un valor único. Así mismo podríamos forzar a que no existiesen dos trabajadores con los mismos nombres y apellidos. Entonces decimos el conjunto de atributos {nombre, apellidos} conforman una clave. En este caso podemos tener trabajadores que tengan igual nombre o iguales apellidos, pero nunca se podrá repetir el mismo nombre y apellidos.

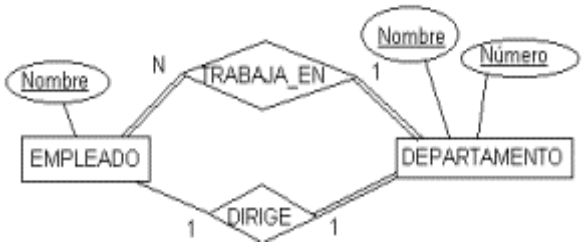
Es de vital importancia identificar correctamente qué atributos o conjuntos de los mismos conforman nuestras claves, ya que un error podría conducirnos tener información errónea en la base de datos o incluso no poder almacenarla. Por ejemplo, puede ser arriesgado establecer una clave sobre {nombre, apellidos} ya que existen personas para los cuales coinciden.

Gráficamente una clave se expresa con el nombre subrayado.



Para expresar claves definidas sobre múltiple atributos no hay más remedio que acudir a la definición de atributos compuestos.

Cuando queremos expresar una asociación o vínculo existente entre dos relaciones acudimos al concepto de relación. En el diagrama relacional la relación se representa por medio de un rombo en cuyo interior denotamos el nombre de la misma. Así por ejemplo si tenemos dos entidades Empleado y Departamento y queremos expresar dos relaciones diferentes: "trabaja en" que me relaciona aquellos empleados y los departamentos para los a que trabajan y "dirige" que me relaciona un departamento con su director, la representación sería:



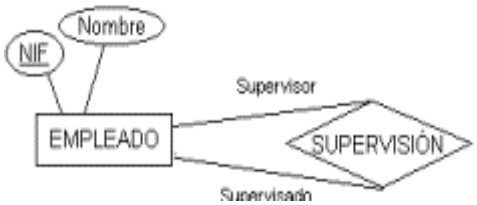
Según su cardinalidad las relaciones pueden ser de tres tipos:

- **Relaciones uno a uno** (1:1): son aquellas en las que cada vínculo o asociación afecta únicamente a un individuo de cada entidad participante. Así en el ejemplo anterior Dirige es un ejemplo de este tipo de asociación, pues todo departamento tiene un director y normalmente un empleado únicamente puede ser a lo sumo director de un departamento.
- **Relaciones uno a muchos** (1:N): son aquellas en las que cada vínculo o asociación afecta por un lado a un individuo de una entidad y por otro lado a un colectivo de individuos de la otra entidad participante. Así en el ejemplo anterior Trabaja_en relaciona cada departamento con el conjunto de sus empleados (un departamento tiene

muchos empleados trabajando para él, pero un empleado sólo puede trabajar para un departamento).

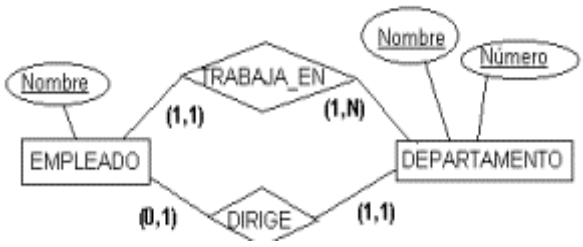
- **Relaciones muchos a muchos** (N:M): Cuando un colectivo de individuos de una entidad se asocian con otro colectivo de individuos de otra se dice que se establece una relación muchos a muchos. Un buen ejemplo de ellos sería la relación Matriculado entre las entidades Alumno y Curso. Un alumno puede estar matriculado en muchos cursos pero al mismo tiempo un curso puede tener muchos alumnos matriculados.

Se define **grado** de una relación como el número de entidades que participan en ella. Generalmente la mayoría de las relaciones son siempre binarias, aunque podemos encontrar ternarias o superiores. En ocasiones es necesario especificar un nombre para describir el rol que ocupa una entidad en una participación. Esto es especialmente útil en relaciones recursivas o reflexivas, que son aquellas que se establecen entre una misma relación. Por ejemplo:



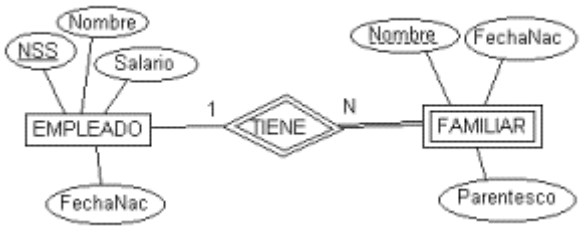
En el ejemplo anterior tenemos el tipo de relación Supervisión que relaciona un empleado con su supervisor y ambos son miembros del mismo tipo de entidad Empleado. Así el tipo de entidad Empleado participa dos veces en la relación Supervisión pero una vez con el rol Supervisor y otro con el rol Supervisado.

Existe una restricción, denominada de participación que nos indica si todo individuo de una entidad está o no obligado a participar en dicha relación, teniendo así participaciones **totales** (cuando todos están forzados a participar) y **parciales** (cuando existen individuos que no se ven involucrados en dicha asociación o vinculación). Este tipo de restricción junto con la razón de cardinalidad constituyen lo que se denomina **restricciones estructurales**. Existe una forma sencilla de expresar de forma directa estas restricciones estructurales, y es indicando en el diagrama las cardinalidades mínimas y máximas en una relación. Veámoslo con un ejemplo:



Aquí todo empleado debe trabajar para un y sólo un departamento. Un departamento debe tener al menos un empleado. Los departamentos tienen un único director y además un empleado, de serlo, sólo puede ser director de un departamento.

Como último tipo de relación vamos a ver las que se denominan **relaciones de identificación** que son aquellas que establecen una relación entre una entidad y una entidad subordinada o dependiente. A estas entidades subordinadas se las denomina débiles y se caracterizan por no tener existencia independiente sino dependiente de otro (en nuestra representación claro está). Las entidades débiles por tanto no poseen claves tal y como las conocemos aunque sí pueden poseer lo que se denomina **clave parcial** y que constituye un identificador único sólo entre los individuos dependientes de uno en concreto. A las entidades no débiles se las denomina en contraposición **entidades fuertes**. Veamos un ejemplo que al mismo tiempo nos muestra su notación:



En la figura anterior tenemos un diagrama que representa dos tipos de entidad: uno débil, FAMILIAR y otro fuerte, EMPLEADO que es el tipo de entidad propietario de FAMILIAR. Como podemos observar, dos entidades del tipo de entidad FAMILIAR pueden ser iguales al no existir un atributo clave que las identifique. Dos entidades familiar pueden tener los mismos valores para sus atributos Nombre, FechaNac y Parentesco. Se identifican como entidades distintas sólo después de determinar la entidad empleado particular con la que está relacionada cada una. Normalmente, los tipos de entidad débiles tienen una **clave parcial**, que es el conjunto de atributos que pueden identificar de manera única las entidades débiles relacionadas con la misma entidad propietaria. En el ejemplo de la figura 11, tenemos como clave parcial de FAMILIAR el atributo Nombre, porque para un mismo empleado no puede haber dos familiares con el mismo nombre. Las claves parciales se subrayan en los diagramas con líneas discontinuas. Hay ocasiones en las que los tipos de entidad débiles se representan en forma de atributos complejos (multivaluados, compuestos).

Un problema de diseño: límite entre atributo y entidad.

Aunque es arriesgado generalizar este tipo de reglas, un buen heurístico a la hora de diseñar un modelo ER partiendo de un texto narrativo es considerar que en el lenguaje natural las entidades suelen ser descritas como sujetos u objetos en las oraciones, es decir como sustantivos, mientras que las relaciones suelen expresarse como verbos que vinculan a sujetos y objetos (otras entidades). Los atributos de una entidad aparecen igualmente como sustantivos pero cuya semántica es puramente descriptiva de otra entidad.

Generalmente cualquier ciclo de vida, ya sea de una aplicación, de una base de datos, o de cualquier otro tipo suele ser siempre un proceso que requiere una realimentación para la detección de errores y el refinamiento progresivo. Durante su ciclo de vida un esquema de base de datos evoluciona y cambia progresivamente para mejorar la representación de la solución del problema que está expresando. En las primeras fases nuestro diseño puede alterarse sustancialmente, en las fases más tardías los cambios tenderán a ser más sutiles, debido principalmente a que generalmente una base de datos suele ser atacada por más de una aplicación, y un cambio en su esquema obligaría a modificar a todas las aplicaciones que la atacasen.

Hasta ahora estamos hablando del diseño como un proceso análisis de un problema y de identificación de conceptos en un problema para representar una solución posible (en el caso de bases de datos una representación válida de las estructuras de datos participantes para poder almacenar, gestionar y procesar la información asociada al mismo). Sin el diseño es un proceso que no está libre de una fuerte carga subjetiva, esto es, para un mismo problema podemos tener múltiples diagramas ER válidos que lo representan (y con múltiples nos no a unos cuantos sino a muchos). Aunque toda esta miríada de diferentes modelos sea válida no todos tendrán las mismas características ni se comportarán igual bajo distintos entornos. Existirán posiblemente modelos complejos, ricos y muy expresivos que nos formalizarán hasta las más mínimas sutilezas de nuestro problema, por otro lado tendremos modelos mucho más sencillos, en los que no habremos expresado muchos de los matices que existen en nuestro problema, pero que quizás almacenan la información de tal forma que puede ser procesada muy eficientemente.

Existen muchos factores que pueden motivar la aparición de múltiples soluciones para representar el mismo problema. Uno de los más comunes es determinar cuándo un atributo constituye una entidad o cuándo una entidad puede o debe ser degradada a un atributo. Técnicamente en un modelo tenemos la capacidad de degradar cualquier entidad a atributos o de ascender atributos a categorías de entidades. Esta capacidad se puede y ha sido llevada a sus máximos extremos: un proceso de normalización (proceso muy común en bases de datos que no veremos en este módulo por su complejidad) exageradamente agresivo tiende a generar multitud de entidades partiendo de atributos; y por otro lado la teoría de la relación universal anidad nos dice que es posible degradar absolutamente todas las entidades de nuestro esquema a atributos de una única superentidad denominada relación o en este caso entidad universal.

Planteémonos un ejemplo para ver mejor este problema. Es muy común encontrar entidades que describen a personas con un atributo que nos indica el nombre del país donde nació o donde tiene fijada actualmente su residencia. El problema está en cuándo “país” pasa de ser un mero atributo que describe una característica de estas personas para constituir una elemento del mundo real con entidad propia e independiente, esto es una nueva entidad vinculada con la de personas por una relación que podría denominarse nacido_en o reside_actualmente_en. Lo cierto es que no existe ningún punto objetivo que nos proporcione ese límite, siempre es una cuestión subjetiva, aunque la verdad es que hay algunos heurísticos que pueden ayudarnos.

Decimos que no hay ningún límite claro objetivo ya que cualquier atributo puede considerarse una entidad, por ejemplo el atributo nombre podría elevarse si consideramos que los nombres son elementos con existencia propia (el nombre 'Pablo' tiene existencia propia independientemente de cualquier individuo que se llame 'Pablo'). Sin embargo este proceso puede conducirnos a situaciones a las que quizás no nos gustaría o no deberíamos llegar. Por ejemplo todos los atributos que indiquen una fecha podrían fusionarse en una única entidad fecha, ya que un día tiene existencia propia. Sin embargo una fecha puede ser interpretado no como una entidad, sino como una relación entre otras tres entidades: día, mes y año. Pero año y día no son más que números, luego realmente podrían ser representados como una entidad Número.

En ambos extremos, ya sea un esquema sobrecargado de entidades y relaciones o un esquema con una única relación universal, nos encontraremos con severísimos problemas de eficiencia y una viabilidad práctica más que dudosa.

El Modelo Entidad-Relación ha sido en varias ocasiones, siendo la más conocida de sus extensiones el denominado Modelo Entidad-Relación Extendido o ERE. Para conocer más sobre esta extensión acúdase a la bibliografía sobre el tema.

De vuelta a nuestro problema

En el ejercicio propuesto al principio se plantea la realización de una base de datos para el almacenamiento información relativa a resistencia antimicrobiana. Para facilitar su seguimiento hemos simplificado al máximo el universo de nuestro problema, reduciendo en todo lo posible el número de entidades, relaciones y atributos participantes en nuestro minimundo (la parte del universo de nuestro problema sobre la que nos vamos a centrar).

Nuestro cometido es el desarrollo de una base de datos sobre la que posteriormente se desarrollarán varias aplicaciones. El analista nos ha simplificado el trabajo esquematizando y reduciendo en lo posible la descripción de nuestro problema; y determinando la información necesaria para el funcionamiento de los futuros aplicativos, así como la información necesaria para el funcionamiento interno de las reglas de integridad que regirán a nuestra base de datos.

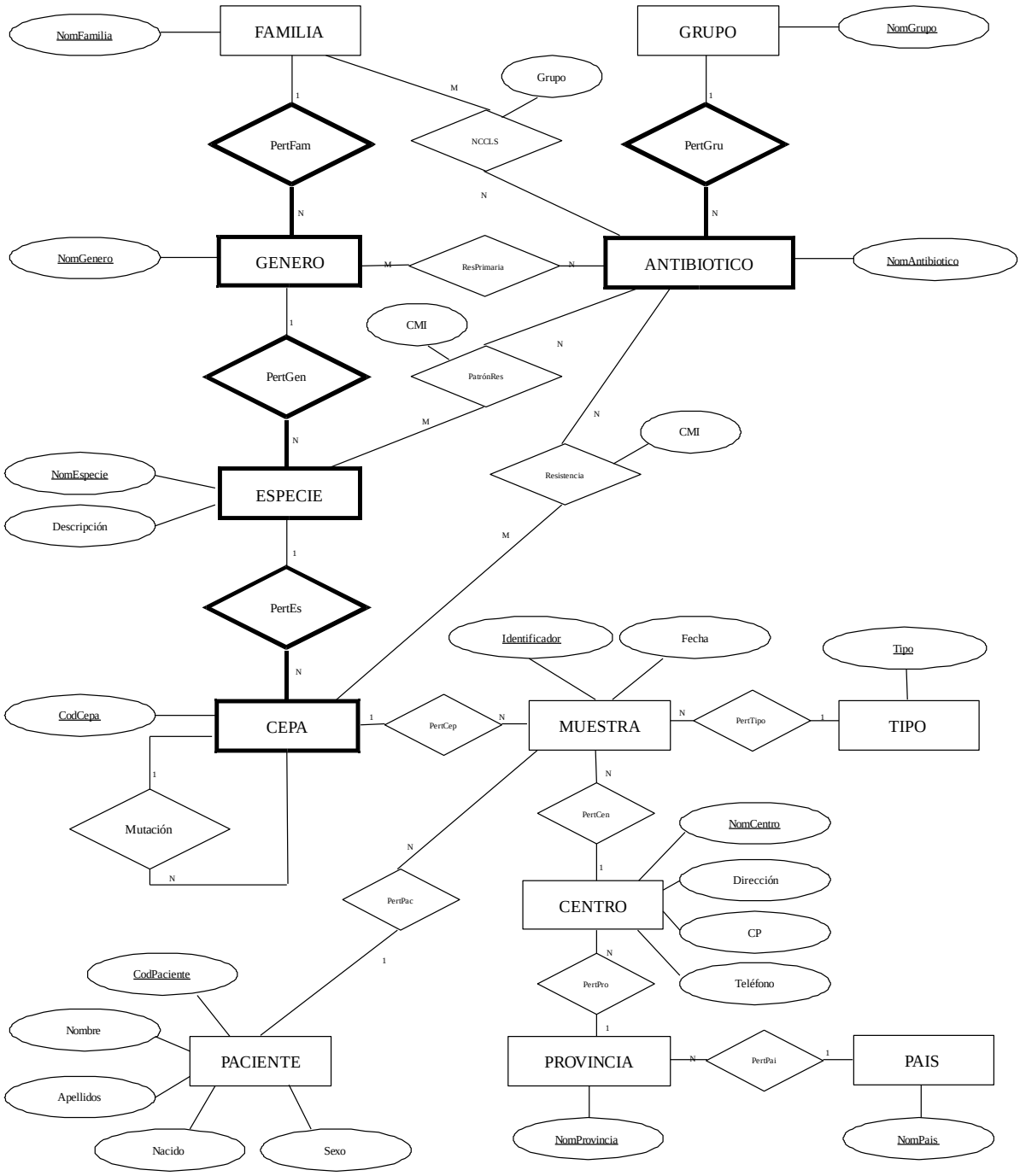
Adicionalmente concretamos un poco más nuestro ejercicio indicando que el sistema debe detectar la aparición de una nueva resistencia de una especie sobre un antibiótico, indicando la fecha, centro, provincia y país donde ocurre. Los mensajes de advertencias serán almacenados en una tabla que contendrá dos atributos. El primero de ellos es la fecha de la muestra y el segundo el mensaje de advertencia, el cual tendrá el siguiente formato:

> Se ha detectado una nueva resistencia sobre la especie *Enterococcus fecalis* al antibiótico Vancomicina en H.R. Carlos Haya (Málaga) – España.

A la hora de realizar el diseño de este esquema hemos de tener en cuenta que no existe una solución única. Dependiendo de la metodología de diseño, de la interpretación del problema y de otros factores tanto objetivos como subjetivos podremos llegar a soluciones diferentes, alguna de las cuales serán consideradas mejores que otras siempre en relación al uso que se realice de la base de datos y a los requisitos que se le exijan.

Ante falta de un escenario real siempre intentaremos optar por esquemas normalizados, claros y expresivos, atendiendo principalmente a las reglas heurísticas que suelen aparecer en la bibliografía más importante sobre bases de datos. Dado que la tabla de advertencias es independiente a todas las demás y utilizada únicamente cuando se detecte la restricción especificada no se va a incluir dicha entidad en el esquema inicial y la generaremos junto con el trigger de detección.

Así pues una primera solución que podemos encontrar es:



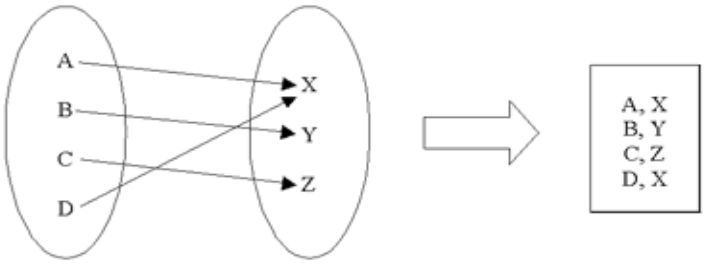
En este primer esquema hemos representado de forma normalizada todos los elementos de nuestro problema. Debido a la simplificación que hemos realizado del enunciado mismo tenemos muchas entidades cuyo único atributo es su identificador.

4 El Modelo de Datos Relacional

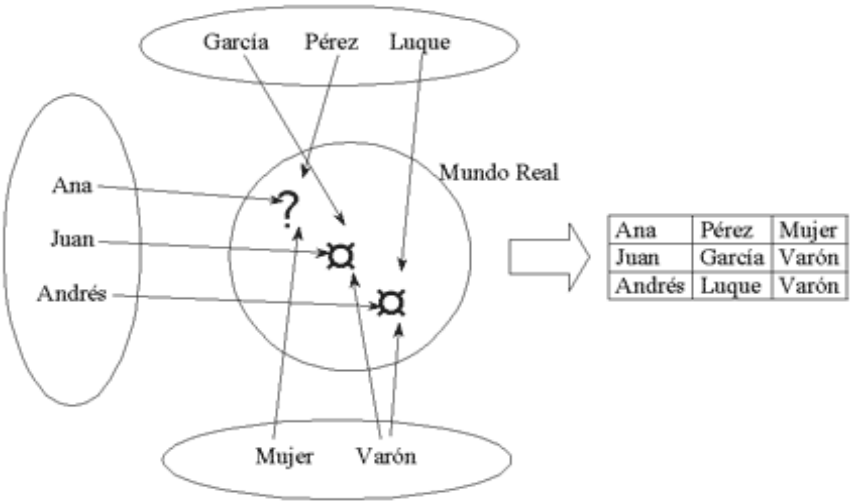
Una vez visto el modelo Entidad-Relación, que como sabemos es un modelo de datos conceptual, vamos a abordar ahora el modelo lógico que ha tenido un mayor éxito y una mayor difusión: el modelo de datos Relacional. Es necesario indicar que el modelo de datos relacional no es el único modelo lógico existente. Tradicionalmente durante lo que hoy día se acepta como los orígenes de las bases de datos (final de los años 60 a principio de los 70) se desarrollaron tres modelos: el modelo Jerárquico, el modelo en Red y el modelo Relacional. Los dos primeros fueron los precursores del Relacional, aunque hoy día podemos encontrar aún sistemas basados en los dos primeros.
Aunque esto es lo que podemos encontrar en casi toda la bibliografía existente sobre bases de datos, varios autores, incluído E.F. Codd, el autor del modelo Relacional, han comentado que la cronología no es exactamente esa. Afirman que en primer lugar se desarrollaron los sistemas Jerárquicos, posteriormente los basados en Red, en 1970 se desarrolló el Modelo Relacional, y posterior a él se desarrollaron los Modelos Jerárquico y en Red y los sistemas Relacionales. Codd argumentaba que el éxito del Modelo Relacional residía, entre otros factores pero principalmente, en que al contrario de sus dos competidores que fueron modelos teóricos desarrollados para formalizar el comportamiento de sistemas ya implementados, el suyo fue el único desarrollado primero teóricamente y después llevado a la práctica, gracias a lo cual lo dotó de un fuerte fundamento matemático, además de una gran simplicidad.

Partiendo del concepto de relación matemática, Codd desarrolló un potente modelo que hace uso de la teoría de conjuntos y de la lógica de predicados de primer orden.

El concepto fundamental del modelo de Codd, la relación tiene una naturaleza dual, ya que es interpretada simultáneamente como relación matemática y como conjunto.



Y se usan para representar elementos del mundo real.



Para Codd un **esquema de relación** está compuesto por una lista de **atributos**, donde cada atributo puede adquirir valores de su correspondiente **dominio**. Al ser el dominio el conjunto de valores posibles para un atributo éste puede definirse como cualquier conjunto matemático, bien por extensión o por intensión. No obstante en la vida real la definición de un dominio simplificarse sobremanera, quedando reducida a la especificación de un tipo de datos y quizás una máscara de formato. Formalmente la asociación de un atributo a un dominio se denomina **restricción de dominio**.

Al igual que teníamos en el modelo Entidad-Relación, el modelo Relacional puede definir restricciones de **clave** sobre un atributo o un conjunto de ellos. No obstante y a diferencia del modelo ER de todas las claves posibles para una relación siempre se selecciona una de ellas que será denominada **clave primaria**, siendo el resto denominado **claves alternativas**. La elección de la clave primaria de entre todas las posibles es en principio un proceso arbitrario, aunque en la práctica está motivado por cuestiones de eficiencia (siendo usualmente los candidatos ideales los tipos numéricos simples, seguido de tipos simples y en último claves compuestas formadas por varios atributos).

A partir de este punto Codd define un **esquema de base de datos** como unión de un conjunto de relaciones y de un conjunto de **restricciones de integridad**. Estas últimas puede ser a su vez de dos tipos: restricción de integridad de entidades y restricción de integridad referencial. La primera de ellas establece que ninguna clave primaria (ni ninguno de los atributos que la componen) puede tomar el valor nulo. La segunda de ellas se establece entre dos entidades (o una en caso de una relación reflexiva) y nos indica que una tupla de una de las entidades está haciendo referencia a una tupla de la otra a través de lo que se denomina **clave externa**. Una clave externa no es más que un atributo o conjunto de ellos existentes en una tabla que coincide en dominio (y por tanto en tipo) con la clave primaria de la entidad referenciada. El valor de esta clave puede ser nulo (si no se especifica lo contrario) o uno de los valores existentes de la clave primaria de la entidad referenciada.

Departamento	
Identificador	**NombreDpto**

1	Ventas
2	Investigación
3	Administración

Empleado			
DNI	**Nombre**	**Apellidos**	**TrabajaEn**
1	Juan	Pérez García	2
2	Antonio	Fernández Abad	
3	Luís	Martín Martín	1

En este ejemplo TrabajaEn es una clave externa de la relación Empleado que está haciendo referencia a la relación Departamento, concretamente a su clave primaria.

5 ALGORITMO DE TRADUCCIÓN DE ENTIDAD RELACIÓN EXTENDIDO A MODELO RELACIONAL

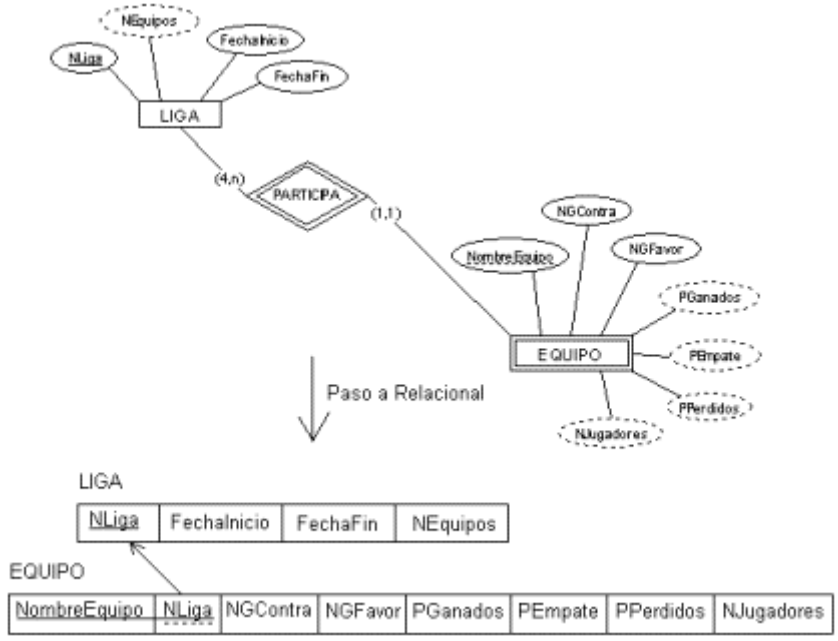
Ahora vamos a ver el para transformar un modelo ER o EER en un modelo Relacional. Los pasos a necesarios para la transformación junto con algunos ejemplos son los siguientes.

Nota: debido a una ambigüedad del lenguaje se denomina igual a la relación del modelo Relacional y a la relación del modelo Entidad-Relación. Para evitar esta ambigüedad en este apartado vamos a denotar siempre que pueda existir duda las relaciones del modelo conceptual como relación^{ER}.

1. Cada entidad normal (no débil) genera una relación equivalente que contiene únicamente sus atributos simples. De existir atributos compuestos únicamente se incluyen los atributos simples componentes. Se selecciona una, en principio cualquiera de las claves de la entidad y se hace clave primaria en la nueva relación.

   En la figura anterior vemos una entidad fuerte y su transformación. Obtenemos una relación de nombre EMPLEADO y con cinco atributos: NIF, Nombre, Apellidos, Departamento y Salario. De las dos claves del modelo hemos elegido como primaria el atributo NIF. El atributo compuesto queda representado en la relación como dos atributos simples.

2. Por cada entidad débil se genera una relación siguiendo el mismo procedimiento del paso anterior pero se incluye entre los atributos de la nueva relación una clave externa que hace referencia a la entidad propietaria. La clave primaria de la nueva relación estará compuesta por esta clave externa junto con una de las claves parciales definida en la entidad débil. Veamos la transformación con otro ejemplo:



Podemos ver que la entidad débil EQUIPO da lugar a la relación EQUIPO con ocho atributos: los atributos de la entidad débil y la clave primaria de la entidad propietaria (NLiga). El atributo NLiga es una clave externa y lo indicamos subrayandola con línea discontinua. La clave primaria está formada por la clave externa NLiga y lo que era la clave parcial del tipo de entidad débil EQUIPO (atributo NombreEquipo).

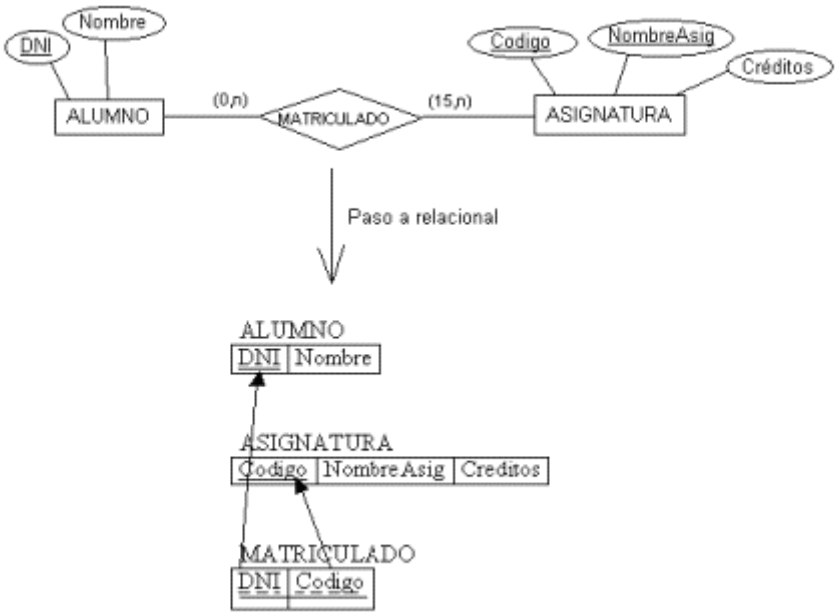
3. Por cada relación[ER] binaria 1:1 tenemos tres opciones:
   3.1. Migrar de forma cruzada las claves primarias de ambas relaciones como externas en la relación opuesta (este procedimiento es algo engorroso por lo que casi siempre suele acudirse a la segunda opción). Los atributos de la relación[ER] son llevados normalmente a una cualquiera de las relaciones.
   3.2. Se migra la clave primaria de una de las relaciones como externa de la otra. De ser posible seleccionaremos como receptora de la clave externa a la relación correspondiente a aquella entidad con participación total (de esta forma controlaremos esto especificando que la dicha clave no puede valer nulo). Estableceremos adicionalmente una restricción de unicidad en la clave externa. Los atributos de la relación[ER] son llevados normalmente a la relación que recibe la clave externa.
   3.3. La tercera es la más rara de todas y únicamente posible cuando la relación[ER] se establece entre una entidad débil y su entidad propietaria. Consiste en la absorción total de la entidad débil por parte de su propietaria, esto es, todos los atributos de la débil pasan a ser atributos de la propietaria, desapareciendo la relación correspondiente a la entidad débil.

4. Las relaciones[ER] binarias 1:N generan una clave externa en la relación correspondiente a la entidad del lado N. Todos los atributos de la relación[ER] son llevados a la relación equivalente del lado N.

Observemos en un ejemplo los dos pasos anteriores:

Primero transformamos las entidades en las relaciones EMPLEADO y DEPARTAMENTO con los atributos correspondientes. Luego transformamos la relación DIRIGE que es una relación[ER] 1:1 de la siguiente forma: como DEPARTAMENTO es una entidad con participación total en el tipo de relación[ER] DIRIGE migramos la clave primaria de EMPLEADO como clave externa en la relación DEPARTAMENTO. Ahora trataremos la relacion[ER] TRABAJA_EN que es una relación 1:N. En este caso migramos la clave primaria de la relación de la entidad del lado 1 a la relación de la entidad del lado N. Es importante destacar, que con la notación (min,max) el tipo de entidad del lado N es el contrario al que aparece la indicación. En este caso migramos la clave primaria de la relación DEPARTAMENTO (que representa el tipo de entidad DEPARTAMENTO) a la relación EMPLEADO (que representa el tipo de entidad EMPLEADO).

5. Para cada relación[ER] M:N se genera una nueva relación que incluye todos los atributos de la relación[ER], y dos claves externas: una de cada entidad. Estas dos claves externas constituirán la clave primaria de esta nueva relación.



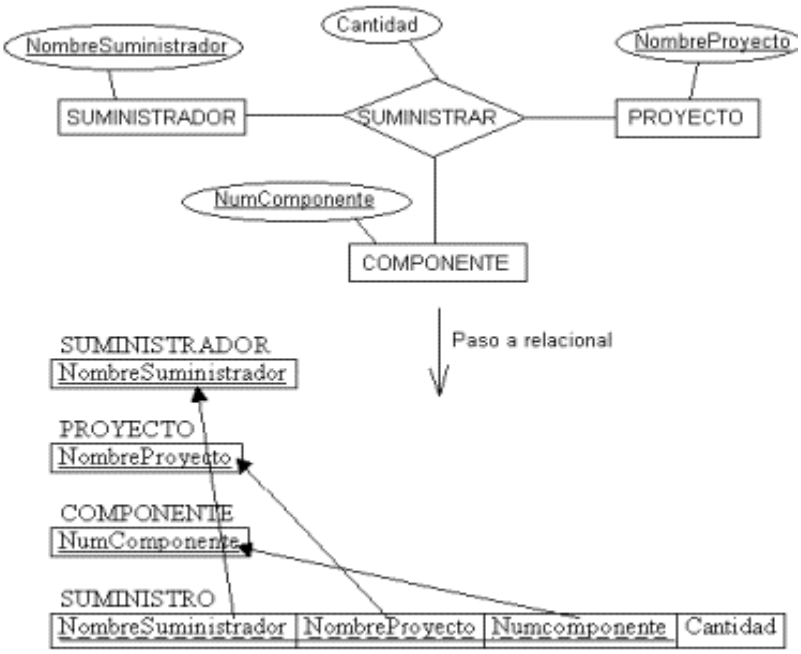
En la figura anterior vemos un relación M:N y su transformación. Además de las relaciones resultantes de la transformación de las entidades fuertes hemos creado una relación (MATRICULADO) con dos claves externas: una es la clave primaria de la relación ALUMNO y la otra la clave primaria de la relación ASIGNATURA.

Aunque se han especificado transformaciones particulares (casos 3 y 4) para las de relaciones[ER] 1:1 y 1:N, siempre podremos optar por utilizar el caso 5 para cualquiera de ellas. Ya que las relaciones 1:1 se pueden ver como casos particulares de relaciones 1:N y de la misma forma las relaciones 1:N pueden verse como particularizaciones de relaciones N:M. Tomar esta alternativa en lugar de los casos 3 y 4 puede ser aconsejables en aquellos casos en los que existen pocas ocurrencias de la relación.

6. Como el modelo relacional no permite atributos multivaluados lo que hacemos es alterar implícitamente el modelo conceptual previamente y elevamos el atributo a la categoría de entidad débil relacionada (1:N) con la entidad principal y donde el atributo que contiene el valor será una clave parcial.

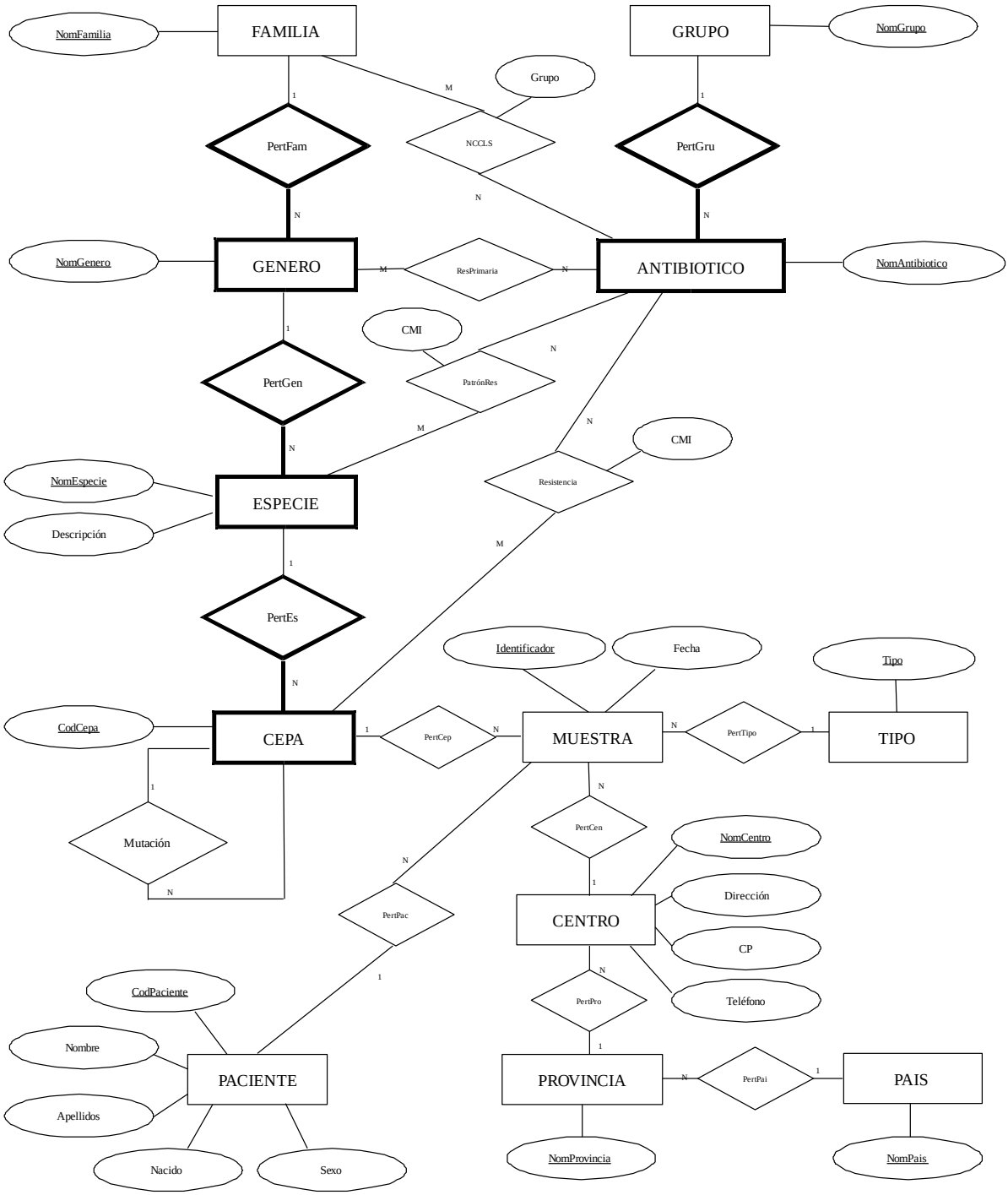
Como vemos en la figura 5 la transformación de un atributo multivaluado es sencilla. Se crea una relación para el atributo multivaluado donde la clave es la combinación del atributo y una clave externa que corresponde a la clave primaria de la relación del tipo de entidad que contiene el atributo.

7. Por cada relación n-aria de grado mayor a 2, esto es, relaciones ternarias o superiores. Se procede con una generalización del paso 5. Se forma una nueva relación que contiene todos los atributos de la relación[ER], y las claves externas que referencian a cada entidad participante. La clave primaria estará compuesta por todas las claves migradas correspondientes a las entidades que participaban con cardinalidad N. Un ejemplo de transformación es el siguiente.

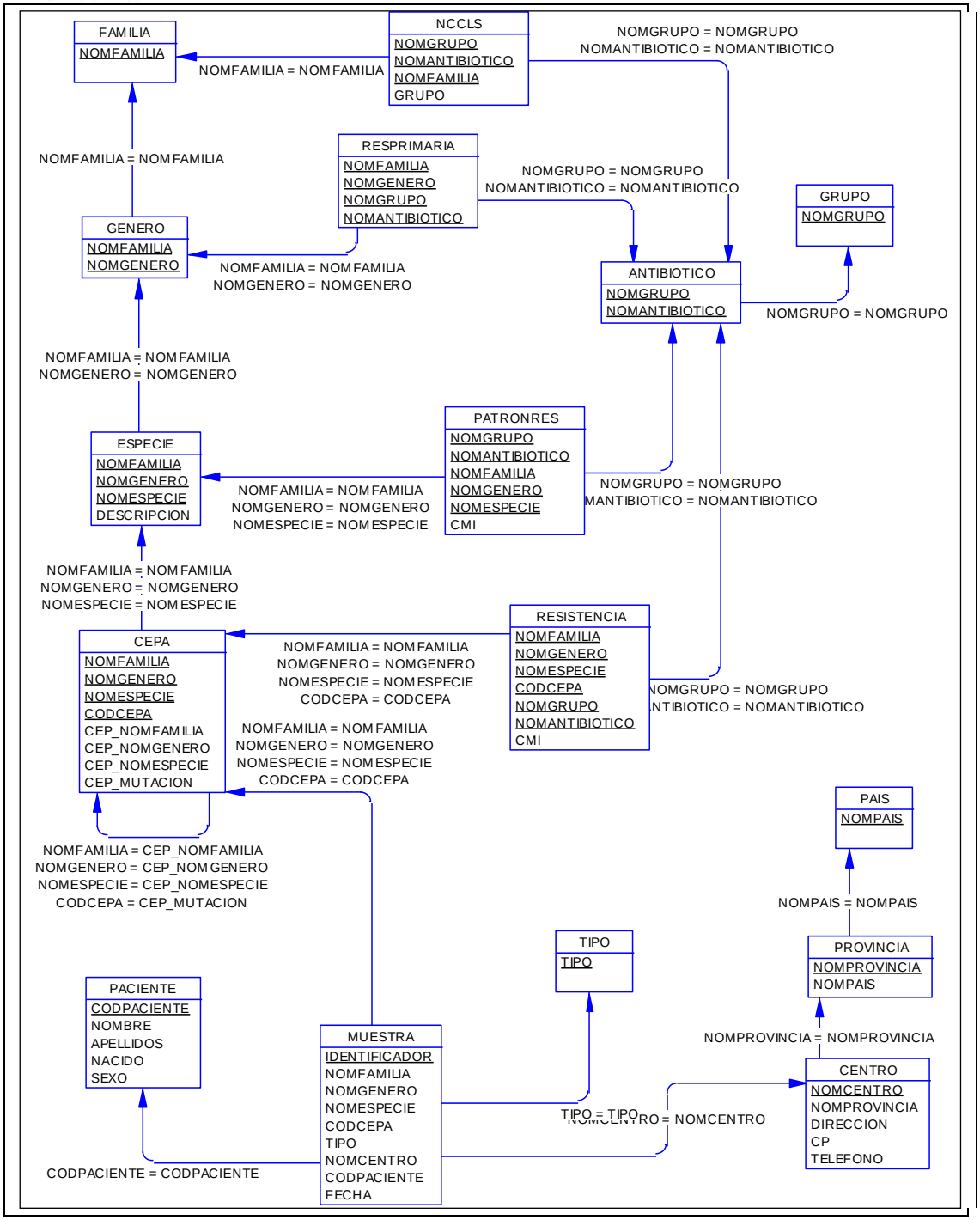


De vuelta con nuestro ejemplo.

Recordemos que ya teníamos nuestro modelo Entidad Relación:



No obstante este esquema entidad relación presenta un problema colateral que se hace patente al representar su esquema relacional.



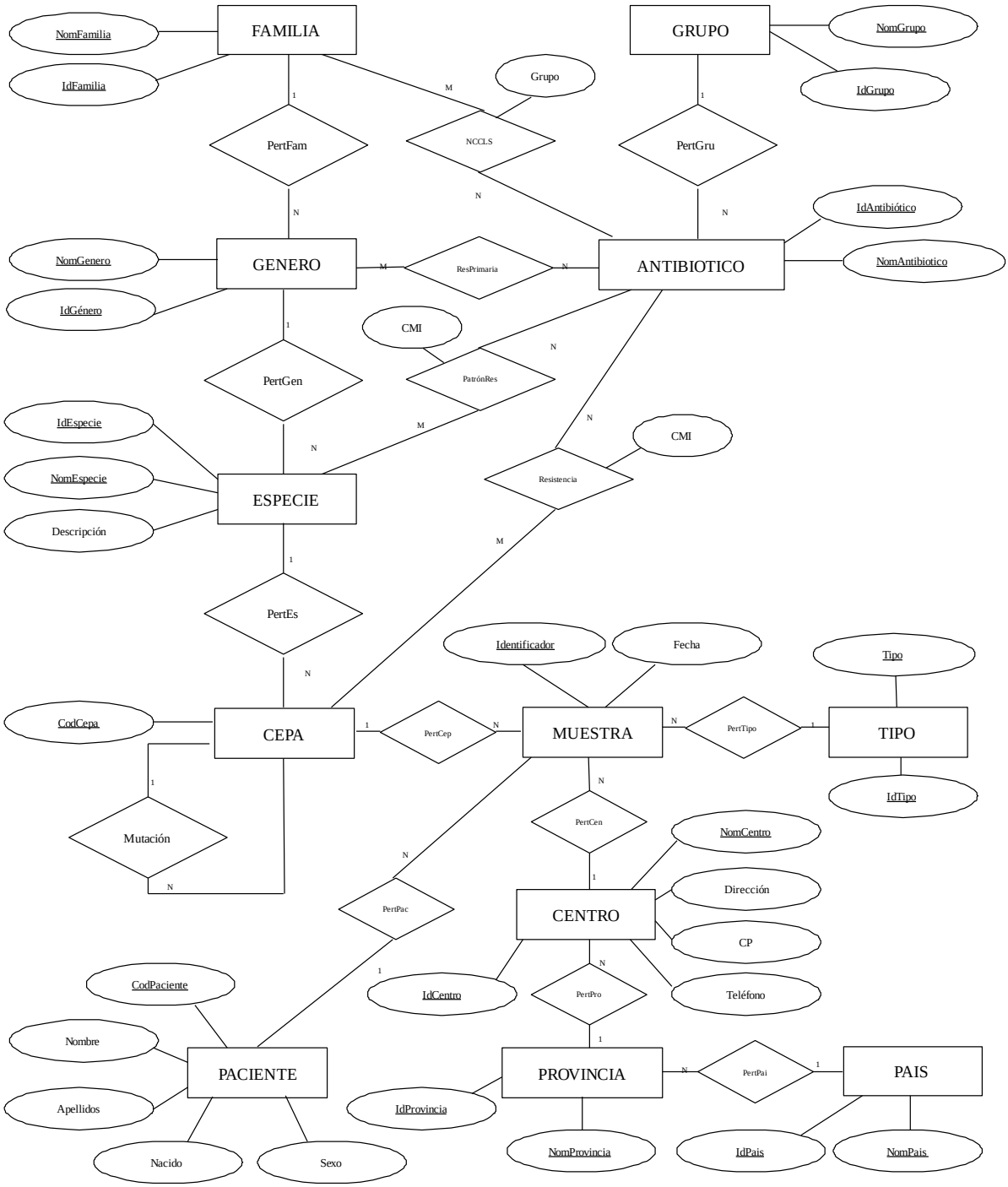
En este esquema relacional se hace patente una alta densidad de atributos participantes en restricciones de integridad referencial. Este problema proviene de un uso excesivo de entidades y relaciones débiles o dependientes. Si nos fijamos en la sucesión familia-género-especie-cepa, a un nivel conceptual tan alto como el modelo entidad-relación es una sucesión de entidades débiles y es quizás la forma más fiel de representación del problema. No

obstante esto motiva una cadena de migración de claves haciendo que cada clave migrada forme parte de la clave primaria de la entidad débil. Así ante cadenas largas obtendremos claves formadas por múltiples atributos, como podemos observar en las sentencias SQL que implementan el esquema y que aparecen en el Apéndice D.

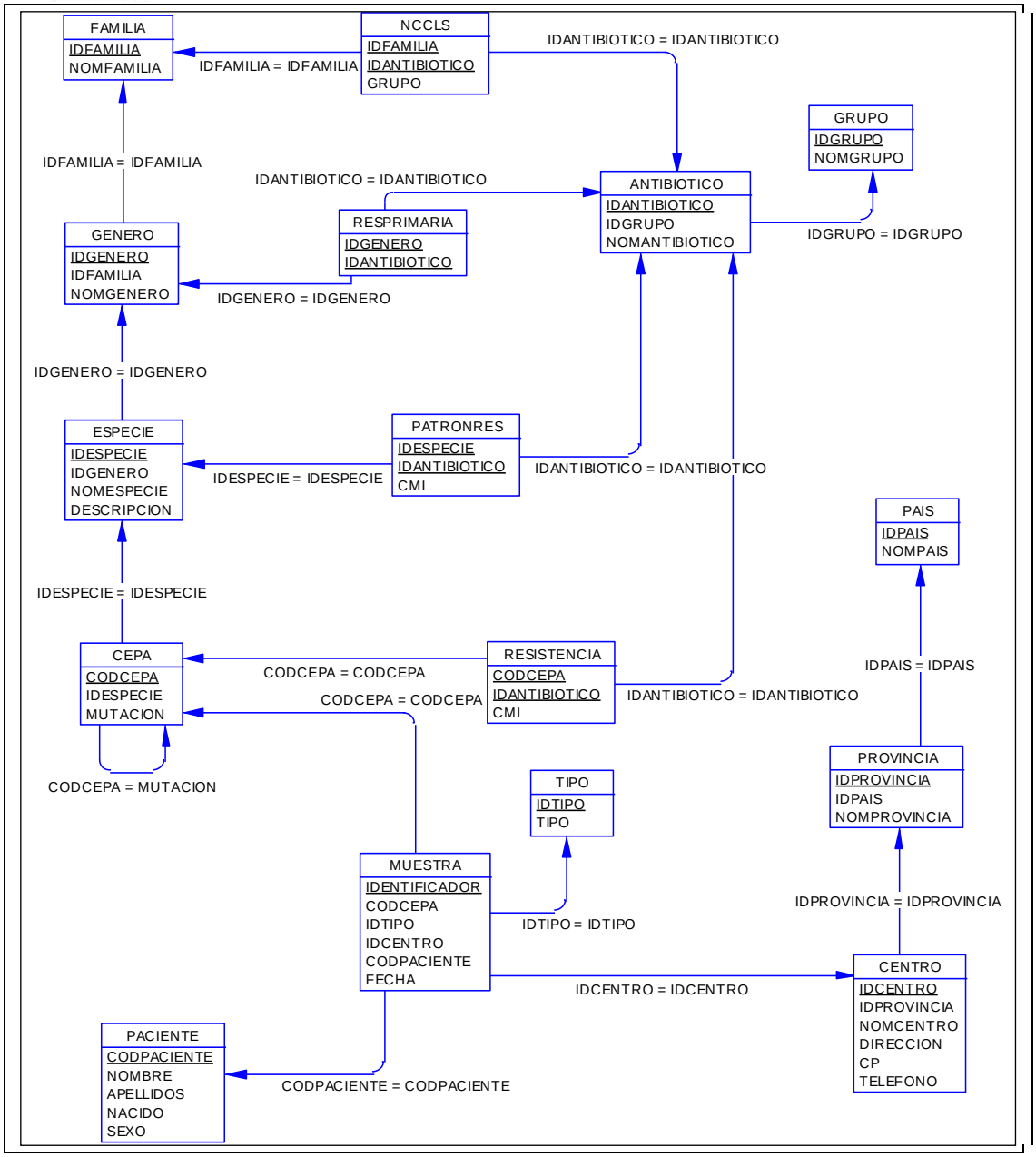
Aunque a nivel conceptual esta era quizás una de las soluciones más fieles al problema original en la práctica el problema de las múltiples claves compuestas se puede traducir en un alto grado de ineficiencia así como en una significativa complicación del código de cualquier aplicativo que opere sobre la base de datos (incluyendo a los propios procedimientos almacenados y triggers de la base de datos).

Por suerte este problema que suele aparecer con relativa frecuencia es fácilmente subsanable. Para ello se utiliza lo que se denomina la figura de un subrogado. Un subrogado es un atributo simple, generalmente numérico y sin significado propio alguno que se usa para identificar unívocamente a cada entidad y que actúa como clave primaria. Su uso está tan generalizado que constituye uno de los heurísticos más utilizados. Y la mayoría de los gestores de bases de datos relacionales incorporan algún mecanismo para su implementación (tipos autonumerados, serial o secuencias). El heurístico en cuestión nos indica que debemos intentar mantener siempre las claves primarias simples y generalmente de tipo numérico.

Si replanteamos nuestro esquema inicial siguiendo este heurístico e intentando evitar las secuencias de entidades débiles obtendremos la siguiente solución:



Esta solución ahora nos produce un esquema relacional más sencillo que el anterior, ya que las restricciones de integridad referencial operan exclusivamente con atributos simples. Produciendo una solución significativamente más sencilla.



La cual se traduce en un código SQL menor y que será gestionado, casi con total probabilidad de forma más eficiente por el gestor de base de datos. Dicho código puede verse en el Apéndice E.

6 SQL: EL ESTÁNDAR DE LAS BASES DE DATOS RELACIONALES

El lenguaje SQL es una herramienta para organizar, gestionar y recuperar datos almacenados en una base de datos. El nombre "SQL" es una abreviación de Structured Query Language (Lenguaje Estructurado de Consultas). Como su nombre indica, SQL es un lenguaje informático que se utiliza para interactuar con una base de datos relacional.

Cuando es necesario obtener datos de una base de datos, la petición se realiza utilizando SQL. El Sistema de Gestión de Base de Datos (SGBD) procesa la petición SQL, recoge los datos solicitados y los devuelve a quien los solicitó.

SQL se utiliza para controlar todas las funciones que suministra un SGBD a sus usuarios, incluyendo:

- Definición de la estructura y de la organización de los datos almacenados.
- Recuperación de datos almacenados en la base de datos.
- Manipulación de datos, añadiendo datos nuevos, borrando datos viejos o modificando los almacenados.
- Control de acceso, protegiendo los datos almacenados contra accesos no autorizados.
- Integridad de datos, protegiéndolos de alteraciones por actualizaciones inconsistentes o fallos del sistema.

Las sentencias SQL se clasifican según su finalidad dando origen a tres 'lenguajes' o mejor dicho sublenguajes:

- el DDL (Data Description Language), lenguaje de definición de datos, incluye órdenes para definir, modificar o borrar las tablas en las que se almacenan los datos y de las relaciones entre estas.
- el DCL (Data Control Language), lenguaje de control de datos, contiene elementos útiles para trabajar en un entorno multiusuario, en el que es importante la protección de los datos, la seguridad de las tablas y el establecimiento de restricciones en el acceso, así como elementos para coordinar la compartición de datos por parte de usuarios concurrentes, asegurando que no interfieren unos con otros.
- el DML (Data Manipulation Language), lenguaje de manipulación de datos, nos permite recuperar los datos almacenados en la base de datos y también incluye órdenes para permitir al usuario actualizar la base de datos añadiendo nuevos datos, suprimiendo datos antiguos o modificando datos previamente almacenados.

SQL se ha convertido en el lenguaje estándar de utilización de bases de datos relacionales. El objetivo de este curso es enseñar al alumno los fundamentos y conceptos básicos del lenguaje SQL, tanto de las operaciones de consultas como las de actualización. Así mismo se le muestra de forma elemental el proceso de creación de tablas en una base de datos relacional.

6.1 Definición de datos y restricciones en SQL

La tabla es la unidad básica de almacenamiento, compuesta por filas y columnas.

En esta sección aprenderemos como insertar tuplas en una tabla, como modificar tuplas ya existentes y como eliminar tuplas de una tabla. El lenguaje de manipulación de datos (DML) forma parte del lenguaje SQL. DML se utiliza para añadir, modificar o borrar datos en la base de datos.

Creación y gestión de tablas

Al final de este apartado debemos haber aprendido a crear tablas, a definir el tipo de datos de las columnas de la tabla, a modificar la definición de una tabla y a borrar una tabla.

```
CREATE TABLE tabla
       (columna tipo_de_datos );
```

Para crear tablas utilizamos la sentencia CREATE TABLE, especificando el nombre de la tabla. Para cada columna debemos especificar su nombre, su tipo de datos y su tamaño. Los tipos de datos permitidos en Oracle se resumen en la siguiente tabla.

Tipo de Datos	Descripción
VARCHAR2(size)	Cadena de caracteres de longitud variable. Se debe especificar un tamaño máximo. El tamaño mínimo por defecto es 1 y el máximo 4000.
CHAR(size)	Cadena de caracteres de longitud fija. El tamaño mínimo por defecto es 1 y el máximo 2000.
NUMBER(p,s)	Número de precisión p y escala s; la precisión es el número total de dígitos decimales, y la escala es el número de dígitos a la derecha de la coma decimal. La precisión puede ir entre 1 y 38y la escala entre –84 y 127.
DATE	Fecha y hora entre el 1 de enero de 4712 A.C. y el 31 de diciembre de 9999 D.C.
LONG	Cadena de caracteres de longitud variable hasta 2 gigabytes.

VER MANUAL DE ORACLE PARA MAS INFORMACIÓN SOBRE LOS TIPOS DE DATOS.

Ejemplo de creación de una tabla

```
SQL> CREATE TABLE departamento
     (numdept       NUMBER(2),
     nombred        VARCHAR2(14),
     localizacion   VARCHAR2(13));
```

Table created.

>Para confirmar la creación de la tabla usamos la sentencia DESCRIBE, que nos devuelve la descripción de la tabla que se le pasa como parámetro.

```
SQL> DESCRIBE departamento;
```

Con la sentencia **ALTER TABLE** podemos modificar la definición de una tabla que ya existe. Mediante esta sentencia podemos añadir o borrar columnas de una tabla o modificar una columna ya existente.

```
ALTER TABLE table
ADD/MODIFY      (columna tipo_de_datos
                [,columna tipo_de_datos]...);
```

Finalmente, para borrar una tabla utilizamos la sentencia DROP TABLE. Con esta sentencia se elimina la tabla completa. Para borrar los datos de la tabla sin borrar la tabla, debemos utilizar la sentencia DELETE (sección 1.2).

Ejemplo de modificación de una tabla

```
ALTER TABLE departamento
       MODIFY (nombred   VARCHAR2(15));


Table altered.
```

Ejemplo de borrado de una tabla

```
SQL> DROP TABLE departamento;

Table dropped.
```

Restricciones

Para más información ver Oracle8 Server SQL Reference, Release 8.0, "CONSTRAINT Clause.

http://www-rohan.sdsu.edu/doc/oracle/server803/A54647_01/ch4e.htm#8703

Las restricciones son condiciones que tienen que cumplir los valores de las columnas de la tabla. Las restricciones válidas en Oracle son las siguientes.

NOT NULL	No se permiten valores nulos en la columna
UNIQUE Key	No se permiten valores repetidos en la columna
PRIMARY KEY	Restricción de clave primaria
FOREIGN KEY	Restricción de clave externa
CHECK	Define una condición que deben cumplir los

	valores de la columna

Ejemplo de definición de restricciones para una tabla

```
SQL> CREATE TABLE empleado
      numemp            NUMBER(4),
      nombre            VARCHAR2(10) NOT NULL,
      trabajo           VARCHAR2(9),
      fecha_nac         DATE,
      salario           NUMBER(7,2),
      numdept           NUMBER(7,2) NOT NULL,
      director          NUMBER(4),
      CONSTRAINT emp_trab_uk UNIQUE,
      CONSTRAINT emp_numemp_ck
      CHECK (DEPTNO BETWEEN 10 AND 99),...
      CONSTRAINT emp_numemp_pk PRIMARY KEY,
      CONSTRAINT emp_numdept_fk FOREIGN KEY (numdept)
```

Un valor nulo no es lo mismo que un 0 o un espacio en blanco. Un valor nulo es un valor desconocido.

6.2 Sentencias insert, delete, y update en SQL

Una vez creadas las tablas, debemos insertar datos en ellas. En este apartado aprenderemos a insertar tuplas en una tabla, a modificar tuplas ya existentes y a borrar tuplas de una tabla.

Para añadir nuevas tuplas a una tabla utilizamos la sentencia INSERT.

```
INSERT INTO tabla [ (columna [,columna…])]
VALUES              (valor [,valor…]);
```

Para insertar una tupla, debemos dar los valores para cada columna, que se deben listar en el orden que se encuentran en la tabla. Opcionalmente podemos listar las columnas en la cláusula INSERT. Los valores tipo carácter y fecha deben ir entre comillas simples.

Ejemplo de inserción de una tupla

```
SQL>  INSERT INTO departamento
      VALUES  (50, 'DESARROLLO', ´MALAGA');

SQL> INSERT INTO departamento (nombred, numdept, localizacion)
VALUES  ('DESARROLLO', 50, ´MALAGA');
```

Es posible que cuando insertemos una tupla en una tabla, desconozcamos el valor de una columna, o simplemente, no deseamos especificarlo. Si no hay definida una restricción NOT NULL para dicha columna, es posible no especificar un valor para la misma. Para indicar este hecho en la sentencia INSERT, o bien omitimos el nombre de la columna al listarlas en la cláusula INSERT, o bien especificamos explícitamente que el valor para esa columna es NULL.

Para modificar tuplas ya existentes en una tabla utilizamos la sentencia **UPDATE**. Si es necesario se modifica más de una tupla.

```
UPDATE  tabla
SET   (columna = VALOR [, columna =VALOR]]
[WHERE condicion]
```

Las tuplas a modificar se indican mediante una condición en la cláusula WHERE. Si no se especifica una cláusula WHERE se modifican todas las tuplas de la tabla.

Ejemplo de modificación de tuplas en una tabla

```
SQL> UPDATE  departamento
SET   numdept = 20;
SQL> UPDATE  empleado
SET   salario = 1000
WHERE salario < 1000;
```

Para borrar tuplas de una tabla utilizamos la sentencia **DELETE**. Si es necesario se modifica más de una tupla.

```
DELETE [FROM]  tabla
[WHERE condicion]
```

Las tuplas a borrar se indican mediante una condición en la cláusula WHERE. Si no se especifica una cláusula WHERE se borran todas las tuplas de la tabla.

Ejemplo de eliminación de tuplas en una tabla

```
SQL> DELETE FROM departamento
       WHERE nombred = 'DESARROLLO';
SQL> DELETE FROM empleado;
```

Hay que tener en cuenta las restricciones para insertar, modificar y borrar tuplas de una tabla. Si estas operaciones violan alguna restricción, la sentencia no se ejecuta.

6.3 Consultas básicas en SQL

La sentencia de selección básica es la sentencia SELECT.

Para recuperar datos de la base de datos, una vez creadas las tablas e insertados los datos, también utilizamos el lenguaje SQL. Este apartado describe todas las sentencias SQL que son necesarias para realizar esta tarea. Al final de este apartado debemos haber aprendido las distintas posibilidades de la sentencia SELECT y a ejecutar sentencias SELECT básicas.

Mediante la sentencia SELECT podemos hacer lo siguiente:

Selección: selecciona las tuplas que queremos que devuelva la consulta. Se pueden usar varios criterios para restringir las tuplas que se visualizarán.

Proyección: selecciona las columnas de una tabla que queremos que devuelva la consulta.

Composición: selecciona datos que están almacenados en diferentes tablas pero que están relacionadas mediante una columna (restricción de clave externa), recuperándolos de forma conjunta.

La sintaxis de la sentencia SELECT básica es

```
SELECT [DISTINCT] {*, column [alias], …}
FROM tabla ;
```

Debemos especificar las columnas que queremos recuperar. Si queremos recuperar todas las columnas, utilizamos *. En la cláusula FROM especificamos la tabla que vamos a consultar. Si queremos eliminar los resultados duplicados, debemos indicarlo utilizando DISTINCT en la cláusula SELECT. También podemos especificar un alias para la columna, que es el nombre con el que se visualizará dicha columna.

Ejemplo de sentencia **SELECT**

```
SQL> SELECT * FROM departamento;
SQL> SELECT numdept, nombred
       FROM departamento;
```

Filtrado y Ordenación

La sentencia de selección básica recupera todas las tuplas de la tabla especificada en la cláusula FROM. Para filtrar aquellas tuplas que cumplan una determinada condición, debemos especificar ésta en la cláusula **WHERE**.

```
SELECT [DISTINCT] {*, column [alias], …}
FROM tabla
[ WHERE condicion(es) ];
```

Las cadenas de caracteres y las fechas deben ir encerradas entre comillas simples.

Ejemplo de utilización de la cláusula WHERE

Recuperar el número de empleado y el salario del empleado 'Jose Perez'

```
SQL> SELECT numemp, salario
     FROM empleado
     WHERE nombre = 'Jose Perez'
```

Para especificar condiciones, podemos usar operadores aritméticos. Los operadores que podemos usar son =, >, >=, <, <=, <> (distinto).

Ejemplo de uso de operadores aritméticos en la cláusula WHERE

Recuperar el numero de empleado y el salario de los empleados cuyo salario sea mayor que 1000

```
SQL> SELECT numemp, salario
     2 FROM empleado
     3 WHERE salario > 1000
```

Podemos especificar otros operadores de comparación más complejos para especificar condiciones en la cláusula WHERE

Operador	Significado
BETWEEN...AND...	Entre dos valores (incluidos)
IN (lista)	Coincide con un valor de la lista
LIKE	Coincide con un patrón de caracteres
IS NULL	Es un valor nulo

El operador **BETWEEN...AND...** se usa para mostrar tuplas basadas en un rango de valores. El operador **IN** se usa para comparar un valor con los de una lista. El operador **LIKE** se utiliza para realizar búsquedas en cadenas de caracteres. **%** indica uno o más caracteres, _ indica un carácter. El operador **IS NULL** comprueba que un valor es nulo

Ejemplo de uso de operadores de comparación complejos

```
WHERE salario BETWEEN 1000 AND 1500;
WHERE numemp IN (7902, 7566, 7788);
WHERE nombre LIKE 'S%'; (nombres que empiezan por S)
WHERE trabajo IS NULL;
```

Nota importante: Valores nulos en las expresiones aritméticas.

> Las expresiones aritméticas que contienen un valor nulo se evalúan como nulas. En Oracle, existe la función NVL(columna, valor), que asigna el valor especificado a la columna si el valor de la columna es nulo.

Para especificar más de una condición, se utilizan los operadores lógicos **AND**, **OR** y **NOT**.

Ejemplo de uso de operadores lógicos

```
WHERE  sal>=1100 AND trabajo='DEPENDIENTE';
WHERE  sal>=1100 OR trabajo='DEPENDIENTE';
WHERE  trabajo NOT IN ('DEPENDIENTE','DIRECTOR,'ANALISTA');
```

> La sintaxis de los operadores aritméticos y de comparación depende del Sistema Gestor de Bases de Datos que se utilice. La sintaxis especificada en este apartado corresponde a la del SGBD Oracle. Para consultar la sintaxis de otros SGBD consultar los manuales correspondientes.

Resumen: Hasta el momento hemos aprendido:

a) a utilizar la sentencia de selección básica
b) a especificar las columnas que queremos recuperar, y
c) a filtrar las tuplas que queremos recuperar

El siguiente paso es especificar si queremos algún tipo de ordenación de las tuplas recuperadas. Para clasificar las tuplas recuperadas utilizamos la cláusula **ORDER BY**. Esta cláusula se añade al final de la sentencia **SELECT**. Debemos especificar la columna por la que queremos ordenar. Por defecto, las tuplas se ordenan por orden ascendente de los valores de la columna de ordenación. También podemos especificarlo explícitamente con 'ASC'. Si queremos ordenar por orden descendente, debemos especificarlo con **'DESC'**. Se pueden especificar más de una columna de ordenación.

```
SELECT [DISTINCT] {*, column [alias], …}
FROM tabla
[WHERE condicion(es) ]
[ORDER BY {column, expr} [ASC/DESC] ];
```

Ejemplo de uso de ORDER BY

```
SQL> SELECT  nombre, trabajo, numdept, salario
     FROM empleado
     ORDER BY salario;
SQL> SELECT  nombre, trabajo, numdept, salario
     FROM empleado
     ORDER BY salario DESC;
```

6.4 Consultas SQL más complejas

Algunas veces es necesario obtener datos de más de una tabla. Al final del apartado debemos haber aprendido a recuperar datos de múltiples tablas utilizando los distintos mecanismos que nos ofrece SQL para ello.

Definición: Composición: Cuando queremos recuperar datos de más de una tabla, utilizamos una condición de composición. Las tuplas de una tabla se componen con las tuplas de otra tabla mediante los valores comunes que existen en las columnas correspondientes, es decir, las columnas para las que se ha definido una clave primaria o una clave externa.

Para recuperar datos de dos o más tablas que están relacionadas, especificamos la condición de composición en la cláusula **WHERE**.

```
SELECT tabla1.columna, tabla2.columna
FROM   tabla1, tabla2
WHERE  tabla1.columna1 = tabla2.columna2;
```

Se precede el nombre de la columna con el nombre de la tabla cuando el mismo nombre de columna aparece en más de una tabla. table.column indica la tabla y la columna de la cual se van a recuperar los datos.

table1.column1 = table2.column2 es la condición que relaciona las dos tablas.

Definición: Producto cartesiano: Si no se especifica ninguna condición de composición, o la condición especificada no es válida, el resultado es el producto cartesiano de las dos tablas. Todas las tuplas de la primera tabla se componen con todas las columnas de la segunda tabla.

El producto cartesiano de dos tablas produce una gran cantidad de tuplas que normalmente no son útiles. Por tanto, siempre hay que especificar una condición de composición cuando componemos dos tablas.

Tipos de Composiciones

Existen distintos tipos de composición:

- **Composición por igualdad:** La condición de composición es una igualdad.
- **θ-Composición:** La condición de composición no es una igualdad.
- **Composición externa:** Se utiliza para recuperar tuplas que no se recuperan normalmente con la composición ya que no hay valores en una de las columnas que coincidan con un valor dado de la otra columna.
- **Composición natural o uniforme:** Cuando las dos columnas que intervienen en la condición de composición contienen exactamente los mismo valores.

Finalmente, podemos componer una tabla consigo misma.

Para simplificar las consultas podemos utilizar alias para las tablas. Por ejemplo, un alias para la tabla empleado podría ser e, y un alias para la tabla departamento podría ser d.

Ejemplo de Composición por igualdad: Seleccionar para cada empleado el número de empleado, nombre y el número del departamento para el que trabaja

```
SQL> SELECT empleado.numemp, empleado.nombre, empleado.numdept
     FROM   empleado, departamento
     WHERE  empleado.numdept=departamento.numdept;
```

Supongamos que hemos creado en nuestra base de datos una tabla llamada gradosalario que adjudica un grado (entre 1 y 5) a distintos rangos de salarios.

```
SQL> CREATE TABLE gradosalario
     ( grado NUMBER,
       salinf NUMBER,
       salsup NUMBER));
```

Table created.

Grado	salinf	salsup
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

Ejemplo de Composición por igualdad: Seleccionar para cada empleado su nombre, su salario y el grado que corresponde a su salario

```
SQL>   SELECT   e.nombre, e.salario, s.grado
       FROM     empleado e, gradosalario g
       WHERE    e.salario
       BETWEEN  g.salinf AND g.salsu
```

Ejemplo de θ - Composición: Seleccionar el nombre de los empleados y el número y nombre del departamento para el que trabajan, ordenados por numero de departamento. Listar también aquellos departamentos para los que no trabaja ningún empleado.

```
SQL> SELECT  e.nombre, d.numdept, d.nombred
     FROM    empleado e, departamento d
     WHERE   e.numdept (+) = d.numdept;
     ORDER BY e.numdept;
```

Ejemplo de Composición de una tabla consigo misma: Listar los empleados y sus directores

```
SQL> SELECT trabajador.nombre, dir.nombre
     FROM   empleado.trabajador, empleado dir
     WHERE  trabajador.director = dir.numemp;
```

Utilizamos los alias trabajador y dir para la tabla empleado de forma que podamos componerla consigo misma.

6.5 Subconsultas

En este apartado aprenderemos una característica avanzada de la sentencia **SELECT**. Podemos escribir subconsultas en la cláusula **WHERE** de otra sentencia SQL para obtener valores basados en una valor condicional desconocido. Al final de este apartado debemos haber aprendido a utilizar los distintos tipos de subconsultas que nos ofrece SQL.

```
SELECT    lista_selección
FROM      tabla
WHERE     expr operador
              ( SELECT  lista_selección
              FROM      tabla);
```

La subconsulta (consulta interior) se ejecuta una vez antes de la consulta principal. El resultado de la subconsulta es usado para la consulta principal (consulta exterior).

Las subconsultas también pueden utilizarse en la cláusula **HAVING**.

Ejemplo de subconsulta: seleccionar el nombre de los empleados cuyo salario es mayor que el salario del empleado cuyo número es el 7566.

```
SQL> SELECT nombre
     FROM   empleado
     WHERE  salario>
              (SELECT salario
              FROM   empleado
              WHERE  numemp=7566);
```

La subconsulta nos devuelve el salario del empleado cuyo número es 7566, y este es el dato que utilizamos en la expresión aritmética de la cláusula **WHERE** para comparar con el salario del resto de los empleados.

Tipos de Subconsultas

- Subconsulta "de una tupla": Devuelven una única tupla. Se usan con operadores aritméticos de una tupla, como **=, <, >,** etc.
- Subconsulta "de múltiples tuplas": Devuelven más de una tupla. Se usan con operadores aritméticos de múltiples tuplas, como IN, ANY o ALL.

Ejemplo de subconsulta “de múltiples tuplas”: seleccionar el número de empleado, el nombre y el trabajo de los empleados cuyo salario es mayor que el salario de cualquier dependiente y no son dependientes.

```
SQL> SELECT  numemp, nombre, trabajo
     FROM    empleado
     WHERE   salario < ANY
                    (SELECT salario
                    FROM      empleado
                    WHERE   trabajo = 'DEPENDIENTE')
     AND      trabajo <> 'DEPENDIENTE';
```

Subconsultas correlacionadas

Las subconsultas correlacionadas se utilizan para procesamiento tupla a tupla. La subconsulta se evalúa una vez por cada tupla de la consulta externa.

```
SELECT exterior1, exterior2, ...
FROM   tabla1 alias1
WHERE  exterior1 operador
                    (SELECT interior1
                  FROM    tabla2 alias2
       WHERE alias1.exterior2 = alias2.interior1);
```

La subconsulta referencia a una columna de una tabla de la consulta padre.

Ejemplo de subconsulta correlacionada: Encontrar todos los empleados que cobran más que el salario medio de sus departamentos.

```
SQL> SELECT numemp, salario, numdept
     FROM   empleado exterior
     WHERE  salario > (SELECT AVG(salario)
                       FROM   empleado interior
                       WHERE  exterior.numdept = interior.numdept);
```

El operador **EXISTS** se utiliza en las subconsultas de la siguiente forma:

- Si se encuentra el valor en una tupla de la subconsulta:
  - La búsqueda no continua en la consulta interna.
  - La condición es puesta a **TRUE**.
- Si no se encuentra el valor en una fila de una subconsulta:
  - La condición es puesta a **FALSE**.
  - La búsqueda continua en la consulta interna.

Ejemplo de subconsulta con operador EXIT: Encontrar todos los departamentos que no tienen ningún empleado.

```
SQL> SELECT  numdept, nombred
     FROM    departamento d
```

```
WHERE NOT EXISTS ( SELECT '1'
                   FROM   empleado e
                   WHERE  d.numdept = e.numdept);
```

Con '1' indicamos que no nos interesa lo que devuelva la subconsulta, únicamente nos interesa si se devuelve o no algún valor.

6.6 Funciones de agregación en SQL

Las funciones de agregación operan sobre un conjunto de tuplas para dar una resultado por cada uno de estos conjuntos.

```
SELECT columna, función_agregación(columna)
FROM         tabla
[WHERE condición]
[ORDER BY columna];
```

La siguiente tabla muestra las funciones de agregación que podemos utilizar en Oracle.

Función	Descripción
AVG([DISTINCT|ALL]*n*)	Media aritmética de los valores de n. Los valores nulos se ignoran.
COUNT **COUNT({*|[DISTINCT|ALL]expr}** **)**	Número de tuplas, donde expr evalúa algo distinto de nulo. Para contar todas las tuplas se usa *, y se incluyen las tuplas duplicadas y con valores nulos.
MAX MAX([DISTINCT|ALL]*expr*)	Valor máximo de expr, ignorando valores nulos.
MIN([DISTINCT|ALL]*expr*)	Valor mínimo de expr, ignorando valores nulos.
STDDEV([DISTINCT|ALL]*x*)	Desviación estándar de n, ignorando valores nulos.
SUM([DISTINCT|ALL]*n*)	Suma de los valores de n, ignorando valores nulos.
VARIANCE([DISTINCT|ALL]*x*)	Varianza de n, ignorando valores nulos.

Ejemplo de uso de funciones de agregación:

```
SQL> SELECT      AVG(salario), MAX(salario),
               MIN(salario), SUM(salario)
     FROM  empleado
     WHERE trabajo LIKE 'VENDE%';
SQL> SELECT MIN(fecha_nac), MAX(fecha_nac)
     FROM   empleado;
SQL> SELECT COUNT(*)
     FROM   empleado
     WHERE  numdept = 30; (cuenta los empleados del departamento 30)
```

Hasta el momento, todas las funciones de agregación tratan las tablas como un único grupo de información. Algunas veces, necesitamos dividir la tabla en pequeños grupos de información. Esto se hace usando la cláusula **GROUP BY**.

```
SELECT       columna, función_agrupación(columna)
FROM         tabla
[WHERE       condición]
[GROUP BY    expresión_de_agrupación]
[ORDER BY    columna];
```

Todas las columnas de la lista **SELECT** que no están en funciones de agregación deben estar en la cláusula **GROUP BY**. La columna de **GROUP BY** no tiene por que estar en la lista **SELECT**. Se puede usar la cláusula **GROUP BY** para múltiples columnas.

Ejemplo de uso de la cláusula GROUP BY

```
SQL> SELECT   numdept, AVG(salario)
     FROM     empleado
     GROUP BY numdept;
SQL> SELECT   AVG(salario)
     FROM     empleado
SQL> SELECT   numdept, trabajo, sum(salario)
     FROM     empleadp
     GROUP BY numdept, trabajo;
```

De la misma manera que utilizamos la cláusula **WHERE** para restringir las tuplas que queremos seleccionar, se usa la cláusula **HAVING** para restringir los grupos.

```
SELECT         columna, función_agrupación(columna)
FROM         tabla
[WHERE         condición]
[GROUP BY      expresión_de_agrupación]
[HAVING        condición_agrupación]
[ORDER BY      columna];
```

Las filas son agrupadas. Se aplica la función de grupo. Se muestran los grupos que pasan la cláusula **HAVING**.

Ejemplo de uso de la cláusula HAVING: Encontrar el salario máximo para cada departamento, pero mostrar solo los departamentos que tienen un salario máximo de más de 2999$

```
SQL> SELECT   numdept, max(salario)
     FROM     empleado
     GROUP BY numdept
     HAVING   max(salario)>2900;
```

6.7 Vistas (tablas virtuales) en SQL

¿Qué es una vista?

Una vista es una tabla virtual basada en otra tabla u otra vista. Una vista no contiene datos por sí misma, pero es como una ventana a través de la cual se pueden ver y modificar los datos de otras tablas.

Las vistas se usan para:

- restringir el acceso a la base de datos
- hacer fácil consultas complicadas
- dar independencia a los datos
- visualizar los datos de distintas formas

Para crear una vista utilizamos la sentencia **CREATE VIEW**, en la que incluimos una subconsulta.

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW vista
       [ (alias [,alias]…)]
AS subconsulta
[WITH CHECK OPTION [CONSTRAINT restriction]]
[WITH READ ONLY]
```

La subconsulta puede contener un **SELECT** con sintaxis compleja. La subconsulta no puede contener una cláusula **ORDER BY**.

Ejemplo de creación de una vista: Crear una vista, **EMPVU10**, que contenga detalles de los empleados del departamento 10.

```
SQL> CREATE VIEW empvu10
     AS SELECT numemp, nombre, trabajo
     FROM empleado
     WHERE numdept = 10;
```

Las vistas se utilizan para evitar que consultas muy complejas (por ejemplo, composiciones sobre múltiples tablas) se evalúen más de una vez. Si tenemos conocimiento de qué consultas se realizan frecuentemente, podemos materializar el resultado de la consulta mediante una vista. De esta forma, la siguiente vez que se evalúe la consulta se accederá a la vista como si fuese una tabla, evitando el coste de componer las tablas una y otra vez.

Para borrar una vista utilizamos la sentencia DROP VIEW. Esta sentencia elimina la vista, pero no se eliminan los datos de la base de datos, ya que los datos están realmente almacenados en las tablas sobre las que se construye la vista.

No siempre podemos insertar o modificar datos en una vista. Dependerá de si estas inserciones o modificaciones pueden realizarse también en las tablas sobre las que se construye la vista o no.

Tiene una extensa relación de ejercicios en el Apéndice F.

7 SQL INMERSO. PROGRAMACIÓN EN PL/SQL

SQL inmerso o embebido permite a los programadores conectar con una base de datos e incluir código SQL en su programa, y poder usar, manipular y procesar datos de la base de datos.
El conjunto de instrucciones formado por sentencias del lenguaje elegido y SQL es compilado por el preprocesador SQL que genera un código con unas librerías especiales mediante el cual podemos acceder a las bases de datos.

7.1 ¿Qué es PL/SQL?

PL/SQL es la abreviatura de 'Procedural Language /SQL'. Es un lenguaje de programación que proporciona Oracle para extender el SQL estándar, añadiéndole a éste estructuras habituales en otros lenguajes como:

- variables y tipos (predefinidos y definidos por el usuario)
- estructuras de control (bucles y condicionales)
- procedimientos y funciones
- tipos de objetos y métodos

Además PL/SQL nos permite capturar los errores de ejecución mediante el manejo de excepciones, la encapsulación de datos, la ocultación de la información y la sobrecarga.

Algunas de las características de este lenguaje son:

- sintaxis basada en ADA
- portabilidad
- fácil manejo
- más potente que SQL
- permite programar operaciones complejas dentro de un servidor de base de datos, lo que implica que los bloques de PL/SQL son procesados por el servidor
- disponible para todo entorno de desarrollo Oracle
- esta estructurado en bloques y se permiten la anidación de estos
- lenguaje muy limitado si lo comparamos con otros lenguajes de propósito general

7.2 ¿Para qué sirve PL/SQL?

Esta pensado principalmente para programar triggers y procedimientos almacenados, fragmentos de código que se ejecutan al ocurrir un evento especifico.

Concretando aún mas podemos añadir que es utilizado como:

- procedimientos ejecutados desde aplicaciones 4GL.
- bloques enviados desde aplicaciones programadas con 3GL.
- programas almacenados y compilados en la BD e invocados desde aplicaciones.
- invocados desde disparadores definidos a nivel de BD, activados por operaciones DML sobre una tabla.

Aclaraciones:

3GL: Third Generation Language. Los lenguajes de tercera generación son aquellos lenguajes de programación utilizados por los especialistas para construir aplicaciones que incluyen el procedimiento, es decir, el programador especifica en su programa qué tiene que hacer el ordenador y cómo debe hacerlo, se trata de un paso más allá del lenguaje máquina. Son lenguajes de tercera generación Cobol, C, Pascal o Fortran.

4GL: Four Generation Language. Es un lenguaje no procedural, el usuario define qué se debe hacer, no cómo debe hacerse. Los 4GL se apoyan en unas herramientas de más alto nivel denominadas herramientas de cuarta generación. El usuario no debe definir los pasos a seguir en un programa para realizar una determinada tarea, tan sólo debe definir una serie de parámetros que estas herramientas utilizarán para generar un programa de aplicación. Los 4GL pueden mejorar la productividad de los programadores en un factor de 10, aunque se limita el tipo de problemas que pueden resolver. Los 4GL abarcan:
Lenguajes de presentación, como lenguajes de consultas y generadores de informes.
Lenguajes especializados, como hojas de cálculo y lenguajes de bases de datos.
Generadores de aplicaciones que definen, insertan, actualizan y obtienen datos de la base de datos.
Lenguajes de muy alto nivel que se utilizan para generar el código de la aplicación.
Los lenguajes SQL y QBE son ejemplos de 4GL.

DML: Data Manipulation Language. Es el conjunto de sentencias que están orientadas a la consulta, y manejo de datos de los objetos creados.
El DML es un subconjunto muy pequeño dentro de SQL, pero es el más importante, ya que su conocimiento y manejo con soltura es imprescindible, consta de 4 sentencias: select, insert, delete, update.

7.3 Creación de programas PL/SQL

7.3.1 Estructura Básica: el bloque.

Cuando se escribe código en PL/SQL este debe estar agrupado en unidades denominadas "Bloques de código", por lo tanto podemos decir que **la unidad básica es el bloque**. Normalmente, cada bloque realiza una unidad lógica de trabajo en el programa, dividiendo unas tareas de otras.

Podemos crear diferentes tipos de bloques:

- **Bloques anónimos:** Se construyen de forma dinámica y se suelen ejecutar una sola vez.
- **Bloques nominados:** Igual que los anónimos pero con una etiqueta que les da nombre.
- **Subprogramas:** Procedimientos, paquetes y funciones, almacenados en la base de datos y que se ejecutan en múltiples ocasiones. Los subprogramas se ejecutarán mediante una llamada.
- **Disparadores("Triggers"):** Bloques nominados que se almacenan en la base de datos y se ejecutan ante algún suceso.

7.3.2 Anidación de bloques

PL/SQL nos permite anidar bloques, es decir, los bloques pueden contener otros bloques. Esto se puede llevar a cabo en la sección de ejecución y en la sección de excepciones.

Veamos el ámbito de las declaraciones definidas en el caso de anidación de bloques:

- Todas las constantes, variables y excepciones definidas en el bloque externo son visibles en el bloque interno, pero no al revés.
- Si existiera conflicto de nombres entre las declaraciones del bloque externo e interno, en cada bloque se toma la declaración local.

El bloque interno se considera como una única instrucción del bloque externo, que finaliza como cualquier otro bloque PL/SQL, es decir, por la finalización de su parte de instrucciones o por la activación y gestión de una excepción, si fuera el caso.

```
BEGIN
      Sentencias;
      Sentencias;
      Sentencias;
      BEGIN
            Sentencias;
            Sentencias;
            Sentencias;
      END;
      Sentencias;
      Sentencias;
      Sentencias;
END;
```

7.3.3 Estructura de un bloque

La estructura de un bloque es la siguiente:

```
DECLARE
      /*Sección de declaraciones*/
BEGIN
      /*Sección de Ejecución*/
EXCEPTION
      /*Sección de Excepciones*/
END;
```

Tenemos 3 secciones, cada una de ellas delimitada por una palabra reservada.
La primera **sección** es la de **declaraciones** y en ella definimos constantes, variables, tipos estructurados y subprogramas locales que se utilizarán en la sección de ejecución y en la sección de excepciones.
La segunda es la **sección de ejecución** que incluye las instrucciones a ejecutar en el bloque PL/SQL.
Y por último, la **sección de excepciones**, donde se definen los manejadores de errores de ese bloque. Utilizando esta sección se separa el código de gestión de errores del cuerpo principal del programa, con lo que se consigue que la estructura de este sea más clara.

De las 3 secciones la única obligatoria es la de ejecución, y debe tener por lo menos una orden, por lo tanto podemos tener bloques en los que no exista la sección de declaraciones ni la de excepciones.

```
BEGIN
NULL;
      -- Al menos una sentencia es obligatoria. NULL es una
sentencia que no hace nada
END;
```

Podemos darle nombre a un bloque, esto se consigue poniéndole una <<etiqueta>> delante de **DECLARE**, los símbolos << y >> también deben escribirse.

```
<<nombre_del_bloque>>

BEGIN
Sentencias;
Sentencias;
END;
```

Veamos cada una de las secciones:

Sección de declaraciones

En esta sección se declaran todos los tipos de datos, las constantes y variables utilizadas en el bloque de ejecución. También se declaran cursores, de gran utilidad para la consulta de datos, excepciones definidas por el usuario y subprogramas locales.

La forma de hacer referencia a variables y constantes es mediante un identificador que ha sido definido en la zona de declaraciones, pues bien, estos identificadores deben de guardar una sintaxis.

La sintaxis es la siguiente: los identificadores válidos empiezan por una letra que puede ser seguida de una secuencia de caracteres que puede incluir letras, números, $, _,#, pero no puede ser una palabra reservada de PL/SQL. La longitud máxima de un identificador es de 30 caracteres.
Es conveniente y recomendable que el nombre de una variable comience por la letra v, el de una constante por c, el de un tipo por t, el de un cursor por cur, y así sucesivamente.

Hay que destacar que PL/SQL no hace distinción entre mayúsculas y minúsculas.

Variables

Podemos declarar variables, cada una debe tener su tipo asociado.

Los tipos posibles son:

Tipos escalares: NUMBER, DATE, CHAR, VARCHAR, BOOLEAN, BINARY_INTEGER.

BINARY_INTEGER	Entero de 32 bits. Subtipos: NATURAL (0..max) y POSITIVE (1..max)
CHAR(n)	Cadena de longitud fija (n<=32767), por defecto 1 byte. Subtipos: STRING y CHARACTER.
DATE	Fecha (año+mes+dia+hora+minuto+segundo).
NUMBER(p,s)	Número (precisión p dígitos, escala 10-s). Subtipos: DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INT, INTEGER,NUMERIC, REAL y SMALLINT
VARCHAR2(n)	Cadena de longitud variable (n<=32767). Subtipos: VARCHAR

BOOLEAN	Variable lógica, solo puede tomar 3 valores: true, false o null

Tipos especiales: **identificador%TYPE** e **identificador%ROWTYPE**. (estos veremos para que se utilizan más adelante).

El tipo **VARCHAR** es un sinónimo de CHAR, siendo más conveniente la utilización del tipo CHAR.

El tipo **Binary-integer** (**enteros binarios**) se usa para almacenar valores que solo van a ser utilizados en cálculos y no se van a almacenar en la BBDD.

```
DECLARE
      V_NombreEstudiante VARCHAR2(20);
      V_FechaActual DATE;
      V_Puntos Number(3);
      V_contadorbucle   binary_integer;
      V_registrado   boolean;
```

Constantes

También podemos declarar constantes, para ello contamos con la palabra reservada '**CONSTANT**'. La cláusula CONSTANT indica la definición de una constante cuyo valor no puede ser modificado, se debe incluir la inicialización de la constante en su declaración.

Veamos el siguiente ejemplo:

```
c_numero CONSTANT NUMBER(3) := 100;
```

Por lo tanto la **sintaxis** para variables y constantes es la siguiente:

```
nombre_variable [CONSTANT] tipo [NOT NULL][{:=|DEFAULT}valor_inicial];
```

La cláusula NOT NULL impide que se le asigne a una variable un valor nulo, por lo tanto debe inicializarse a un valor diferente de NULL al declararlas. Las variables que no son inicializadas toman el valor inicial NULL. La inicialización puede incluir cualquier expresión legal de PL/SQL, que lógicamente debe corresponder con el tipo del identificador definido.
Para inicializarla se puede utilizar la asignación '**:=**' o la palabra reservada **DEFAULT**.

Variables especiales (%TYPE y %ROWTYPE)

En ciertos casos, nos hará falta definir variables para acceder a los datos contenidos en una tabla, para ello debemos definir las variables del mismo tipo que las columnas de la tabla.

El tipo **%TYPE** nos permite obtener el tipo de una columna o una variable de una tabla, no es necesario conocer cómo está definida esa columna en la tabla y, en caso que la definición de la columna sea modificada, automáticamente se cambia la variable.
Si por el contrario no lo utilizamos podría ocurrir que modificásemos el tipo de una columna, con lo cual tendríamos que revisar todo el código para actualizar el tipo.

```
DECLARE
      Sueldo NUMBER(8);
            --Puede dar lugar a problemas si se cambia la
            --definición de VENDEDOR.SALARIO
…
```

Una forma más correcta será indicar al sistema que esa variable es del mismo tipo que la columna correspondiente sería:

```
DECLARE

      Sueldo VENDEDOR.SALARIO%TYPE;
            --Indicamos el tipo de la columna correspondiente
      OtroSueldo Sueldo%TYPE;
            --Podemos acceder al tipo de las variables.
…
```

El tipo %ROWTYPE nos permite crear un registro (RECORD) que tenga la misma estructura que una tabla.
Los nombres son los mismos en el registro que en la tabla, así, la siguiente declaración sería correcta:

```
DECLARE
      registroVendedores  VENDEDORES%ROWTYPE;
      --Ahora podemos acceder a registroVendedores.Nombre.
…
```

Tablas y Registros

En PL/SQL se admite también el uso de tipos definidos por el usuario como tablas y registros.

Existen dos tipos de datos que no hemos mencionado anteriormente: los registros (o estructuras) y las tablas (o arrays o vectores). Los dos tipos deben ser definidos como un nuevo tipo antes de declarar variables de ese nuevo tipo, el modo de definir nuevos tipos de variables en PL/SQL es a través de la palabra reservada TYPE:

```
      TYPE nuevo_tipo IS tipo_original.
```

Una vez definido en nuevo tipo, ya se pueden definir variables de ese nuevo tipo:

```
Una_variable nuevo_tipo;
```

Registros

Los registros no son más que agrupaciones de tipos de variables que se acceden con el mismo nombre.
La sintaxis de definición de registros es:

```
TYPE tipo_registro IS RECORD(
Campo1 tipo1 [NOT NULL][:= expr1],
Campo2 tipo2 [NOT NULL][:=expr2],
...
CampoN tipoN [NOT NULL][:=exprn]);
```

Por ejemplo:

```
Type alumno IS RECORD (
     n-alumno   VARCHAR2 (5),
     nombre     VARCHAR2(25),
     apellido_1 VARCHAR2(25),
     apellido_2 VARCHAR2(25),
     tlf        VARCHAR2(15));

TYPE tipo_empleado_reg IS RECORD
 (nombre VARCHAR2(10),
  puesto VARCHAR2(8),
  sueldo NUMBER(6));

v_empleado_reg tipo_empleado_reg
```

Para hacer referencia a los campos de un registro se utiliza el punto '.' , nombre_reg.nombre_campo.
Para poder asignar un registro a otro ambos deben ser del mismo tipo.
También se pueden asignar valores a un registro completo mediante la orden SELECT que extraería datos de la BD y los almacenaría en un registro.

Si vamos a utilizar el tipo %ROWTYPE, no hace falta definir un tipo como registro.

```
Declare
V_reg     nombre_tabla%ROWTYPE
```

Tablas

Una tabla no es más que una colección de elementos identificados cada uno de ellos por un índice. En muchos lenguajes se les denomina arrays.
La sintaxis de definición de tablas es:

```
TYPE nombre_tipo IS VARRAY (tamaño_maximo) OF tipo_datos [NOT NULL];
TYPE nombre_tipo IS TABLE OF tipo_datos [NOT NULL];
```

En ambos casos los índices se empiezan a contar a partir de 1, y para poder utilizar los vectores, deben ser previamente creados vacíos o con elementos.

El tamaño de la tabla se define durante la declaración de la variable:

```
Nombre_variable nombre_tabla:= nombre_variable ( lista de elementos);
```

Por ejemplo:

```
DECLARE
     TYPE array_enteros IS TABLE OF INTEGER;
     Un_array array_enteros := array_enteros (0,0,0,0,0);
BEGIN

END;
```

El ejemplo anterior define un tipo de array de enteros y después declara una variable de ese tipo, inicializándola a 5 elementos (todos con 0).
Veamos otro ejemplo:

```
DECLARE

TYPE tipo_varray IS VARRAY (50) OF alumnos.nombre%TYPE;
v_array1 tipo_varray;
v_array2 tipo_varray;

BEGIN
...
v_array1 := tipo_varray('Maria', 'Sonia');
     -- se crea con dos elementos
v_array1.EXTEND;
v_array1(3) := 'Mario';
     -- si a continuación pusiesemos v_varray1(4) := 'Pedro';
     -- Esto sería un error porque no se ha extendido
v_array2 := t_varray(); -- se crea vacío
IF v_array2 IS NULL THEN
v_array2 := v_varray1; --asignación de vectores
....
END;
```

Como podemos ver en el ejemplo, si anteriormente hemos creado una tabla con dos elementos, solo podemos insertar elementos adicionales si extendemos la tabla con la orden EXTEND.

Diferencias entre TABLE y VARRAY:

- Si definimos un vector como TABLE el vector puede crecer ilimitadamente, pero si lo definimos como VARRAY solo puede crecer hasta el tamaño máximo definido.
- En los tipos VARRAY no se pueden borrar elementos, por lo que sus posiciones se deben ocupar consecutivamente. Sin embargo, en los tipos TABLE se pueden borrar elementos con la instrucción DELETE, pudiendo quedar huecos intermedios vacíos y sin poderse referenciar, aunque se pueden volver a llenar con una asignación.

Veamos el siguiente ejemplo:

```
DECLARE
TYPE tipo_table IS TABLE OF alumnos.nombre%TYPE;
v_table1 tipo_table;
v_table2 tipo_table;
BEGIN
...
v_table1 := tipo_table('Maria', 'Sonia');
v_table1(2) := NULL;
     --Dejar una posición vacía no es igual que borrarla
v_table1.DELETE(1);--borramos una posición
v_table1.EXTEND;
v_table1(3) := 'Mario';
     -- si a continuación pusiesemos v_table1(4) := 'Pedro';
     -- Esto sería un error porque no se ha extendido
v_table2 := tipo_table();-- se crea vacía
IF v_table1(1).EXISTS  THEN
...;
ELSE
v_table1(1) := 'Victor'; --se vuelve a crear el elemento 1
END IF;
...
END;
```

Estas tablas tienen una serie de atributos que son:

ATRIBUTO	TIPO DEVUELTO	DESCRIPCIÓN
COUNT	NUMBER	Devuelve el nº de filas de una tabla.
DELETE		Borra las filas de una tabla.

EXISTS	BOOLEAN	Devuelve TRUE si existe en la tabla el elemento especificado.
FIRST	BINARY_INTEGER	Devuelve el índice de la primera fila de la tabla.
LAST	BINARY_INTEGER	
NEXT	BINARY_INTEGER	

Subprogramas locales

Son bloques PL/SQL que se declaran dentro de otro bloque y sólo pueden ser llamados dentro del bloque al que pertenecen. Se declaran al final de la sección declarativa del bloque (veremos como se declaran en el apartado de funciones y procedimientos).

Objetos

Al igual que cualquier otra variable PL/SQL un objeto se declara simplemente incluyéndolo en la sección declarativa del bloque. Desde la versión PL/SQL 8 podemos utilizar objetos, próximamente profundizaremos en este tema.

Sección de ejecución

En esta sección debemos de tener en cuenta que cada sentencia debe terminar con un punto y coma.

Dentro de esta sección podemos distinguir:

- Instrucciones de asignación.
- Estructuras de control de flujo.
- Bucle.
- Instrucciones DML (las veremos en la sección de acceso a datos).

Instrucción de asignación.

Las variables pueden ser manipuladas directamente, asignándoles un valor o consultando el que tienen, o a través de sentencias SQL. El valor es una expresión que debe ser obligatoriamente del mismo tipo que la variable a la que se asigna.

Las instrucciones de asignación tienen como sintaxis:

```
<variable> := <expresión_PL/SQL>;
```

Las expresiones pueden ser constantes, variables definidas en el bloque, cadena de caracteres delimitadas por comillas simples y funciones.
Para crear expresiones podemos hacer uso de distintos operadores:

- Operadores sobre números: `+,-,*,/,**(exponencial), MOD (resto).`
- Operadores sobre cadenas: `||` (concatenación)
- Operadores lógicos: `AND,OR,NOT.`
- Operadores sobre cursores: `%ROWCOUNT, %NOTFOUND, %FOUND, %ISOPEN.`

Los números reales pueden especificarse en formato decimal y científico.
Los valores lógicos aparecen como resultado de comparaciones y verificaciones de valores.
Los operadores de comparación que podemos utilizar son :

- Comparadores clásicos: `<, <=, =, !=, ^=, >=, >.`
- Comparadores SQL: `[NOT] LIKE, IS [NOT] NULL, [NOT] BETWEEN..AND.., [NOT] IN.`

En cuanto a las funciones, podemos hacer uso de todas las definidas en SQL y otras propias del lenguaje:

- Funciones cuyo argumento es un número: `ABS, CEIL, FLOOR, MOD, POWER, ROUND, SIGN, SQRT, TRUNC.`
- Funciones sobre fechas: `ADD_MONTHS, LAST_DAY, MONTHS_BETWEEN, NEW_TIME,NEXT_DAY, ROUND, SYSDATE, TRUNC.`
- Funciones sobre cadenas de caracteres: `ASCII, CHR, INITCAP, INSTR, LENGTH, LOWER,LPAD, LTRIM, REPLACE, RPAD, RTRIM, SOUNDEX, SUBSTR, TRANSLATE, UPPER.`
- Funciones de conversión: `TO_CHAR, TO_DATE, TO_NUMBER.`
- Funciones de control de errores: `SQLCODE, SQLERRM.`
- Funciones varias: `UID, USER, DECODE, GREATEST, LEAST, NVL, USERENV.`

Y funciones concretas creadas por el programador.

Asignaciones asociadas a sentencias SQL:

Existen dos alternativas, asignar el resultado a una lista de variables o a un cursor.
Si utilizamos una **sentencia SELECT** para realizar una asignación a una lista de variables, la consulta asociada debe de dar como resultado una única fila, en caso contrario se generaría una excepción.

El número de variables escalares en la lista de variables debe corresponder con el número de atributos del SELECT.
Para asignar el resultado de una sentencia SELECT a una lista de variables se utiliza la sintaxis:

```
SELECT lista_select INTO lista_variables FROM ... WHERE...;
```

Veamos un ejemplo:

```
DECLARE
v_numdep NUMBER(2);
v_edificio VARCHAR2(15);
BEGIN
SELECT numdep, edificio INTO v_numdep, v_edificio
FROM departamentos
WHERE nombre='Informática'; --Seguro que sólo devuelve
una fila
END;
```

Si no conocemos a priori el número de filas del resultado, debemos de utilizar **cursores** (los veremos más adelante).

Estructura de condición.

La estructura condicional, también denominada alternativa, se basa en evaluar una condición lógica, y dependiendo del resultado de esa evaluación, se elige uno u otro camino de ejecución.
La sintaxis completa de una estructura condicional es la siguiente:

```
IF condición1 THEN
      Secuencia de ordenes1; (instrucciones PL/SQL)
     /*secuencia de ordenes a ejecutar si la condición1 es
       cierta*/
[ELSIF condición2 THEN
      Secuencia de ordenes2; (instrucciones PL/SQL)
      /*secuencia de ordenes a ejecutar si la condición2 es
        cierta ( y la condición1 es falsa)*/
      …
[ELSE
      secuencia de ordenes3;] (instrucciones PL/SQL)
      /*secuencia de ordenes a ejecutar si todas las
       condiciones son falsas*/
END IF;
```

En cada sección del IF podemos poner el número de órdenes que necesitemos.

```
DECLARE
      v_numasientos  habitación.num_asientos%TYPE;
      v_tipo VARCHAR(20);
BEGIN
      SELECT num_asientos
      INTO v_numasientos
      FROM habitaciones
      WHERE num_habitacion=124;
      IF v_numasientos < 50 THEN
            V_tipo:= 'pequeña';
      ELSIF v_numasientos<100 THEN
            V_tipo:='mediana';
      ELSE
            V_tipo:='grande';
      END IF;
END;
```

Estructura de repetición.

Un bucle es una estructura repetitiva, que nos permite ejecutar varias veces una secuencia de órdenes. Existen tres estructuras diferentes.
El bucle básico se implementa con la estructura LOOP ... END LOOP;:

```
LOOP
      Sentencias;
END LOOP;
```

Las sentencias dentro del bucle se ejecutarán un número indefinido de veces y solo finalizará si incluimos una sentencia EXIT. Esta no sería la forma más correcta de salir del bucle, además podríamos tener bucles anidados y dicha instrucción podría ser utilizada para finalizar los demás bucles, este estilo de programación no es aconsejable. Este tipo de estructura se denomina bucle incondicional.
La forma más correcta de salir (aunque equivalente a la anterior) es usar la sentencia `EXIT WHEN condición;` por lo que al bucle básico le añadimos la sentencia `EXIT WHEN condición,` que hará que finalice el bucle cuando se cumpla la condición:

```
LOOP
      Sentencias;
      EXIT WHEN condición;
      Sentencias;
END LOOP;
```

El bucle anterior es equivalente al siguiente:

```
LOOP
      Sentencias;
      IF condicion THEN
            EXIT;
      END IF;
      Sentencias;
END LOOP;
```

Un tipo de bucle más común son los bucles condicionales:

```
WHILE condición LOOP
Sentencias;
END LOOP;
```

Sobre este tipo de bucles debemos indicar que la condición se comprueba siempre *antes* de ejecutar las sentencias del bucle, por lo que si la primera vez no se cumple, ya no entra en el bucle. En los anteriores, las sentencias se ejecutaban al menos una vez.
La tercera estructura que tenemos es el bucle FOR:

```
FOR contador IN [REVERSE] valor_inicial..valor_final LOOP
          Sentencias;
END LOOP;
```

Este tipo de bucles tiene la particularidad de que la variable del bucle, contador, es declarada implícitamente, es decir, no hay que declararla en la sección de declaración del bloque PL/SQL, y es declarada como BYNARY_INTEGER.

Los valores inicial y final pueden ser cualquier expresión que pueda ser convertida a un valor numérico, como una función que devuelve un valor numérico o una constante.
El bucle inicializa la variable contador a valor_inicial, en cada iteración se incrementa en uno y cuando contador es mayor que el valor_final el bucle termina.
Si se incluye la palabra REVERSE el bucle se ejecuta desde el valor final hasta el inicial. Nótese que siempre se indica primero el valor inicial y después el valor final, independientemente de si se usa REVERSE o no.

Sección de excepciones

Esta sección se encarga del control y el tratamiento de errores en tiempo de ejecución.
Es el punto al que se transfiere el control del programa cuando surge un problema. Existe un conjunto de excepciones predefinidas que informan de los errores producidos en la ejecución de las sentencias SQL por parte del sistema de gestión de bases de datos y además de éstas, el

programador puede definir excepciones de uso específico, cuyo control es enteramente gestionado por él.

7.3.4 Comentarios

En PL/SQL los comentarios de una sola línea comienzan con '--', los de varias líneas comienzan
con /* y terminan con */.

7.3.5 Funciones y procedimientos.

Una vez que tenemos escrito un bloque de código, podemos guardarlo en un fichero .sql para su posterior uso, o bien guardarlo con la base de datos en una caché de librerías para que pueda ser ejecutado por cualquier aplicación. Por estar almacenados se les llama **procedimiento** o **función almacenado** (stored procedure/function).

Entre sus características podemos nombrar:

- Tienen un nombre.
- Aceptan parámetros.
- Retornan valores.
- Se almacenan en el diccionario de datos.

A la hora de guardar un bloque de código hay que tener en cuenta ciertas normas:

- Palabra reservada **DECLARE se omite**.
- Podremos crear procedimientos y funciones. Los procedimientos no podrán retornar ningún valor, mientras que las funciones deben retornar un valor de un tipo de dato básico. Generalmente usamos un procedimiento para ejecutar una acción y una función para calcular un valor.

Para crear un procedimiento (stored procedure:procedimiento almacenado) usaremos la siguiente sintaxis:

```
CREATE [OR REPLACE] PROCEDURE nombre_proc(argumento1
[IN|OUT|IN OUT] tipodato1, argumento2 [IN|OUT|IN OUT]
tipodato2, ...) IS|AS
     /*declaraciones locales*/
BEGIN
     /*sentencias */
[EXCEPTION]
          /*sentencias para el tratamiento de la excepción*/
END [nombre];
```

Para crear una función usaremos la siguiente sintaxis:

```
CREATE [OR REPLACE] FUNCTION nombre_func(argumento1 [IN|OUT|IN
OUT] tipodato1, argumento2 [IN|OUT|IN OUT] tipodato2, ...)
RETURN tipodato IS|AS
     /*declaraciones locales*/
BEGIN
     /*sentencias */
     RETURN expresion /*debe de haber una instrucción
                       return*/
[EXCEPTION]
         /*sentencias para el tratamiento de la excepción*/

END [nombre];
```

Las funciones y los procedimientos son estructurados de la misma forma excepto que las funciones cuentan con la estructura RETURN.

Recordemos que la función debe devolver aunque sea un 0 por lo que debe aparecer un RETURN en el cuerpo del programa, en caso de no estar presente será invocada la excepción PROGRAM_ERROR durante la ejecución. Con la estructura RETURN se completa la ejecución del subprograma y hace que regrese el control al programa que la llamó.

El uso de OR REPLACE permite sobrescribir un procedimiento o función existente, si se omite, y el procedimiento o la función ya existe, se producirá un error.

Una cuestión importante son los parámetros, estos permiten comunicarse con los procedimientos o funciones. Pueden ser de tres tipos:

- De entrada, en este caso se utiliza la palabra reservada IN.
- De salida, se utiliza OUT.
- De entrada y salida, se utiliza IN OUT.

El tipo de un parámetro no puede tener restricciones de tamaño y su modo por defecto es IN, los parámetros en la llamada tienen que ir en orden. Si no hay parámetros, se omiten los paréntesis.

Para ejecutarlos se invocan por su nombre con el comando EXECUTE, aunque previamente deben ser compilados con el comando START. Para depurar los errores de compilación se puede utilizar el comando SHOW ERRORS. Para conocerlos se pueden utilizar dos vistas del diccionario de datos user_objects y user_source.
Si se desea eliminar (borrar) un procedimiento almacenado, se usa la instrucción:

```
SQL> DROP PROCEDURE nombre;
```

Si se desea eliminar (borrar) una función, se usa la instrucción:

```
SQL> DROP FUNCTION nombre;
```

Veamos ahora un ejemplo de procedimiento y la forma de ejecutarlo:

```
CREATE OR REPLACE PROCEDURE consulta_empleados
(v_id IN empleado.numemp%TYPE,
v_nombre OUT empleado.nombre%TYPE,
v_sueldo OUT empleado.sueldo%TYPE,
v_comis OUT empleado.comision%TYPE)
IS
BEGIN
SELECT nombre, sueldo, comision
INTO v_nombre, v_sueldo, v_comis
FROM empleado
WHERE numemp=v_id;
END consulta_empleados;
/
```

Fichero consultaEmp.sql

Si tenemos almacenado la consulta en el fichero consultaEmp.sql, para ejecutarlo tenemos que hacer lo siguiente:

```
START consulta1.sql; /*compilación*/
```

Para recoger los valores de salida y/o para introducir valores en la ejecución a través de los parámetros podemos utilizar constantes y variables externas, previamente definidas en SQL*Plus con la instrucción VARIABLE nombre_var. De esta manera, aunque no se pueden utilizar variables externas dentro de procedimientos ni funciones, en una llamada a procedimiento se pueden utilizar poniendo su nombre precedido del carácter ‘:’.

Declaramos tres variables externas:

```
VARIABLE ex_nombre VARCHAR2(15);
VARIABLE ex_sueldo NUMBER;
VARIABLE ex_comision NUMBER;
```

Ejecutamos la consulta:

```
EXECUTE
consulta_empleados(6565,:ex_nombre,:ex_sueldo,:ex_comision);
```

```
PRINT ex_nombre; --imprimimos los resultados.
PRINT ex_sueldo;
PRINT ex_comision;
```

7.3.6 Uso de etiquetas

PL/SQL permite utilizar etiquetas que identifiquen un punto en el código. Esto nos proporciona poder utilizar las sentencias GOTO que hacen que el flujo del programa salte a donde este situada la etiqueta a la que hace referencia el GOTO. Pero aunque PL/SQL nos proporcione poder utilizar esta sentencia, se desaconseja su uso.

```
DECLARE
v_contador BINARY_INTEGER := 1;
BEGIN
LOOP
INSERT INTO temp_table
VALUES (v_contador, 'Loop count');
v_contador := v_contador + 1;
IF v_contador > 50 THEN
GOTO l_findelLoop;
END IF;
END LOOP;

<<l_findelLoop>>
INSERT INTO temp_table (char_col)
VALUES ('Done!');
END;
/
```

7.3.7 Acceso a datos

El acceso a datos en PL/SQL se realiza utilizando sentencias SQL. Podemos utilizar directamente sentencias DML (INSERT, DELETE, UPDATE) y también realizar consultas (SELECT), utilizando tanto constantes como variables para manipular u obtener datos de las tablas. Prestaremos especial atención a las consultas que devuelven un numero (a priori, indeterminado) de filas, ya que requieren el uso de una nueva estructura: el *cursor*.
También se puede realizar el control de transacciones (COMMIT, etc.) directamente. No es posible, en cambio, ejecutar sentencias DDL del tipo CREATE TABLE directamente en PL/SQL, aunque si hay una forma alternativa de ejecutar ese tipo de sentencias.

7.3.8 Manipulación de datos (sentencias DML)

Las sentencias DML se escriben directamente en el código PL/SQL, y forman una sentencia "normal", pudiendo estar en un bucle o en una de las ramas de una estructura condicional. Cuando la sentencia necesite un valor, este puede ser una constante o una variable previamente declarada, del mismo tipo que la columna que referencia.

7.3.9 Ejecución de sentencias DDL

Como se ha indicado antes, las sentencias DDL no pueden ejecutarse directamente en un bloque PL/SQL, sin embargo, pueden ejecutarse utilizando SQL dinámico, que ejecuta un string que contiene una sentencia SQL válida. Así, el siguiente bloque de código sentencia crea una tabla (el string debe ir entre comillas simples):

```
BEGIN
EXECUTE IMMEDIATE 'CREATE TABLE TABLA(CAMPO CHAR(10))';
END;
```

Aunque el SQL dinámico ofrece grandes posibilidades, no entraremos en más detalle sobre él. Para más información, consulte la información que ofrece Oracle.

7.3.10 Depuración de programas, entrada y salida

En los apartados anteriores hemos estudiado muy diversos tipos de instrucciones o sentencias, pero no hemos hecho referencia a las instrucciones de entrada y salida; estas nos sirven para ver los resultados obtenidos tras la ejecución o bien para depurar el programa.

DBMS_OUTPUT nos permite presentar visualmente en pantalla los datos seleccionados. Esto es controlado con el parámetro SERVEROUTPUT (en SQL*Plus), para verlo usaremos SHOW SERVEROUTPUT y para activarlo utilizamos la orden SET SERVEROUTPUT ON.

```
SQL> set serveroutput on
```

Se pueden elegir gran cantidad de opciones para manejar datos de input o output.

- DBMS_OUTPUT.ENABLE: Permite output processing.
- DBMS_OUTPUT.DISABLE: Disables output processing.
- DBMS_OUTPUT.PUT_LINE(<argumento>): el <argumento> en la pantalla, y acaba con un retorno de carro.
- DBMS_OUTPUT.PUT(<argumento>): Pone en el buffer de salida el <argumento>, pero no lo escribe hasta recibir la orden DBMS_OUTPUT.NEW_LINE. Se suele utilizar para escribir varias cosas en una línea.

- DBMS_OUTPUT.GET_LINE--Obtiene una línea del buffer.

Veamos un ejemplo:

```
SET SERVEROUTPUT ON;

BEGIN
DBMS_OUTPUT.PUT_LINE('Estamos a '||SYSDATE);
DBMS_OUTPUT.PUT('a ');
DBMS_OUTPUT.PUT('b');
DBMS_OUTPUT.NEW_LINE;
END;
/

Estamos a 15-MAR-04
a b
Procedimiento PL/SQL terminado correctamente.
```

7.3.11 Como crearlos y ejecutarlos

Podemos crear programas con cualquier editor y ejecutarlos desde el prompt de sql con STAR o @. Los ficheros creados serán de texto y tendrán la extensión sql.
Para que un fichero se ejecute correctamente debe de tener en su última línea el símbolo "/".

```
BEGIN
NULL;
END;
/
```

7.4 Ejemplos

1.- En la tabla emp incrementar el salario el 10% a los empleados que tengan una comisión superior al 5% del salario.

```
BEGIN
UPDATE emp
SET sal=sal+sal*(10/100)
WHERE comi>(sal*5/100);
END;
/
```

2.- Insertar un empleado en la tabla emp. Su número será superior a los existentes y la fecha de incorporación a la empresa será la actual.

```
DECLARE
num_empleado emp.empno%TYPE;
fecha emp.hiredate%TYPE;
BEGIN
SELECT MAX(empno) INTO num_empleado FROM emp;
SELECT SYSDATE INTO fecha FROM DUAL;
num_empleado:=num_empleado +1;
INSERT INTO emp VALUES
(num_empleado,'PEDRO','MATEMATIC',7839,fecha,3000,NULL,20,NULL
);
END;
/
```

3.- Realizar un procedimiento para borrar un empleado recibiendo como parámetro el número de empleado.

```
DECLARE
num_empleado emp.empno%TYPE;
MRW ROWID;
BEGIN
SELECT empno,ROWID INTO num_empleado,MRW FROM emp WHERE
empno=&EMPLEADO;
DELETE FROM emp WHERE ROWID=MRW;
END;
/

UNDEFINE EMPLEADO;
```

4.- Realizar un procedimiento para modificar la localidad de un departamento. El procedimiento recibe como parámetros la localidad y el número de departamento.

```
DECLARE
localidad dept.loc%TYPE;
numero dept.deptno%TYPE;
R ROWID;
BEGIN
SELECT loc,deptno,ROWID INTO localidad,numero,R FROM dept
WHERE deptno=&&NUMERO;
UPDATE dept
SET loc='&LOCALIDAD' WHERE ROWID=R;
END;
/

UNDEFINE NUMERO;
UNDEFINE LOCALIDAD;
```

5.- Añadir un nuevo empleado en la tabla emp. El número de empleado será el del último + 10. La fecha la actual, el departamento el 40.

```
DECLARE
empleado emp.empno%TYPE;
fechaA emp.hiredate%TYPE;

PROCEDURE ejercicio_5_1(fecha IN emp.hiredate%TYPE,
numero_emp IN emp.empno%TYPE,departamento IN emp.deptno%TYPE)
IS
BEGIN
INSERT INTO emp VALUES
(numero_emp,'NOMBRE','JOB',NULL,fecha,999,NULL,departamento);
END ejercicio_5_1;

FUNCTION transformacion1 RETURN NUMBER IS
retorno emp.empno%TYPE;
mayor_numero emp.empno%TYPE;
BEGIN
SELECT MAX(empno) INTO mayor_numero FROM emp;
retorno:=mayor_numero+10;
RETURN(retorno);
END transformacion1;


FUNCTION transformacion2 RETURN DATE IS
retorno emp.hiredate%TYPE;
BEGIN
retorno:=SYSDATE;
RETURN(retorno);
END transformacion2;

BEGIN
empleado:=transformacion1;
fechaA:=transformacion2;
ejercicio_5_1(fechaA,empleado,40);
END;
/
```

6.- Sobre la tabla emp insertar un nuevo 'VENDEDOR'. Su número de empleado será consecutivo. Su departamento será el mismo que el de su jefe 'VICTOR'.

```
DECLARE
empleado emp.empno%TYPE;
departamento emp.deptno%TYPE;
manager emp.empno%TYPE;
```

```
PROCEDURE ejercicio_5_2(numero_emp IN emp.empno%TYPE,
departamento_emp IN emp.deptno%TYPE,
manager_emp IN emp.empno%TYPE)
IS
BEGIN
INSERT INTO emp
VALUES(numero_emp,'JUAN','VENDEDOR',manager,'20/02/80',1500,NU
LL,departamento);
END ejercicio_5_2;

FUNCTION transformacion3 RETURN NUMBER IS
retorno emp.empno%TYPE;
mayor_numero emp.empno%TYPE;
BEGIN
SELECT MAX(empno) INTO mayor_numero FROM emp;
retorno:=mayor_numero+1;
RETURN(retorno);
END transformacion3;

FUNCTION transformacion4 RETURN NUMBER IS
departamento1 emp.deptno%TYPE;
BEGIN
SELECT deptno INTO departamento1 FROM emp WHERE
UPPER(ename)='VICTOR';
RETURN(departamento1);
END transformacion4;

FUNCTION transformacion5 RETURN NUMBER IS
manager1 emp.empno%TYPE;
BEGIN
SELECT empno INTO manager1 FROM emp WHERE
UPPER(ename)='VICTOR';
RETURN(manager1);
END transformacion5;

BEGIN
empleado:=transformacion3;
departamento:=transformacion4;
manager:=transformacion5;
ejercicio_5_2 (empleado,departamento,manager);
END;
/
```

7.- Realizar un procedimiento para guardar el nombre y el número de empleado de los empleados cuyo apellido contenga una cadena. La cadena se le pasa al procedimiento.

```
DECLARE
--T emp.ename%TYPE;


PROCEDURE buscar(cadena IN emp.ename%TYPE) IS
E emp.empno%TYPE;
P emp.ename%TYPE;
BEGIN
SELECT empno,ename INTO E,P FROM emp WHERE P LIKE 'JUAN';
DBMS_OUTPUT.PUT_LINE(E || P);
END buscar;


BEGIN

buscar('JUAN');
END;
```

8.-Procesa pedidos cds de 90 min, recupera la cantidad en "stock", si es mayor que 0 actualiza la venta, si no, inserta un mensaje de alerta.

```
DECLARE
cantidad_act NUMBER(5);
BEGIN
SELECT cantidad INTO cantidad_act FROM inventario WHERE
producto = 'Cds de 90 min' FOR UPDATE OF cantidad;
IF cantidad_act > 0 THEN -- comprueba cantidad
     UPDATE inventario SET cantidad = cantidad - 1 WHERE
     producto ='Cds de 90 min';
     INSERT INTO reg_ventas VALUES ('Compra de CDs de 90
     minutos', SYSDATE);
ELSE
     INSERT INTO reg_ventas VALUES ('No quedan CDs de 90
     minutos',SYSDATE);
END IF;
COMMIT;
END;
```

7.5 ¿Qué es un cursor?

Un *cursor* es un apuntador, puntero o manejador para el área de contexto.

El área de contexto es la memoria designada para procesar una instrucción SQL, la cual incluye:

- el número de registros procesados.
- un apuntador a la representación de la instrucción SQL analizada.
- en el caso de una consulta, el conjunto de registros que regresan de la consulta.

7.6 ¿Para qué sirve un cursor?

Sirve para que un programa PL/SQL, por medio de él, pueda controlar el *área de contexto*.

Cuando escribimos una consulta SELECT en un intérprete de SQL, este nos muestra las distintas filas de resultados por pantalla, pero cuando utilizamos la misma consulta dentro de un lenguaje de programación lo más común es almacenar los resultados en variables para tratarlos posteriormente, por lo que nos surge el problema de cómo realizar esto.

Tenemos dos soluciones dependiendo del número de filas que retorna la consulta SELECT:

Devuelve cero o una fila: El valor se almacena en tantas variables como columnas consultadas.

Por ejemplo, si escribimos un SELECT de dos columnas, y sólo devuelve una fila (matriz 1x2), podremos almacenar el valor dentro de dos variables definidas para este uso.

```
SELECT <lista de atributos> INTO <lista de variables>;
o
SELECT <lista de atributos> INTO <registro PL/SQL>;
```

En esta situación realizar una consulta SELECT …INTO dentro de un bloque de código no hay más que escribir la consulta en el lugar adecuado y ésta se ejecutará y retornará el valor a las variables correspondientes.

Devuelve más de una fila: Utilizamos los cursores porque la solución anterior no es aplicable en este caso.

Existen dos tipos de cursores:

- Los *cursores implícitos* son creados por Oracle para manejar alguna instrucción SQL y no son declarados por el programador. Sirve para procesar las órdenes INSERT, UPDATE, DELETE y las órdenes SELECT...INTO de una sola fila y es el motor PL/SQL quien abre y cierra el cursor.

- Los *cursores explícitos* son aquellos que se declaran, generalmente por medio de una consulta SQL y son declarados por el programador. Sirven para procesar sentencias SELECT que devuelven un número indeterminado de filas.

7.7 Creación y utilización de cursores

Si nos encontramos en la situación en la cual nos hace falta un cursor, es decir, si intuimos o sabemos que el resultado de la consulta va a devolver más de una fila, tenemos que:

1. Declarar el cursor (dentro de la sección DECLARE).
2. Abrir el cursor.
3. Recuperar cada una de sus filas (bucle).
4. Cerrar el cursor.

1. Declarar el cursor.

Definimos una variable de tipo cursor (en la sección DECLARE), es decir, asociamos un nombre de cursor a una sentencia SELECT. Podemos incluir variables declaradas en esta sección. Es importante indicar que sólo se está declarando, y no ejecutando, la consulta, por lo que no es necesario que las variables estén inicializadas.
La sintaxis básica es:

```
Declare
      Cursor nombre_cursor IS
      SELECT …
      FROM …;
```

Una vez que el cursor está declarado ya podrá ser utilizado dentro del bloque de código.

2. Abrir el cursor.

La apertura del cursor hace que se ejecute la sentencia SELECT, en una zona de memoria reservada para el cursor se almacenan las filas que esta devuelve y sitúa el cursor en la primera fila.
Si la consulta incluía variables (que se denominan variables *acopladas*), estas deben tener el valor deseado antes de abrir el cursor. Una vez abierto el cursor, podemos cambiar el valor de las variables acopladas ya que esto no afecta al cursor. Esta acción se debe realizarse sólo una vez.
La sintaxis de apertura de un cursor es:

```
      OPEN nombre_cursor;
```

Veamos un ejemplo de variables acopladas:

```
DECLARE
--Declaración del cursor
CURSOR c_estudiantes (v_principal estudiantes.principal%TYPE)
IS
     SELECT id, nombre, apellidos
     FROM estudiantes
     WHERE principal=v_principal;
BEGIN
     --Utilizamos la orden OPEN para enviar el valor de
     v_principal
     OPEN c_estudiantes('Ciencias de la computación');
```

3. Recuperar cada una de sus filas.

Una vez hecha la apertura del cursor podemos recuperar una por una las filas resultantes de la ejecución de la sentencia SELECT. Este paso es similar a hacer una consulta SELECT de una sola fila, porque se nos asegura que no vamos a recuperar más de una fila. Cada fila que se obtenga se almacena, al igual que se hacía con SELECT ... INTO, en una lista de variables o en un registro PL/SQL.
La sintaxis de la orden que obtiene la fila actual y desplaza el apuntador del cursor a la siguiente fila, es:

```
FETCH nombre_cursor INTO lista de variables | registro PL/SQL;
```

Podremos recuperar filas mientras la consulta SELECT tenga filas pendientes de recuperar. Para conocer si hay o no más filas por recuperar utilizamos los llamados atributos del cursor. Los atributos nos permiten evaluar el estado en que se encuentra el cursor.

Los atributos de estado son los siguientes:

- **%FOUND**: Es un atributo booleano que devuelve cierto si la última orden FETCH devolvió una fila, y falso en caso contrario. Si se trata de comprobar el valor de <nombre_cursor>%FOUND mientras el cursor no está abierto, se devuelve el error ORA-1001 (cursor no válido).

- **%NOTFOUND**: Es lo opuesto a %FOUND, devuelve cierto si la orden FETCH no devuelve una fila, y falso si la devuelve

- **%ISOPEN**: También booleano, devuelve cierto si el cursor está abierto, y falso en caso contrario.

- **%ROWCOUNT**: Es un atributo numérico devuelve el número de filas extraídas por el cursor hasta el momento.

Veamos algunas clausulas que complementan a FETCH:

- FETCH FIRST recupera la primera fila de resultados.
- FETCH LAST recupera la última fila de resultados.
- FETCH PRIOR recupera la fila de resultados que precede inmediatamente a la fila actual del cursor.
- FETCH NEXT recupera la fila de resultados que sigue inmediatamente a la fila actual del cursor. Este es el comportamiento por omisión si no se especifica movimiento y corresponde al movimiento estandar del cursor.
- FETCH ABSOLUTE recupera una fila específica mediante su número de fila.
- FETCH RELATIVE mueve el cursor hacia delante o hacia atrás un número específico de filas relativo a su posición actual.

Como el proceso de recuperar filas es repetitivo podemos utilizar un bucle para procesar todas las filas hasta llegar a la última.
Esto lo podremos hacer a través del siguiente bloque de código:

```
LOOP
     FETCH nombre_cursor INTO lista de variables |
     registro PL/SQL;

     EXIT WHEN nombre_cursor%NOTFOUND;

     <procesar cada una de las filas>

END LOOP;
```

Además de la estructura LOOP…END LOOP podemos utilizar los bucles WHILE y FOR.
Este tipo de bucles utiliza lo que se denomina lectura adelantada, ya que se lee antes de iniciar el bucle (así se puede comprobar si hay o no datos) y luego, en el bucle, se procesa la fila leída y finalmente se obtiene la siguiente. Esto es necesario para que la condición del bucle sea evaluada en cada iteración del bucle.

Bucles de cursor FOR:

Los dos tipos de bucles vistos hasta ahora de extracción descritos requieren un procesamiento explícito del cursor mediante las órdenes OPEN, FETCH y CLOSE. PL/SQL proporciona un nuevo tipo de bucle, mucho más simple, que realiza de modo implícito el procesamiento del cursor. Este bucle recibe el nombre de bucle de cursor FOR, y que tiene la sintaxis general siguiente:

```
FOR <registro PL/SQL>IN <nombre cursor>LOOP
/* Procesar la fila actual */
END LOOP;
```

Este tipo de cursor tiene varias características especiales:

- El registro PL/SQL que se usa no se declara en la sección declarativa, sino que el bucle FOR lo declara de forma implícita, al igual que los índices de los bucles FOR numéricos. En este caso, además, el registro implícito tiene sólo los campos que se seleccionan.

- No hay que abrir el cursor, ya que el FOR lo hace automáticamente antes del bucle, ni comprobar cuando se acaban las filas (el bucle comprueba automáticamente el atributo %FOUND antes de cada iteración), ni cerrarlo al terminar, que es cerrado automáticamente al terminar.

Como se ve, los bucles de cursor FOR tienen la ventaja de que proporcionan la funcionalidad de un bucle para recorrer un cursor de una forma simple y limpia con una sintaxis mínima.

4.Cerrar el cursor

Cuando hayamos procesado todas las filas, se deben liberar todos los recursos concernientes al cursor, como la memoria en la cual se encuentra almacenado el resultado de la consulta SELECT. Si no cerrásemos el cursor esa zona de memoria quedaría almacenada con el nombre dado al cursor por lo que la siguiente vez que ejecutásemos ese bloque de código, nos daría la excepción CURSOR_ALREADY_OPEN (cursor ya abierto) cuando intentásemos nuevamente abrir el cursor.
Evidentemente, una vez que se cierra un cursor es ilegal realizar extracciones (FETCH) de él, o tratar de volver a cerrarlo.
Ambos casos provocarían un error.

Para cerrar el cursor se utiliza la siguiente sintaxis:

```
CLOSE nombre_cursor;
```

7.7.1 Cursores actualizables

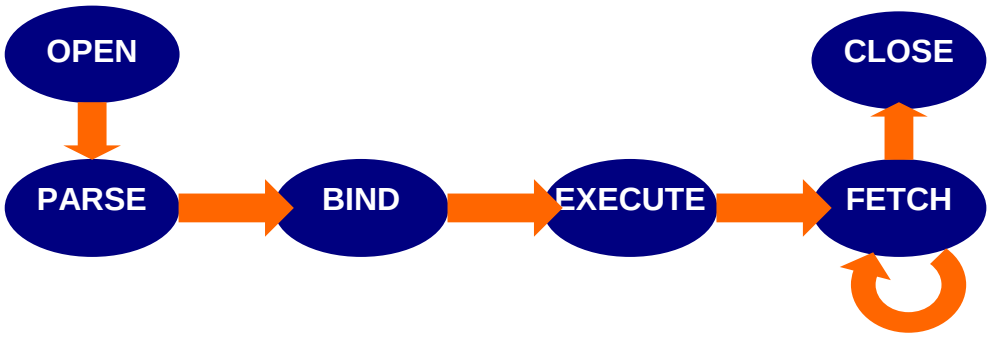
Los cursores vistos hasta ahora son de "sólo lectura", es decir, podemos obtener los datos, pero el conjunto activo que devuelve el cursor no es modificable. En algún caso es útil poder actualizar la fila actual del cursor, y eso se puede implementar con cursores actualizables o cursores FOR UPDATE.

Este tipo de cursores se declaran añadiendo FOR UPDATE después de la sentencia SELECT. Para actualizar la fila actual se usa una orden UPDATE, y la condición será WHERE CURRENT OF <nombre_cursor>,como se ve en el siguiente ejemplo:

```
DECLARE
     --Créditos a añadir al total de cada estudiante
v_numCreditos    clases.num_creditos%TYPE;
```

```
      --El cursor selecciona los alumnos matriculados en
Matemáticas 100
CURSOR c_registroEstudiantes IS
      SELECT *
            FROM estudiantes
            WHERE id IN  (SELECT estudiante_id
                  FROM registro_estudiantes
                  WHERE departamento = 'Matemáticas'
                  AND curso =100)
            FOR UPDATE OF creditos_actuales;
BEGIN
--Crea el bucle de extracción
FOR v_estudiantesInfo IN c_registroEstudiantes LOOP
--obtiene el número de créditos Matemáticas 100
SELECT num_creditos
INTO v_numCreditos
      FROM clases
      WHERE departamento = 'Matemáticas'
      AND curso=100;
--Actualiza la fila que acaba de recuperar con el cursor.
UPDATE estudiantes
      SET creditos_actuales = creditos_actuales + v_numCreditos
WHERE CURRENT OF c_registroEstudiantes;
END LOOP;
--Confirma el trabajo.
COMMIT;
END;
/
```

Veamos a continuación las fases para procesar una instrucción SQL:



7.8 Ejemplos

1.- Añadir la columna total2 y en ella escribir la suma del salario y la comisión de los empleados con comisión distinta de 0.

```
ALTER TABLE emp ADD(total2 NUMBER(7,2));
DECLARE
CURSOR cursor2 IS SELECT comi,sal FROM emp
WHERE comi IS NOT NULL AND comi<>0 FOR UPDATE;

BEGIN
FOR reg IN cursor2 LOOP
UPDATE emp
SET total2=sal+comi
WHERE CURRENT OF cursor2;
END LOOP;
END;
/
```

2.- Realizar un procedimiento para cambiar la fecha por el número de año.

```
DECLARE
fecha VARCHAR2(4);
CURSOR cursor1 IS SELECT TO_CHAR(hiredate,'YYYY') FROM emp FOR
UPDATE;
BEGIN
OPEN cursor1;
LOOP
FETCH cursor1 INTO fecha;
EXIT WHEN cursor1%NOTFOUND;
UPDATE emp
SET hiredate='FECHA' WHERE CURRENT OF cursor1;
END LOOP;
CLOSE cursor1;
END;
/
```

3.- Que encuentre el primer empleado que tiene un salario superior a 2000.

```
DECLARE
PROCEDURE encontrar IS
CURSOR C1 IS SELECT empno,ename FROM emp WHERE sal>2000;
registro C1%ROWTYPE;

BEGIN
OPEN C1;
FETCH C1 INTO registro;
IF C1%ROWCOUNT=1 THEN
```

```
DBMS_OUTPUT.PUT_LINE(registro.empno || ' '||registro.ename);
ELSE
DBMS_OUTPUT.PUT_LINE('No se encontro');
END IF;
CLOSE C1;
END encontrar;

BEGIN
encontrar;
END;
```

4.- Listar todos los empleados que tienen un nombre de 5 letras.

```
DECLARE
PROCEDURE ejercicio_5_4 IS
CURSOR C1 IS SELECT ename,trabajo,sal,deptno FROM emp WHERE
LENGTH(ename)=&long;
registro C1%ROWTYPE;

BEGIN
OPEN C1;
LOOP
FETCH C1 INTO registro;
EXIT WHEN C1%NOTFOUND;
DBMS_OUTPUT.PUT_LINE (registro.ename ||' '||registro.trabajo
||' '||registro.sal||' '||registro.deptno);
END LOOP;
CLOSE C1;
END ejercicio_5_4;
BEGIN
ejercicio_7_4;
END;
/
```

5.- Buscar todos los empleados que tienen un salario + comisión superior a 2000 y asignarles como nuevo salario esta suma. Sólo para los que tienen comisión.

```
DECLARE
PROCEDURE actualizar IS
CURSOR C1 IS SELECT ename,sal,comi FROM emp WHERE
sal+comi>2000 FOR UPDATE;
registro C1%ROWTYPE;
BEGIN
OPEN C1;
LOOP
FETCH C1 INTO registro;
```

```
EXIT WHEN C1%NOTFOUND;
UPDATE emp
SET sal=sal+comi WHERE CURRENT OF C1;
END LOOP;
CLOSE C1;
END actualizar;

BEGIN
actualizar;
END;
/
```

6.- Realizar un procedimiento que guarde en una vista el nombre y la fecha de alta de los empleados por nombre.

```
DECLARE
PROCEDURE guardar IS
CURSOR cursor1 IS SELECT ename,hiredate FROM emp ORDER BY
ename ;
registro cursor1%ROWTYPE;

BEGIN
OPEN cursor1;
LOOP
FETCH cursor1 INTO registro;
EXIT WHEN cursor1%NOTFOUND;
INSERT INTO Borrar
VALUES (registro.ename,registro.hiredate);
END LOOP;
CLOSE cursor1;
END guardar;

BEGIN
guardar;

END;
```

7.9 ¿Qué es una excepción?

Una excepción es una situación especial que se produce durante la ejecución de un bloque de PL/SQL, puede ser controlada de forma implícita (Oracle lo gestiona) o explícita (el programador lo gestiona).
Una excepción puede ser un error de ejecución (una división entre 0) o cualquier otro tipo de suceso.

7.10 ¿Para qué sirve una excepción?

El manejo de excepciones es muy importante en el diálogo con los sistemas de gestión de bases de datos, ya que permite responder ante cualquier problema que pueda ocurrir en la ejecución de cualquier operación. Por defecto, la ejecución de la aplicación finaliza al presentarse algún error grave, pero con la definición de manejadores específicos es posible realizar una serie de acciones y continuar la ejecución de la aplicación.

7.11 Programación del manejo de excepciones

Las excepciones se manejan en la sección de control de excepciones, que va situada después de la sección ejecutable del bloque y comienza por la palabra clave EXCEPTION. Esto nos permite separar el control de errores de la ejecución normal del código, con lo que se obtiene un código más claro y legible.

Existen dos tipos de excepciones, las predefinidas por Oracle:

Nombre de la excepción	Código asociado en Oracle	Descripción
DUP_VAL_ON_INDEX	ORA-0001	Cuando se intenta insertar una clave primaria duplicada
TIMEOUT_ON_RESOURCE	ORA-0051	Tiempo fuera ocurrido mientras esperaba un recurso.
TRANSACTION_BACKED_OUT	ORA-0061	La transacción fue desecha por un "bloqueo mortal".
INVALID_CURSOR	ORA-1001	Operación ilegal con un cursor.
NOT_LOGGED_ON	ORA-1012	Sin conexión a Oracle.
LOGIN_DENIED	ORA-1017	Nombre de usuario o password inválido.
NO_DATA_FOUND	ORA-01403	No se encontraron datos.
TOO_MANY_ROWS	ORA-01422	La instrucción `SELECT ... INTO` devuelve más de un registro.
ZERO_DIVIDE	ORA-01476	Un valor numérico está dividido por cero.
INVALID_NUMBER	ORA-1722	Conversión inválida a un número.
STORAGE_ERROR	ORA-06500	No hay memoria suficiente.
PROGRAM_ERROR	ORA-06501	Error interno de PL/SQL.
VALUE_ERROR	ORA-06504	Error al trucar o convertir valores, o en una operación aritmética.
CURSOR_ALREADY_O	ORA-6511	Al intentar abrir un cursor que ya está

PEN		abierto.

Las **definidas por el Programador**:

Son mensajes de error que genera el usuario debido a las posibles violaciones de las reglas de la aplicación. Las excepciones deben ser declaradas dentro de la sección DECLARE, como si de una variable se tratase:

```
DECLARE
        <nombre>  EXCEPTION;
```

Una vez definida la excepción esta puede ser lanzada, bien automáticamente por Oracle o manualmente mediante la instrucción *Raise*.

```
SELECT COUNT(*)
INTO num_alumnos;

IF num_alumnos=0 THEN
RAISE e_sin_alumnos;
END IF;
```

Cuando la excepción es lanzada, el control pasa a la sección de excepciones de un bloque PL/SQL, concretamente al manejador apropiado. Si no existe esta sección, en el caso de bloques anidados el error se propaga al bloque que contiene al que reporta el error, es decir, si el bloque interno finaliza manteniendo activa una excepción, se ejecuta el manejador de excepciones del bloque externo para su gestión como si la excepción se hubiera producido en cualquier otra instrucción, de este modo, si se desea hacer frente a excepciones que pudieran aparecer en la ejecución de una sentencia SQL sin finalizar el bloque, la solución consiste en incluir la sentencia en un sub-bloque PL/SQL que gestione la excepción.

```
DECLARE
        -- Declaracion de la excepcion
        a EXCEPTION;
    BEGIN
        ...

        RAISE a;
        -- El resto del codigo de esta seccion
        -- no es ejecutado
        ...

    EXCEPTION
```

```
    -- El control pasa al manejador cuando se lanza a
    WHEN a THEN
      -- Código a ejecutar para
        -- para manejar la excepcion
END;
```

La sección de excepciones consiste de una serie de manejadores. Un *manejador de excepción* consiste en código que es ejecutado cuando la excepción es lanzada. La sintaxis es la siguiente:

```
EXCEPTION
    WHEN nombre_excepcion1 THEN
        secuencia_de_instrucciones1;
    WHEN nombre_excepcion2 THEN
        secuencia_de_instrucciones2;
    WHEN OTHERS THEN
        secuencia_de_instrucciones3;
END;
```

Cuando se usa `WHEN OTHERS` y se desea saber que error ocurrió, las funciones `SQLCODE` y `SQLERRM` proporcionan esta información.

SQLCODE
: regresa el código del error que lanzó la excepción

SQLERRM
: regresa el mensaje de error correspondiente al error que lanzó la excepción. Esta función también acepta un argumento numérico, el cual debe de corresponder al texto asociado con ese número.

Veamos como se realiza la ejecución en la parte de manejadores:

- Se busca, por orden secuencial, un manejador que corresponda con la excepción activada, si no se encuentra ningún manejador, se finaliza el bloque manteniendo activa la excepción.

- La cláusula OTHERS corresponde a todas las excepciones, por lo que resulta conveniente ponerla si se desea gestionar todas las excepciones, y poner su manejador el último para que los anteriores se puedan ejecutar.

- Una vez encontrado el manejador, se desactiva la excepción y se ejecutan las instrucciones asociadas, finalizándose a continuación el bloque.

- Si se deseara mantener activa la excepción, o se desea activar cualquier otra, es posible incluir la sentencia RAISE que finalizaría el bloque y activaría la excepción asociada.

Además de lo expuesto anteriormente existe otra forma de manejar las excepciones: la función `RAISE_APPLICATION_ERROR(código,'Mensaje de error')` que permite definir nuevos códigos y mensajes de error. El código es un NUMBER entre –20000 y –20999, y el mensaje es VARCHAR2512.
Esta función nos la proporciona el paquete predefinido DBMS_STANDARD y STANDARD.

7.12 Ejemplos

1.- Realizar un procedimiento para crear una vista con el nombre de cada departamento y número de empleados que tiene.

```
CREATE OR REPLACE PROCEDURE dinamico (instruccion VARCHAR2) AS
cid INTEGER;
dummy INTEGER;
BEGIN
cid := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(cid, instruccion, dbms_sql.v7);
dummy:=DBMS_SQL.EXECUTE(cid);
DBMS_SQL.CLOSE_CURSOR(cid);
EXCEPTION
WHEN OTHERS THEN
DBMS_SQL.CLOSE_CURSOR(cid);
RAISE;
END dinamico;
/

EXECUTE dinamico('CREATE VIEW Vista7 AS (SELECT
COUNT(DISTINCT(ename)) n_emp,deptno FROM emp GROUP BY
deptno)');
```

2.- Cómo borrar o crear una tabla utilizando PL*SQL.

```
DECLARE
PROCEDURE drop_table IS
cid INTEGER;
BEGIN
```

```
cid := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(cid, 'DROP TABLE ' || 'emp', dbms_sql.v7);
DBMS_SQL.CLOSE_CURSOR(cid);
EXCEPTION
WHEN OTHERS THEN
DBMS_SQL.CLOSE_CURSOR(cid);
RAISE;
END drop_table;

BEGIN
drop_table;
END;
```

3.Borrar un departamento de la tabla departamento controlando si es posible. Será posible si el departamento no tiene empleados.

```
DECLARE
e_hay_emp EXCEPTION;
v_depnum departamento.depnum%TYPE := 777;
BEGIN
IF (SELECT COUNT(*) FROM empleados WHERE depnum = v_depnum)=0
THEN DELETE FROM departamento WHERE depnum = v_depnum;
ELSE RAISE e_hay_emp;
END IF;
COMMIT;
EXCEPTION
WHEN e_hay_emp THEN
RAISE_APPLICATION_ERROR(-20001, 'No se puede borrar el
departamento ' || TO_CHAR(v_depnum)||' ya que tiene
empleados.');
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('abortado por error desconocido');
END;
```

4.- Definición de procedimiento que incrementa el salario del empleado cuyo código se indica a través del parámetro 1º en la cantidad que se indica en el parámetro 2º.

```
CREATE PROCEDURE incr_sal (emp_id IN NUMBER, sal_incr IN
NUMBER ) AS
BEGIN
 UPDATE emp
 SET sal = sal + sal_incr
 WHERE empno = emp_id;
 IF SQL%NOTFOUND THEN
```

```
 raise_application_error(-20011,'Código de empleado
inexistente: '||TO_CHAR(emp_id));
 END IF;
 END incr_sal;
 / -- desde SQL*Plus es preciso
```

Para incrementar 200 el salario del empleado 7369. Desde SQL*Plus introducimos:

```
 BEGIN
 incr_sal (7369,200);
 END;
 /
 o
 EXECUTE incr_sal
```

7.13 ¿Qué es un paquete?

Un paquete es un objeto PL/SQL que agrupa lógicamente otros objetos PL/SQL relacionados entre sí, encapsulándolos y convirtiéndolos en una unidad dentro de la base de datos.

7.14 ¿Para que sirven los paquetes?

Los Paquetes permiten encapsular elementos relacionados entre sí (tipos, variables, procedimientos, funciones), en un único módulo PL/SQL que llevará un nombre que identifique la funcionalidad del conjunto.

Ventajas del uso de Paquetes:

Dentro de las ventajas que ofrece el uso de paquetes podemos citar que:

- Permite modularizar el diseño de nuestra aplicación.
- Otorga flexibilidad al momento de diseñar la aplicación.
- Permite ocultar los detalles de implementación.
- Agrega mayor funcionalidad a nuestro desarrollo.
- Introduce mejoras al rendimiento
- Permite la "Sobrecarga de funciones" (Overloading).

7.15 Creación y Utilización de paquetes.

Un paquete puede contener procedimientos, funciones, cursores, variables, tipos y subtipos. En general, cualquier objeto que se pueda declarar dentro de la sección DECLARE de un bloque de código.

La creación de un paquete consiste en:

- Crear la cabecera del paquete donde se definen los tipos, variables, constantes, excepciones, cursores, procedimientos y funciones que podrán ser invocados desde fuera del paquete. La especificación o encabezado es la interfaz entre el Paquete y las aplicaciones que lo utilizan (obligatoria).

- Crear el cuerpo del paquete, donde se implementa los bloques de código de las funciones y procedimientos definidos en la cabecera del paquete (no obligatoria).

La parte de especificación y el cuerpo del paquete se almacenan por separado en el diccionario de datos. La especificación del paquete se hace en un fichero que contiene información acerca de sus contenidos, y el código de ejecución se almacena en otro fichero aparte con el cuerpo del paquete.

Para crear la cabecera del paquete utilizaremos la siguiente instrucción:

```
CREATE [OR REPLACE] PACKAGE nombre IS|AS –especificación (parte visible)
-declaración de tipos y variables públicas
-especificación de procedimientos y funciones
END [nombre];
```

Para crear el cuerpo del paquete:

```
CREATE [OR REPLACE] PACKAGE BODY nombre IS|AS –cuerpo (parte oculta)
--declaración de tipos y variables privadas
--cuerpo de procedimientos y funciones
[BEGIN
--rutinas de ejecución inicial]
END [nombre];
```

Tal y como esta constituido un paquete, sólo las declaraciones hechas en la especificación del paquete son visibles y accesibles desde fuera del paquete (por otras aplicaciones o procedimientos almacenados) quedando los detalles de implementación del cuerpo del paquete totalmente ocultos e inaccesibles desde el exterior. Las cabeceras de las funciones y procedimientos en la parte de especificación del paquete tienen que ser idénticas a las del cuerpo.

Previamente a compilar el cuerpo y ejecutar sus procedimientos y funciones, hay que compilar la cabecera. Después de compilar la especificación y el cuerpo de un paquete (con la instrucción START), quedan almacenados en el diccionario de datos, y se pueden invocar desde otros bloques PL/SQL, con lo que los paquetes permiten disponer de variables globales. A la hora de actualizar el valor de una variable global desde cualquier bloque PL/SQL tenemos dos opciones: acceder directamente a la variable global definida en el paquete o ejecutar el procedimiento del paquete que la actualiza.

Cuando se quiera acceder a las funciones, procedimientos y variables de un paquete se debe anteponer el nombre de este:

Nombre_paquete.funcion(x)
Nombre_paquete.procedimiento(x)
Nombre_paquete.variable
Nombre_paquete.cursor
Nombre_paquete.tipo
Nombre_paquete.excepción

Esta notación es válida tanto para referenciar desde procedimientos o triggers externos al paquete como desde aplicaciones escritas en otros lenguajes o desde herramientas como el Sql*Plus, sin embargo, si queremos referenciar objetos desde el interior del paquete donde han sido declarados no es necesario especificar el nombre del paquete y pueden (deberían) ser referenciados directamente por su nombre. Los paquetes no pueden ser llamados o pasados como parámetros.

Opcionalmente tenemos la posibilidad de incluir una sección de inicialización en la parte declarativa del cuerpo del paquete, en ella podemos inicializar variables utilizadas en el paquete. La sección de inicialización sólo se ejecuta la primera vez que una aplicación referencia a un paquete, es decir, sólo una vez por sesión.

Oracle define los siguientes paquetes de funciones predefinidas:

- **DBMS_STANDARD** y **STANDARD:** Estos paquetes permiten el uso de las funciones SQL dentro de programas PL/SQL.

Eje. To_char(), sin(), etc. Además incluyen una serie de funciones como: RAISE_APPLICATION_ERROR (-20200, 'Valor inválido) y funciones de uso específico desde los disparadores.
Todas las funciones que incluyen pueden ser invocadas sin anteponer el nombre del paquete.

- **DBMS_OUTPUT**: Orientado a facilitar la verificación de los programas PL/SQL.
- **DBMS_PIPES:** Proporciona módulos para enviar/recibir operaciones entre procesos en una misma instancia.
- **DBMS_ALERT:** Proporciona recursos para la intercomunicación de procesos en una misma instancia.

- **DBMS_LOCK:** Para gestionar bloqueos entre procesos que acceden a los mismos recursos.
- **DBMS_TRANSACTION:** Manejo de transacciones desde PL/SQL.
- **DBMS_SESSION**: Implementan el comando Alter Session.
- **DBMS_SNAPSHOT:** Procedimientos para refrescar los Snapshot (instantáneas).
- **DBMS_UTILTY:** Utilidades de DBA.
- **DBMS_DDL:** Acceso limitado al DDL desde procedimientos almacenados.
- **DBMS_SQL:** Soporte de PL/SQL dinámico.

7.16 Ejemplos

1.-Crear un paquete para albergar un contador y una operación de reset.

```
CREATE OR REPLACE PACKAGE paquete_cont IS
g_cont NUMBER := 0; -- se inicializa la variable global
PROCEDURE reset_cont (v_nuevovalor IN NUMBER);
END paquete_cont;

CREATE OR REPLACE PACKAGE BODY paquete_cont IS
PROCEDURE reset_cont (v_nuevovalor IN NUMBER) IS
BEGIN
g_cont := v_nuevovalor;
END reset_cont;
END paquete_cont;
```

Ahora para actualizar el valor de la variable global desde cualquier bloque PL/SQL tenemos dos opciones: acceder directamente a la variable global definida en el paquete o ejecutar el procedimiento del paquete que también la actualiza.

```
EXECUTE paquete_cont.g_cont:=5;
EXECUTE dbms_output.put_line(paquete_cont.g_cont);
EXECUTE paquete_cont.reset_cont(5);
EXECUTE dbms_output.put_line(paquete_cont.g_cont);
```

2.-Crear un paquete con dos operaciones: dar de alta a un empleado y dar de baja a un empleado.

```
CREATE OR REPLACE PACKAGE accion_emp IS
 PROCEDURE alta_emp (emp# NUMBER, nombre CHAR, ...);
 PROCEDURE baja_emp (emp_id NUMBER);
END accion_emp;

CREATE OR REPLACE PACKAGE BODY accion_emp IS
```

```
 PROCEDURE alta_emp (emp# NUMBER, nombre CHAR, ...) IS
 BEGIN
   INSERT INTO emp VALUES (emp#, nombre, ...);
 END alta_emp;

 PROCEDURE baja_emp (emp_id NUMBER) IS
 BEGIN
   DELETE FROM emp WHERE emp# = emp_id;
 END baja_emp;
END accion_emp;
```

7.17 ¿Qué es un objeto?

Un tipo de objeto es similar a un paquete con una especificación y un cuerpo. La especificación del tipo contiene los atributos y las declaraciones formales para los métodos, el cuerpo contiene el código real de los métodos, así, podemos decir que la definición de un método consta de dos partes diferenciadas y separadas: Especificación e implementación.

Oracle presenta un tipo de programación orientada a objetos a partir de la versión 8.

7.18 Creación de Objetos

7.18.1 Métodos asociados a tipos de objetos

La definición de métodos en un tipo de objeto tiene distintas aplicaciones:

- definir una relación de orden para el tipo de objeto, lo que permite al sistema ordenar las instancias del tipo.
- definir atributos derivados, cuyo valor no se almacena sino que se calcula aplicando una regla de derivación.
- definir operaciones de manipulación, que permiten actualizar la base de datos de forma consistente.

Por ejemplo podemos crear un objeto para el tipo Estudiantes de la siguiente manera:

```
CREATE OR REPLACE TYPE EstudianteObj AS OBJECT (
ID NUMBER(5),
nombre VARCHAR2(20),
apellidos VARCHAR2(30),
```

```
principal VARCHAR2(30),
Creditos_actuales number(3));
```

Al igual que cualquier otra variable PL/SQL, un objeto se declara simplemente incluyéndolo en la sección declarativa del bloque, para inicializarlo se requiere que esté asociado con una clase de objeto en particular. Los objetos tienen que estar instalados, y para saber si están instalados cuando entremos a SQL tendremos que ver lo siguiente:

```
Oracle8 Enterprise Edition....
With the Partitioning and Objects Option
...
```

Un constructor es una función que devuelve un objeto inicializado y toma como argumentos los valores de los atributos del objeto.

```
DECLARE
- Crea una instancia del objeto con unos atributos.
v_estudiante EstudianteObj :=
EstudianteObj(10020, 'Jose', 'Millán', NULL, 0);
BEGIN
- Modifica el principal atributo a 'Musica'.
v_estudiante.principal := 'Musica';
END;
/
```

Los métodos se declaran en la especificación del tipo de objeto, detrás de los atributos y se implementan en el cuerpo del tipo. La especificación incluye la indicación del tipo de método (función o procedimiento), su nombre, sus argumentos y tipos asociados, y el valor devuelto por el método en caso de que este sea de tipo función.

La sintaxis para la especificación es:

```
especificación_método := { especificación_método_ordenación |
especificación_método_general}

especificación_método_ordenación :=
[MAP | ORDER] MEMBER especificación_función
[directivas_compliación]

especificación_método_general :=
MEMBER {especificación_función
|especificación_procedimiento}[directivas_compliación]

especificación_función :=
```

```
FUNCTION nombre_función (parámetros) RETURN tipo_datos

especificación_procedimiento :=
PROCEDURE nombre_procedimiento (parámetros)

directivas_compilación :=
PRAGMA RESTRICT_REFERENCES(nombre_método, lista_opción)
opción:= {WNDS | WNPS | RNDS | RNPS}
```

La implementación de un método se incluye en la implementación del tipo:

```
CREATE TYPE BODY nombre_tipo AS especificación_método IS
bloque_PL/SQL
```

Los métodos de ordenación *(*MAP, ORDER) permiten definir una relación de orden para los objetos de un tipo. Esta relación será utilizada por el sistema para ordenar las instancias del tipo.
Los métodos de ordenación de tipo MAP, definen la relación de orden basándose en el valor de algunos de los atributos del objeto y en la relación de orden ya definida sobre los tipos de datos básicos. Estas funciones de ordenación no llevan argumentos y devuelven un valor de un tipo básico que es calculado a partir de los valores de uno o varios atributos del objeto para el que se invoca la función. El sistema ordena las instancias del objeto según el valor devuelto por esta función. La relación de orden definida por la función MAP es invocada implícitamente por el sistema en la evaluación de comparaciones y en la ejecución de la cláusula ORDER BY de la sentencia SELECT.
Los métodos de ordenación de tipo ORDER, definen la relación de orden de forma explícita. Estas funciones reciben como argumento una instancia del tipo de objeto y lo comparan, según la lógica de la función, con la instancia del objeto para el cual se invoca el método, el resultado de la comparación se devuelve en forma de valor entero: un valor negativo, cero o un valor positivo, indicando respectivamente que la instancia implícita (instancia para la que se invoca el método) es menor, igual o mayor que el objeto recibido como parámetro.
Las directivas de compilación son directivas que evitan que los métodos manipulen la base de datos o las variables del paquete PL/SQL. Es necesario definir directivas junto con la especificación de cualquier método. Tienen el siguiente significado:

· WNDS: (writes no database state) no se permite al método modificar las tablas de la base de datos.
· WNPS: (writes no package state) no se permite al método modificar las variables del paquete PL/SQL.
· RNDS: (reads no database state) no se permite al método leer las tablas de la base de datos.
· RNPS: (reads no package state) no se permite al método leer las variable del paquete PL/SQL.

Veamos un ejemplo:

```
CREATE OR REPLACE TYPE EstudianteObj AS OBJECT (
ID NUMBER(5),
nombre VARCHAR2(20),
apellidos VARCHAR2(20),
principal VARCHAR2(30),
creditos_actuales NUMBER(3),

-- Retorna el nombre y los apellidos  separados por espacios.
MEMBER FUNCTION FormatoNombre

RETURN VARCHAR2,
PRAGMA RESTRICT_REFERENCES(FormatoNombre, RNDS, WNDS, RNPS, WNPS),

-- Actualiza el principal con el valor especificado en p_NuevoPrincipal.
MEMBER PROCEDURE CambiarPrincipal(p_NuevoPrincipal IN VARCHAR2),
PRAGMA RESTRICT_REFERENCES(CambiarPrincipal, RNDS, WNDS, RNPS, WNPS),

-- Actualiza creditos_actuales sumándole el numero de créditos de p_CompletadaClase actual
MEMBER PROCEDURE ActualizaCreditos(p_CompletadaClase IN ClaseObj),
PRAGMA RESTRICT_REFERENCES(ActualizaCreditos, RNDS, WNDS, RNPS, WNPS),

-- ORDER función usada para ordenar estudiantes.
ORDER MEMBER FUNCTION OrdenarEstudiantes(p_estudiante IN EstudianteObj)
RETURN NUMBER
);
/
```

Para llamar a un método tendremos que usar la sintaxis

```
nombre_objeto.nombre_método
```

Cuestiones importantes a recordar respecto a los objetos:

- Todos los métodos tienen que ir precedidos de la palabra clave MEMBER en la declaración formal.
- En lugar de un punto y coma al final de cada declaración se incluye una coma.
- Las declaraciones de métodos deben hacerse después de las declaraciones de atributos.

- La cláusula RESTRIC_REFERENCES puede usarse para permitir que se llame a un método desde una orden SQL.

Los tipos de objetos son muy similares a los paquetes en muchos detalles:

- Ambos favorecen la abstracción de datos, separando la definición y el cuerpo del objeto en diferentes diccionarios de datos.
- Ambos crean objetos en el diccionario de datos.

Sin embargo existen diferencias significativas:

- Un cuerpo de paquete puede incluir declaraciones adicionales que no estén en la especificación.
- Los tipos de objetos son tipos PL/SQL.
- Los paquetes pueden contener una sección de inicialización, los objetos no.

Un cuerpo de un tipo de objeto sólo puede contener la implementación de los métodos indicados en la especificación, sin embargo existen diferencias significativas:

- Un cuerpo de paquete puede incluir declaraciones adicionales que no estén en la especificación.
- Los tipos de objetos son tipos PL/SQL.
- Los paquetes pueden contener una sección de inicialización, los objetos no.
- Un cuerpo de un tipo de objeto sólo puede contener la implementación de los métodos indicados en la especificación.

7.19 ¿Qué son los disparadores?

Un trigger define una acción que se ejecuta cuando ocurre un cierto evento en la base de datos. Oracle permite definir procedimientos llamados TRIGGERS que se ejecutan de manera automática cuando una estructura INSERT, UPDATE o DELETE es empleada sobre una tabla e incluso en algunas ocasiones sobre una vista; así mismo, pueden estar asociados a eventos que ocurran sobre la Base de datos.
Estos procedimientos almacenados en la Base de Datos, pueden ser escritos en PL/SQL, Java o implementados en C como llamadas externas.
Un TRIGGER difiere en gran medida de un procedimiento almacenado en la BD ya que unos son ejecutados de manera explícita por el usuario (una aplicación) y los otros son ejecutados de manera implícita por Oracle ante la ocurrencia de algún evento.

7.20 ¿Para que sirven los disparadores?

Los triggers se utilizan para extender las restricciones en la base de datos o para supervisar los cambios que ocurren sobre los datos.

Son ampliamente empleados en la personalización de la administración de la BD, por ejemplo, un TRIGGER puede restringir las operaciones DML sobre una tabla en particular en ciertas horas del día, durante una o varias semanas o incluso meses.

Otras aplicaciones de los TRIGGERS pueden ser las siguientes:

- Generación automática de valores derivados de una columna.
- Prevenir transacciones inválidas.
- Proporcionar auditorias sofisticadas.
- Mantener la sincronía en tablas replicadas.
- Generar estadísticas de acceso.
- Modificar los valores de una vista.
- Publicar información de los eventos generados por la BD, las actividades de los usuarios o de las estructuras SQL que se ha ejecutado.

7.21 Creación de disparadores.

Para crear un disparador en Oracle utilizaremos la siguiente sintaxis:

```
CREATE [OR REPLACE] TRIGGER trigger
{BEFORE | AFTER | INSTEAD OF}
[DELETE] [OR INSERT] [OR UPDATE [OF columna [,columna]]] ON
tabla | vista
[FOR EACH {STATEMENT | ROW [WHEN condición]}]
[DECLARE]
...
BEGIN
...
[EXCEPTION]
...
END;
```

Como podemos ver en la sintaxis un trigger esta compuesto por:

1. **Nombre del disparador:** es el conjunto de identificadores válidos que pueden emplearse como nombres de un objeto. Los disparadores existen en un espacio de nombre diferente con lo cual pueden tener el mismo nombre que una tabla o un procedimiento, aunque no puede existir dos triggers con el mismo nombre dentro de un mismo esquema.

2. **Tipos de disparadores:** implica cuando se va a ejecutar este disparador, podemos tener doce tipos posibles de trigger: 3 órdenes (insert, delete o update), dos opciones de temporización (before o update) y 2 niveles (a nivel de fila, se caracterizan porque llevan la cláusula FOR EACH ROW o a nivel de orden FOR EACH STATEMENT).

Los tipos de eventos que se pueden supervisar con triggers y que hacen que se active el trigger son:

a) **INSERT :** Al insertar un dato.
b) **DELETE:** Al borrar un dato.
c) **UPDATE:** Al actualizar un dato.

En función del momento en que se ejecuta (temporización):

- **BEFORE:**
  Se ejecutan inmediatamente antes de la acción que los dispara.
- **AFTER:**
  Se ejecutan inmediatamente después de la acción que los dispara.
- **INSTEAD OF:**
  Se ejecutan en vez del evento que los dispara, son siempre de tipo for each row y sólo se pueden aplicar a vistas. Se aplica para ejecutar operaciones sobre las tablas subyacentes a una vista, por ejemplo, cuando se define una vista con varias tablas, Oracle no permite que la vista se modifique directamente, pero se puede implementar un disparador que se dispare en lugar de la operación de actualización de la vista, ejecutando las modificaciones adecuadas sobre las tablas que alimentan la vista.

Para diferenciar dentro del bloque PL/SQL cuál de los posibles sucesos es el que ha activado al disparador, se pueden utilizar los predicados condicionales INSERTING, UPDATING y DELETING.

```
IF INSERTING THEN
v_valor := 'I';
ELSIF UPDATING THEN
v_valor := 'U';
ELSE
v_valor := 'D';
END IF;
```

A continuación va la palabra ON y el nombre de la tabla o vista a la que se asocia el disparador.

Los dos niveles posibles son:

- **FOR EACH STATEMENT:** El trigger se ejecuta una sola vez, antes o después de la orden que lanza su ejecución**.** Por defecto todos los disparadores son de tipo `for each statement`.
- **FOR EACH ROW:** Especifica que el trigger se debe ejecutar para cada fila afectada por el evento. Hay que tener en cuenta que desde el cuerpo de los disparadores de nivel de fila (for each row) no es posible ejecutar órdenes SQL de lectura o actualización sobre la tabla asociada al disparador, ni sobre las claves de las tablas

referenciadas desde la tabla asociada al disparador por medio de una clave ajena, Oracle bloquea el acceso a estos datos durante la ejecución del disparador para poder asegurar la integridad de la tabla (error de las tablas mutantes).

Tablas mutantes

Hay ciertas restricciones sobre las tablas y columnas a las que puede acceder el cuerpo de un disparador. Para poder definir estas restricciones es necesario entender los conceptos de tablas mutantes y de tabla de restricción.
Una tabla mutante es una tabla que está modificándose actualmente por una sentencia DML, para un disparador es la tabla a la que este está asignado.
Una tabla de restricción es una tabla de la que puede ser necesario leer para una restricción de integridad referencial. Si creamos una tabla, con dos restricciones de integridad referencial, y creamos un trigger sobre esta tabla, tenemos que tener en cuenta que las órdenes SQL en el cuerpo del disparador no pueden leer o modificar ninguna tabla mutante de la orden que provoca el disparo, ni leer o modificar las columnas de clave primaria, única o externa de una tabla de restricción de la tabla del disparador.

3. **Cuerpo del disparador:** es un bloque PL/SQL, pero tiene una serie de restricciones, ya que no podemos emitir ninguna orden de control de transacciones (commit, rollback o savepoint).

Existen dos variables externas old y new que almacenan respectivamente los valores de cada tupla antes y después de ejecutar sobre ella la instrucción que lanza al disparador. Estas variables solamente se pueden utilizar cuando se escoge la opción de ejecutar el disparador una vez por cada fila (for each row). En estos disparadores con nivel de fila podemos utilizar la cláusula WHEN, en este caso la cláusula optativa WHEN sirve para especificar una condición adicional que debe cumplir una tupla afectada por el disparador para que este sea ejecutado sobre ella.
Las variables externas old y new deben ser precedidas por dos puntos para que puedan ser accedidas desde dentro del bloque PL/SQL, sin embargo, si las utilizamos dentro de la clausula WHEN, no debemos poner los dos puntos.
Del párrafo anterior podemos concluir que las variables old, new y la clausula WHEN no se utilizan con **FOR EACH STATEMENT.**

Un mismo disparador puede tener varios sucesos asociados, en cambio una instrucción SQL sólo se corresponde con un suceso, aunque afecte a varias tuplas. Los triggers pueden referenciar tablas distintas de aquella cuyas modificaciones disparan el trigger. Para crear un trigger sobre una tabla son necesarios privilegios de modificación sobre dicha tabla y de creación de triggers.

Para activar un disparador se tiene que compilar con éxito (instrucción start), y se puede desactivar con la instrucción siguiente:

```
ALTER TRIGGER nombre_disparador [DISABLE|ENABLE]
```

Para borrar un disparador se ejecuta la acción:

```
DROP TRIGGER nombre_disparador
```

Para consultar la información de los disparadores se puede ejecutar la siguiente consulta sobre el diccionario de datos:

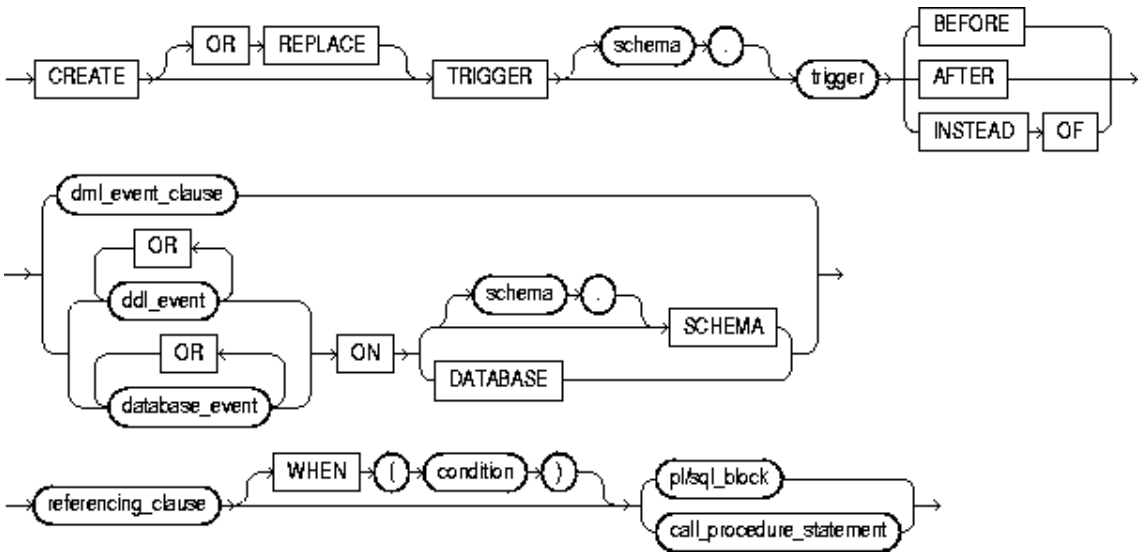
```
SELECT trigger_type, table_name, triggering_event
FROM user_triggers
WHERE trigger_name = '...';
```

Es posible obtener los errores asociados con un trigger:

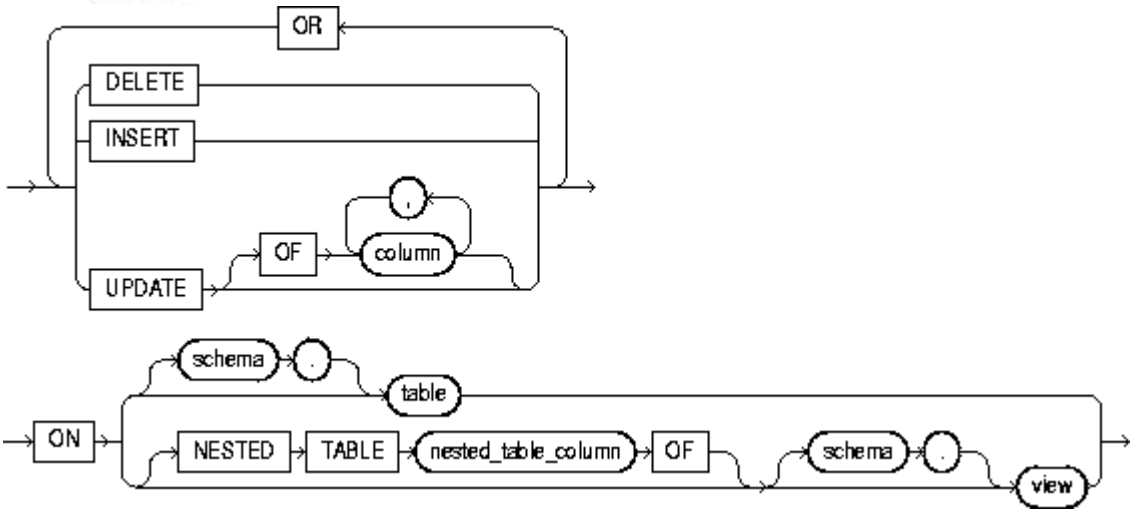
```
Show errors trigger <trigger_name>;
```

Cuando se utilicen bases de datos con varios disparadores activados hay que tener en cuenta que éstos pueden interaccionar entre sí, esto significa, que una acción del usuario puede lanzar un disparador que a su vez ejecute acciones que lancen otros disparadores, y así sucesivamente.
También es importante no definir conjuntos de disparadores que puedan lanzarse mutuamente en un bucle infinito, ni que se contradigan entre sí.

CREATE TRIGGER en Oracle 8:



Dml_event_clause



7.22 Ejemplos

EJEMPLO 1: Dadas las tablas1 y tabla2, insertar una tupla en tabla2 cada vez que se inserta en tabla1:

```
CREATE TABLE TABLA1 (a INTEGER, b CHAR(10));
CREATE TABLE TABLA2 (c CHAR(10), d INTEGER);
...
CREATE TRIGGER trig
      AFTER INSERT ON TABLA1
      FOR EACH ROW
      WHEN (NEW.a <=10)
      BEGIN
            INSERT INTO TABLA2 VALUES (:NEW.b, :NEW.a);
      END trig;
.
run;
```

EJEMPLO 2:

```
CREATE OR REPLACE TRIGGER ChequearCreditos
BEFORE INSERT OR UPDATE OF creditos_actuales ON estudiantes
FOR EACH ROW
WHEN (new.creditos_actuales>20)
BEGIN
......
END;
```

EJEMPLO 3: Queremos mantener una serie de estadísticas referentes a las diferentes especialidades en que se matriculan los alumnos. Usaremos estos datos almacenándolos en la tabla de principales_esp, y para actualizar las estadísticas de esta tabla podríamos generar un trigger que lo que haría sería modificar esta tabla cada vez que se modifique la tabla de estudiantes, el trigger quedaría de la siguiente manera:

```
CREATE OR REPLACE TRIGGER UpdatePrincipalEsp
/* Mantiene la tabla principales_esp actualizada con los
cambios hechos en la tabla de estudiantes. */
AFTER INSERT OR DELETE OR UPDATE ON estudiantes
DECLARE
CURSOR c_Estadisticas IS
SELECT principal, COUNT(*) total_estudiantes,
SUM(creditos_actuales) total_creditos
FROM estudiantes
GROUP BY principal;
BEGIN
/* Bucle por cada principal. Actualiza las estadísticas in
principales_esp correspondientes a esa principal. Si la fila
no existe la crea. */

FOR v_RegistroEsp in c_Estadisticas LOOP
UPDATE principales_esp
SET total_creditos = v_RegistroEsp.total_creditos,
total_estudiantes = v_RegistroEsp.total_estudiantes
WHERE principal = v_RegistroEsp.principal;
/* Chequea si la fila existe. */
IF SQL%NOTFOUND THEN
INSERT INTO principales_esp (principal, total_creditos,
total_estudiantes)
VALUES (v_RegistroEsp.principal, v_RegistroEsp.total_creditos,
v_RegistroEsp.total_estudiantes);
END IF;
END LOOP;
END UpdatePrincipalEsp;
/
```

Este trigger se divide en:
1. Creación del trigger:

```
CREATE OR REPLACE TRIGGER UpdatePrincipalEsp
AFTER INSERT OR DELETE OR UPDATE ON estudiantes
```

2. Declaración de las variables que formarán parte del trigger, estas variables almacenarán los datos antes de pasarlos a la tabla definitiva que sería la de principales_esp, en este caso los datos se almacenarán en un cursor:

```
DECLARE
CURSOR c_Estadisticas IS
SELECT principal, COUNT(*) total_estudiantes,
SUM(creditos_actuales) total_creditos
FROM estudiantes
GROUP BY principal;
```

3. Iniciaremos el proceso de lo que va a tener que realizar este trigger:

```
BEGIN
/* Hacemos un bucle para cada principal, intentaremos
actualizar las estadísticas de la tabla principales_esp con el
registro correspondiente (v_RegistroEspPrin), y si no existe
insertaremos la fila.*/
FOR v_RegistroEsp in c_Estadisticas LOOP
UPDATE principales_esp
SET total_creditos = v_RegistroEsp.total_creditos,
total_estudiantes = v_RegistroEsp.total_estudiantes
WHERE principal = v_RegistroEsp.principal;
/* Chequea si la fila existe. */
IF SQL%NOTFOUND THEN
INSERT INTO principales_esp (principal, total_creditos,
total_estudiantes)
VALUES (v_RegistroEsp.principal, v_RegistroEsp.total_creditos,
v_RegistroEsp.total_estudiantes);
END IF;
END LOOP;
END UpdatePrincipalEsp;
```

EJEMPLO 4:

```
CREATE OR REPLACE TRIGGER Sal_Total
AFTER INSERT OR UPDATE OF salario ON empleado
FOR EACH ROW WHEN (new.dep_num IS NOT NULL)
/*El disparador solamente se va a ejecutar para aquellas
tuplas
insertadas o modificadas que tengan un dep_num no nulo*/
BEGIN
UPDATE departamento
SET sal_total= sal_total + :new.salario
WHERE dep_num = :new.dep_num;
END;
```

EJEMPLO 5:

```
CREATE OR REPLACE TRIGGER hacer_pedido
/*Disparador para vigilar que cuando la cantidad de piezas
caiga por
debajo del mínimo se haga un pedido de más piezas, si no se ha
hecho ya*/
AFTER UPDATE OF cantidad ON piezas
FOR EACH ROW WHEN (new.cantidad < new.cant_min)
DECLARE
v_pendientes NUMBER;
BEGIN
SELECT count(*) INTO v_pendientes FROM pedidos
WHERE numpie = :new.numpie;
IF v_pendientes = 0 THEN
INSERT INTO pedidos VALUES (:new.numpie, :new.cant_pedido,
SYSDATE);
END IF;
END;
```

EJEMPLO 6:

```
CREATE TRIGGER triggerUno
AFTER INSERT ON tabla1
BEGIN
INSERT INTO tabla2 VALUES ('T_mod', :new.a);
END;
/
```

EJEMPLO 7:

```
CREATE TRIGGER triggerDos
AFTER INSERT ON tabla1
BEGIN
INSERT INTO tabla2 VALUES ('T_mod', 1);
END;
/
```

EJEMPLO 8:

```
CREATE TRIGGER triggerTres
BEFORE INSERT OR UPDATE OF a ON tabla1
FOR EACH ROW
BEGIN
:new.a := :new.a * 166.386;
END;
/
```

EJEMPLO 9:

```
CREATE TRIGGER triggerCuatro
BEFORE INSERT OR UPDATE OF a ON tabla1
FOR EACH ROW
BEGIN
:new.b:= UPPER(:new.b);
END;
/
```

EJEMPLO 10:

```
CREATE TRIGGER triggerCinco
BEFORE INSERT OR UPDATE OF a ON tabla1
FOR EACH ROW
DECLARE var CHAR(1);
Error_tipo EXCEPTION;
BEGIN
SELECT b INTO var
FROM tabla1 WHERE b<>'A';
IF var='D' THEN
     RAISE error_tipo;
END IF;
EXCEPTION
WHEN error_tipo THEN
     Raise_aplication_error (-20001, 'Tipo D');
END;
/
```

EJEMPLO 11:

```
CREATE TRIGGER checa_salario
BEFORE INSERT OR UPDATE sal, trabajo ON prueba
FOR EACH ROW
WHEN (new.trabajo != 'PRESIDENT');
DECLARE
minsal NUMBER;
maxsal NUMBER;
BEGIN
SELECT bajosal, altosal, INTO minsal, maxsal FROM sals;
IF (:new.sal < minsal OR :new.sal > maxsal) THEN
raise_aplication_error(-20225,'Salario fuera de Rango');
END IF;
END;
```

7.23 Entornos de desarrollo de PL/SQL

FROG	http://www.507pm.com/pcs
Oracle Procedure Builder	http://www.oracle.com
SQL/Expediter	http://www.compuware.com
SQL Navigator	http://www.quests.com
SQL*Object Builder	http://www.idb-consulting.fr
SQL Programmer	http://www.sfi-software.com
SQL Station	http://www.platinum.com
CAST Workbench	http://www.castsofware.com
PL/Formatter	http://www.revealnet.com
PLEdit	http://www.benthicsoftware.com

8 FINALIZANDO EL EJEMPLO.

Una vez ya hemos generado el código SQL de nuestro ejemplo (Apéndice E), ¿Finaliza el diseño de nuestra base de datos?. La respuesta es no. Después de implementar nuestro esquema relacional debemos introducir aquellas funcionalidades y reglas de integridad que se le exigen a nuestra base de datos. Para ello iremos extendiendo el esquema por pasos. A partir de este punto vamos a presentar elementos ajenos al foco de este módulo pero que pueden ser utilizados como introducción a la programación en gestores de bases de datos relacionales, pues manipulan conceptos como procedimientos almacenados y triggers. Todo este código será desarrollado utilizando PL/SQL de Oracle.

En primer lugar se nos informa que con cierta frecuencia puede ocurrir que dos muestras de una misma cepa sean identificadas inicialmente por error como distintas. Así que debemos añadir un procedimiento que dada dos cepas las fusione en una (dada dos cepas la primera indicada es la correcta y la segunda la errónea). El procedimiento necesario para realizar esta operación es sencillo:

```
create or replace procedure SonMismaCepa(Cepa1 number, Cepa2 number) as
begin
  update resistencia set codcepa = Cepa1 where codcepa = Cepa2;
  update muestra set codcepa = Cepa1 where codcepa = Cepa2;
  update cepa set codcepa = Cepa1 where codcepa = Cepa2;
end;
/
```

En segundo lugar se nos informa de que se desea trazar la evolución geográfica del contagio de una cepa en concreto. Para ello se desea visualizar cronológicamente los primeros casos detectados en cada provincia (con su respectivo país) para una cepa dada. En este caso el procedimiento sería:

```
create or replace procedure EstallidoCepa(CodigoCepa number) as
  cursor provincias is select provincia.nomprovincia, pais.nompais,
min(muestra.fecha) fecha
    from muestra, provincia, pais
    where muestra.codcepa = CodigoCepa
    group by provincia.nomprovincia, pais.nompais
    order by min(muestra.fecha);
begin
  for t in provincias loop
    dbms_output.put_line( t.fecha || ' - ' || t.nomprovincia || ' (' ||
t.nompais || ')' );
  end loop;
end;
/
```

Puesto que usamos el paquete DBMS_OUTPUT de Oracle debemos recordar activar la visualización del servidor (la cual permanecerá activa durante toda la sesión o hasta que se desactive específicamente):

SET SERVEROUTPUT ON;

En tercer lugar es de vital importancia localizar los microorganismos multiresistentes que para nuestro ejemplo vamos a definir como todas aquellas cepas que han dado siempre como resultado un antibiograma resistente. Para ello se nos pide que se muestre el género, la especie y el código de cepa ordenados alfabéticamente.

```
create or replace view MultiResistentes as
  select distinct genero.nomgenero, especie.nomespecie, cepa.codcepa
    from cepa, especie, genero, resistencia
    where cepa.codcepa = resistencia.codcepa and
          especie.idespecie = cepa.idespecie and
          genero.idgenero = especie.idgenero and
          cepa.codcepa not in
          (
          select resistencia.codcepa
           from resistencia, cepa, patronres
           where resistencia.codcepa = cepa.codcepa and
                 cepa.idespecie = patronres.idespecie and
                 resistencia.cmi < patronres.cmi
          )
    order by genero.nomgenero, especie.nomespecie, cepa.codcepa;
/
```

En último lugar se pide la detección automática de toda nueva resistencia de una especie. Para ello el sistema debe detectar la aparición de una nueva resistencia de una especie sobre un antibiótico, indicando la fecha, centro, provincia y país donde ocurre. Los mensajes de advertencias serán almacenados en una tabla que contendrá dos atributos. El primero de ellos es la fecha de la muestra y el segundo el mensaje de advertencia, el cual tendrá el siguiente formato:

```
Se ha detectado una nueva resistencia sobre la especie Enterococcus fecalis
al antibiótico Vancomicina en H.R. Carlos Haya (Málaga) – España.
```

Para dar solución a este problema debemos crear en primer lugar la tabla de advertencias que inicialmente no se especificaba en el problema. Seguidamente la resolución de este tipo de problemas se basa en la construcción de un trigger sobre la tabla “resistencia”. Un trigger es un bloque de código que es ejecutado de forma automática por el sistema ante condiciones particulares. El trigger ante el que no encontramos necesitará además acceder a los datos de dicha tabla y deberá ser activado para cada inserción nueva. Esto nos motivará la aparición de un problema de mutación de tablas (en oracle un trigger de fila -cláusula FOR EACH ROW- no puede acceder directamente a la tabla sobre la que está definido) Para ello debemos

utilizar una combinación de triggers de bloque y triggers de fila, utilizando una tabla temporal.

```
-- tabla de advertencias
create table advertencias
 (
    fecha date,
    mensaje varchar2(255)
 );


-- tabla temporal para almacenar el código de las cepas de las nuevas
-- resistencias
create global temporary table modificados
 (
    id number(6)
 );


-- primer trigger que asegura que la tabla modificados esté vacía
create or replace trigger PrimeraCepaResistenteB before insert on
resistencia
begin
  delete from modificados;
end;
/

-- trigger de fila que almacena los códigos de las cepas de las nuevas
-- resistencias
create or replace trigger PrimeraCepaResistenteR after insert on
resistencia for each row
begin
  insert into modificados values(:new.codcepa);
end;
/

-- vista auxiliar utilizada por el trigger principal que nos indica
-- aquellos pares de resistencia (especie, antibiótico) donde
-- únicamente existe una resistencia en la base de datos
create or replace view PrimerosResistentes as
  select cepa.idespecie, resistencia.idantibiotico
    from cepa, resistencia, patronres
    where cepa.codcepa = resistencia.codcepa and
          cepa.idespecie = patronres.idespecie and
          -- nos saltamos la composicion por especie (no es necesaria)
          resistencia.cmi >= patronres.cmi
          -- nos da los resistentes
    group by cepa.idespecie, resistencia.idantibiotico
    having count(*) = 1;
/

create or replace trigger PrimeraCepaResistenteA after insert on
resistencia
begin
  -- tras la inserción borramos aquellas cepas que no son
```

```
  -- "PrimerosResistentes"
  delete from modificados
    where id not in
     (
      select cepa.codcepa
       from cepa, resistencia
       where cepa.codcepa = resistencia.codcepa and
            (cepa.idespecie, resistencia.idantibiotico) in
            (select idespecie, idantibiotico from PrimerosResistentes)
     );
  -- insertamos las advertencias
  insert into advertencias
    select muestra.fecha,
           'Se ha detectado una nueva resistencia sobre la especie '
           || genero.nomgenero || ' ' || especie.nomespecie ||
           ' al antibiotico ' || antibiotico.nomantibiotico ||
           ' en ' || centro.nomcentro ||
           ' (' || provincia.nomprovincia || ') - ' || pais.nompais
      from muestra, cepa, especie, genero, centro, provincia,
           pais, resistencia, antibiotico, modificados
      where modificados.id = cepa.codcepa and
            muestra.codcepa = cepa.codcepa and
            cepa.idespecie = especie.idespecie and
            especie.idgenero = genero.idgenero and
            muestra.idcentro = centro.idcentro and
            centro.idprovincia = provincia.idprovincia and
            provincia.idpais = pais.idpais and
            cepa.codcepa = resistencia.codcepa and
            resistencia.idantibiotico = antibiotico.idantibiotico;
  -- finalmente limpiamos la tabla modificados
  -- no sería necesario pues el primer trigger realiza el borrado
  -- pero incluimos el borrado para liberar recursos
  delete from modificados;
end;
/
```

9 BIBLIOGRAFÍA

- Elmasri y Navathe. Fundamentos de los Sistemas de Base de Datos. Tercera Edición. The Benjamin/Cummings Publishing Company Inc.. 2001.
- Henry F. Korth y Abraham Silberschatz. Fundamentos de Bases de Datos. Segunda Edición. MacGraw Hill. 1993.
- C.J. Date. Introducción a los Sistemas de Base de Datos. 7º Edición. Vol I. Addison-Wesley Iberoamericana.
- E. Rivero Cornelio. Bases de Datos Relacionales. Ed. Parninfo. 1988.
- Jeffrey D. Ullman. Principles of Database Systems. Computer Science Press. 1982.
- Jeffrey D. Ullman. Database and Knowledge-Base Systems. Vol. I y II. Computer Science Press. 1988.
- Manuales del Sistema Gestor de Base de Datos Objeto-Relacional Oracle 9i. Oracle Corporation. 2001.

APÉNDICE A: SOBRE LOS SISTEMAS DE BASES DE DATOS. UN POCO DE HISTORIA.

La tecnología de Bases de Datos ha tenido una gran influencia en el creciente uso de los computadores. No nos equivocamos al decir que las Bases de Datos juegan un papel critico en casi todas las áreas donde se usan computadoras. La gestión de Bases de Datos ha pasado de ser una aplicación especializada más a constituir el componente central de un entorno de computación moderno.

Como consecuencia de ello los Sistemas de Base de Datos se han convertido en una parte esencial de la formación en Informática. El término Base de Datos (BD) se ha hecho tan común que no se puede posponer más la necesidad de definirlo. Una primera definición, bastante general pero muy útil, puede establecer que una BD es una colección de Datos lógicamente interrelacionados.

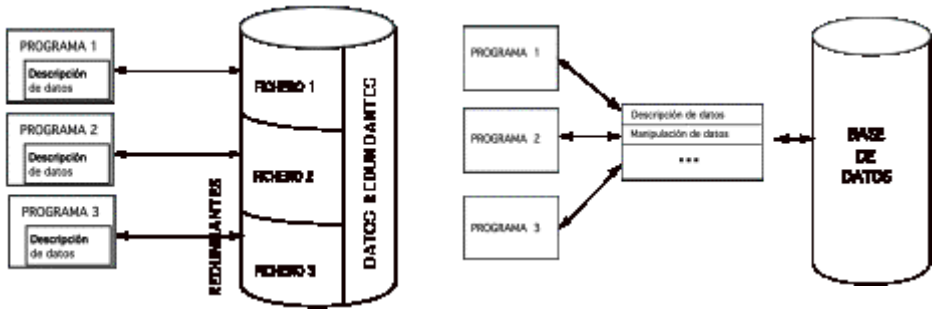


Figura 1a. Figura 1b.

Figura 1. Procesamiento tradicional orientado a Ficheros Vs. Procesamiento orientado a BD.

Los sistemas de Base de Datos nos han llevado desde un paradigma del procesamiento de Datos, en el cual cada aplicación definía y mantenía sus propios datos (figura 1a.) a una en la cual los datos son definidos y administrados de forma integrada (figura 1b.). Esta nueva orientación tiene como consecuencia la independencia de los datos, por la cual los programas de aplicación son insensibles a los cambios en la organización física o lógica de los datos y viceversa.

Sin embargo, la expresión BD no comenzó a popularizarse hasta principios de los años 60. Antes de esa época, en el mundo de la informática se hablaba de ficheros y de conjuntos de Datos. En el origen y la evolución de las Bases de Datos podemos distinguir varias etapas en las que iremos destacando algunos de los acontecimientos principales.

Para superar las limitaciones de los sistemas de ficheros[1] surgen las técnicas de bases de Datos. El objetivo fundamental es unificar toda la información del sistema para evitar redundancias, sin perder las diferentes perspectivas que, de la misma, tienen los distintos usuarios. Los datos se organizan y mantienen como un todo estructurado que no está diseñado para una aplicación concreta, sino que tiende a satisfacer las necesidades de información de toda la organización.

En estos momentos se empiezan a establecer los requisitos actuales para los sistemas de Base de Datos. El software de gestión procura la independencia lógica y física de los datos, permitiendo que exista una visión lógica global independiente de los posibles cambios en las vistas de los programas de aplicación o de la distribución física de los datos. Los datos pueden modificarse sin que se incurra en excesivos costos de mantenimiento. Se suministran medios para que el administrador de Datos actúe como controlador y asegure que la organización de estos es siempre la mejor para los usuarios en general. Se proveen los procedimientos eficaces para controlar el secreto, la seguridad y la integridad de los datos. Las Bases de Datos se diseñan de forma que sean capaces de suministrar de forma eficaz respuestas a tipos de preguntas no previstas por el diseñador. Se facilita la migración de los datos y por último el software suministra un lenguaje para la descripción de Datos para el administrador de los datos, un lenguaje de ordenes para el programador de aplicaciones y a veces un lenguaje de consulta para el usuario.

A finales de los 60, concurrentemente con el desarrollo de los sistemas gestores de ficheros, IBM desarrolla el Modelo de Datos Jerárquico. Con la finalidad de resolver problemas de diseño aeroespacial y de producción se desarrolla Information Management System (IMS). Aparece IMS DB/DC (database/data communication), el primer sistema de Base de Datos de gran escala. Este sistema permitía representar vistas con topología de red sobre las jerarquías. Sobre 1969 IMS dio como resultado un sistema de gestión de Bases de Datos de tipo jerárquico de propósito general: el IMS/l de IBM que constituye la primera familia de sistemas de gestión de Bases de Datos. Aparece una extensión de PL/1 desarrollada por Dodd en GM que se denomina APL (Associative Programming Language). American Airlines e IBM desarrollan SABRE el primer sistema que proporciona acceso a datos compartidos por múltiples usuarios a través de una red de comunicación.

En la década de los 70 la tecnología de Bases de Datos experimenta un rápido crecimiento. Desde 1968 el comité CODASYL establece, inspirado en el modelo de Bachman, las características básicas que debería poseer un sistema de gestión de Bases de Datos y define el

[1] Los principales problemas que presentan los sistemas de ficheros clásicos se pueden resumir en los siguientes puntos:

- **redundancia**: motivada fundamentalmente por el desarrollo de aplicaciones separadas para cada sección o departamento de la empresa. Esto trae consigo un desaprovechamiento de la memoria y la posible aparición de inconsistencias.
- **dependencia**: los programas dependen completamente de los datos. Esto implica poca flexibilidad frente a futuras variaciones en los requisitos de información y un alto coste de mantenimiento de los programas.
- **carencias expresivas**: debida a que los ficheros no pueden representar correctamente todos los objetos que aparecen en el análisis de los problemas del mundo real.
- **insuficientes medidas de seguridad**: relacionadas sobre todo con el control de los accesos concurrentes, la recuperación de ficheros y el control de autorizaciones.

Modelo de Datos en Red. Las primeras recomendaciones del DBTG de CODASYL [CODASYL 69] y [CODASYL 71] fueron implementadas por diferentes casas comerciales y dieron lugar a la segunda familia histórica de sistemas de gestión de Bases de Datos. Algunos sistemas, desarrollados a lo largo de los años 70, que siguen las propuestas de CODASYL son: IDS II de HONEYWELL, DMS-1.100 de UNIVAC, DMS-II de Burroughs, DMS-170 de CDC, IDMS de BF Goodrich, PHOLAS de Philips, DBMS-11 de DIGITAL, etc. Sin embargo, ninguna de estas implementaciones desarrolló completamente las propuestas de la CODASYL. Algunos fabricantes desarrollan sus propios modelos en red y se implementan algunos sistemas integrados de BD y de comunicación de Datos como TOTAL de Cicom System y ENVIRON/1.

El Modelo de Datos en Red siempre tuvo pretensiones de generalización y estandarización, mientras que la familia de sistemas jerárquicos está constituida por una serie de sistemas de gestión de Base de Datos de los que, posteriormente, se obtuvo la abstracción del Modelo de Datos Jerárquico.

Hacia los años 1969-70, E.F. Codd, basándose en el álgebra y en la teoría de conjuntos, propone un nuevo Modelo de Datos llamado modelo Relacional. Se desarrollaron proyectos de investigación que dieron lugar a algunos prototipos entre los que destacan INGRES de la Universidad de Berkeley (1973-75) y SYSTEM R de IBM (1974-77), System 2000 de la Universidad de Austin en Texas, el proyecto Sócrates de la Universidad de Grenoble en Francia y ADABAS de la Universidad técnica de Darmstadt en Alemania. Durante este periodo se desarrollaron diversos lenguajes de consulta: SQUARE, SEQUEL (SQL), QBE y QUEL. Posteriormente aparecieron numerosos sistemas relacionales comerciales, tales como el INGRES de RTI (1980), SQL/DS de IBM (1981), ORACLE de RSI (1981), DB2 de IBM (1983), RDB de DIGITAL (1983), etc.

En 1972 el grupo de interés especial en gestión de Datos de la ACM (Special Interest Group on Management of Data -SIGMOD-) organiza la primera conferencia internacional. En 1975 la fundación para las Bases de Datos muy Grandes (Very Large Data Base -VLDB-) organiza la primera conferencia internacional. En 1976 el grupo de trabajo 2.6 de la Federación Internacional de Sociedades para el Procesamiento de la Información (International Federation of Information Processing Society´s -IFIPS´s-) organiza la primera conferencia internacional. En 1976 Chen presenta el Modelo de Datos Entidad-Relación. En 1977 y 78 aparecen la propuestas de ANSI/SPARC y CODASYL en las que se define una arquitectura en tres niveles para los Sistemas Gestores de Base de Datos que pretende conseguir una completa independencia de los datos.

Durante los años 80 se desarrolla la familia de los "micro" sistemas gestores de Base de Datos. Estos permiten al usuario definir sus propios datos y manipularlos con facilidad y flexibilidad pero carecen del soporte necesario para la gestión de múltiples vistas y accesos concurrentes. Además no garantizan el aislamiento entre programas y datos (la independencia de los datos).

Un estudio realizado por ANSI/SPARC en 1983 observa que a principios de los 80 se han implementado más de 100 sistemas relacionales. En 1985 se publican los preliminares de la

estandarización del SQL y el proceso se completa a principios de los 90. En mundo de los negocios se ve fuertemente influenciado por los lenguajes de cuarta generación (4GLs). A finales de los 80 y principios de los 90 junto con los lenguajes de cuarta generación comienzan a utilizarse técnicas de ingeniería del software asistida por computador (CASE). Comienzan a generarse programas de aplicación completos usando como interface un lenguaje de alto nivel no orientado a la programación. Aparecen los primeros sistemas gestores de Base de Datos completos para microcomputadores.

El creciente auge de las tecnologías de las comunicaciones y de la informática distribuida (redes de computadores) resuelve las insuficiencias técnicas de los sistemas de Base de Datos interconectados en red y convierte en una realidad práctica a los sistemas de Base de Datos Distribuida de propósito general.

El Modelo de Datos relacional se ha constituido en un estándar de facto pero al mismo tiempo ha demostrado sobradamente sus carencias (lenguajes de consulta no Turin-completos; tipos de Datos muy limitados; fuertes carencias expresivas del modelo; etc.) y por lo tanto su incapacidad para gestionar diversos tipos de aplicaciones. Como consecuencia de ello se proponen diversas extensiones de él en forma de nuevos modelos de Datos. Cada una de estas extensiones intenta evitar alguno de los problemas del Modelo de Datos relacional. Dos de las extensiones más extendidas son el uso de la lógica como un Modelo de Datos y el uso de modelos de Datos orientados a objetos.

La informática está alcanzando su madurez como ciencia y con ello está desarrollando un fuerte proceso de convergencia, integración y estandarización. Disciplinas que antes se veían como aisladas descubren sus interrelaciones y comienzan a estudiarse en conjunto. Los Sistemas de Base de Datos no constituyen una excepción a esta tendencia general.
Los Sistemas de Base de Datos (centralizada o distribuida) conectados a redes de computadores tienden a integrar sus modelos de Datos y de gestión propios para construir Sistemas de Múltiples Bases de Datos y Federaciones de Sistemas de Base de Datos que extienden la funcionalidad individual de sus componentes.

Las extensiones del Modelo de Datos relacional son modelos de Datos de muy alto nivel que poseen asociados lenguajes que, sin ser lenguajes de programación de propósito general, son ya Turing-completos. Estas extensiones, que fueron propuestas para resolver algunas de las carencias especificas del modelo relacional, se aprecian como complementarias y sufren una fuerte tendencia integradora.

Las nuevas técnicas y herramientas CASE se han acoplado con éxito especialmente notable en los Sistemas de Base de Datos gracias a las características específicas de estos.
Por otro lado, la misma tecnología de la programación ha ido evolucionando desde conceptos como "datos dentro de los programas" y "programas y datos" hasta otros como "programas dentro de los datos".

La unificación de lenguajes de programación y Bases de Datos se presenta como un nuevo y potente medio para construir aplicaciones orientadas a los datos. Los lenguajes de programación y en general toda la tecnología y teoría sobre compiladores y lenguajes han

sido considerada durante mucho tiempo como una disciplina completamente separada de las Bases de Datos. La combinación de estas dos campos no sólo será beneficiosa sino que empieza ya a manifestarse como necesaria [Saltor 84].

El problema fundamental no consiste en decidir cual será la potencia expresiva o la funcionalidad de este tipo de sistemas ya que es fácil conseguir que un lenguaje de programación o de un Sistema de Base de Datos sea Turing-completo. El nuevo reto es la ingeniería del software, la integración de lenguajes de programación y Bases de Datos permitirá construir, modificar y potenciar aplicaciones software más capaces y que de otra forma no serían posibles.

APÉNDICE B: BASE DE DATOS Y SISTEMA DE GESTIÓN DE LA BASE DE DATOS

Durante su evolución histórica la frontera entre los ficheros y las Bases de Datos no está muy clara. Actualmente la separación conceptual de ambos es total. Las siguientes cuatro características pueden ayudar a distinguir entre un sistema de ficheros integrados y un Sistema de Base de Datos (SBD):

- **Naturaleza autocontenida**: un SBD no contiene sólo la Base se Datos, también posee un directorio o diccionario de Datos que contiene una descripción integrada de todos los datos (estructura de cada fichero, tipo y formato de almacenamiento de cada tipo de Datos, etc.). Esta información contenida en el diccionario de Datos se denomina metadatos.
- **Independencia programas-datos**: En el procesamiento tradicional, orientado a ficheros, la estructura de los ficheros de Datos se encuentra embebida en los programas que los acceden. Cambiar la estructura de un fichero puede suponer cambiar todos los programas que acceden al fichero. En un SBD los programas de acceso se escriben independientemente de cualquier fichero concreto y la información sobre la estructura del fichero se almacena en el diccionario de Datos separada de los programas de acceso a los ficheros.
- **Abstracción de Datos**: Un Sistema Gestor de Base de Datos (SGBD) suministra al usuario una representación conceptual de los datos que omite muchos o todos los detalles acerca de cómo los datos se encuentran almacenados. Un Modelo de Datos es el tipo de abstracción de Datos que proporciona esta representación conceptual. El Modelo de Datos usa conceptos más fáciles de entender y cercanos al mundo real que los relacionados con el almacenamiento de información en computadores. Algunos de estos conceptos son: objetos, propiedades de objetos, e interrrelaciones entre objetos.
- **Gestión de múltiples vistas de usuario**: un SBD posee normalmente múltiples usuarios cada uno de los cuales necesita un uso o vista distinto de los datos.

Inicialmente definimos una BD como una colección de Datos lógicamente interrelacionados. Una BD típica representa algunos aspectos del mundo real y es usada para propósitos específicos por uno o más grupos de usuarios. Un SGBD es el software genérico necesario para implementar y mantener una BD en un sistema informático. La BD junto con el SGBD que la soporta se denomina SBD.

Un SGBD debe presentar una serie de capacidades que aportan las siguientes características, además de las cuatro enunciadas anteriormente, a la BD

- **Redundancia controlada**: En el procesamiento tradicional orientado a ficheros, cada grupo de usuarios mantiene sus propios ficheros gestionados por sus aplicaciones de procesamiento de Datos. Gran parte de los datos se almacena varias veces, de hecho cada grupo de usuarios adicional duplica todos o parte de los datos en sus propios ficheros. Esta redundancia da lugar a múltiples problemas: multiplicación del

esfuerzo, desperdicio de espacio de almacenamiento, y sobre todo inconsistencias. En la aproximación orientada a BD las diferentes visiones que de los datos poseen los distintos grupos de usuarios se integran durante el proceso de diseño. Cada dato se almacenará una sola vez. Esto evita inconsistencias. Cierto tipo de redundancia controlada se introduce en ocasiones en aras de un acceso más eficiente a algunos datos.

- **Compartición de la Base de Datos**: Un SGBD debe permitir a múltiples usuarios acceder simultáneamente a la BD. Esto incluye el control de los accesos concurrentes a los datos y la capacidad de establecer múltiples vistas de la BD.
- **Restricción de accesos no autorizados**: Puesto que la BD es compartida por múltiples usuarios, no todos los datos tienen porque ser accedidos por todos ellos. El Sistema Gestor de la Base de Datos debe proporcionar seguridad y privacidad así como un mecanismo para gestionar autorizaciones de acceso a los datos.
- **Soporte de múltiples interfaces a distintos niveles**: A una BD acceden múltiples usuarios con muy distintos niveles de conocimiento técnico y por ello, debe incluir distintos tipos de interfaces: lenguajes de consulta para usuarios casuales, lenguajes de programación para los programadores de aplicaciones y sofisticados intefaces gráficos, basados en menús o en lenguaje natural para usuarios no técnicos.
- **Representación de relaciones complejas entre los datos**: Un SGBD debe ser capaz de representar una gran variedad de complejas relaciones entre los datos así como de recuperar y actualizar los datos relacionados de forma fácil y eficiente. Esto es esencial siempre que los usuarios deseen identificar y procesar los datos de acuerdo con las relaciones que existen entre ellos.
- **Control y mantenimiento de las restricciones de integridad**: La mayoría de las aplicaciones de BD poseen ciertas restricciones de integridad que se deben mantener en los datos que usan (el tipo más simple de restricción de integridad puede ser la especificación de un tipo para cada dato). Existen restricciones de integridad mucho más complejas que se derivan de la semántica de los datos y la parcela del mundo real que representan. Estas restricciones se especifican durante el diseño de la BD e idealmente serán mantenidas en parte por el Sistema Gestor de la Base de Datos. En algunos casos programas específicos comprueban las actualizaciones para asegurar la integridad semántica.
- **Soporte para la seguridad y la recuperabilidad**: Un SGBD debe ser capaz de recuperarse de fallos hardware y/o software. Un sistema combinado de copia de la información y recuperación permite devolver la BD a un estado consistente después de un fallo en el sistema y continuar la operación de la BD desde el punto en que fue interrumpida.

Además de los objetivos perseguidos por la aproximación orientada a BD existen una serie de beneficios secundarios de esta:

- **Capacidad de forzar estándares** entre los distintos grupos de usuarios.
- **Flexibilidad** para el cambio en la estructura o la inclusión de nueva información sin afectar a los usuarios.

- **Reducción del tiempo de desarrollo de las aplicaciones**, una vez que la BD está en funcionamiento, como consecuencia de los interfaces suministrados por el Sistema Gestor de la Base de Datos.
- **Disponibilidad inmediata de la información actualizada** sin necesidad de procesos específicos adicionales.
- **Economía de escala** al permitir la consolidación de los datos y las aplicaciones, reduciendo el solapamiento existente entre las distintas actividades de procesamiento de Datos en distintos grupos de usuarios.

Una posible definición de BD que intente abarcar sus principales características podría ser la siguiente:

> "Colección de Datos estructurados según un modelo que refleje las entidades, relaciones y restricciones existentes en una parcela del mundo real. Los datos que han de poder ser compartidos y presentados en la mejor forma posible ante diferentes usuarios y aplicaciones que harán distintos usos de ellos. Estos datos deben mantenerse independientes de las aplicaciones; su definición y descripción han de ser únicas estando almacenadas junto con los mismos datos; y deberán estar almacenados sin redundancia perjudiciales o innecesarias. Los tratamientos habrán de conservar la integridad y privacidad de la información y además deberán realizarse de forma que permitan la recuperabilidad de la calidad de la información si esta se deteriora por cualquier causa".

Las dos características que distinguen a los Sistemas Gestores de Base Datos existentes son su arquitectura y el Modelo de Datos subyacente.

Arquitectura de Referencia: Independencia en los Datos

La arquitectura de un sistema define su estructura. Esto significa que los componentes del sistemas están identificados, la función de cada componente está especificada, y las interrelaciones e interacciones entre estos componentes están definidas. Una arquitectura de "referencia" para Sistemas Gestores de Base de Datos es una vista "idealizada" del sistema tal que muchos de los sistemas comerciales disponibles se derivan de ella; sin embargo, servirá como una estructura razonable con la cual podremos comentar los problemas relacionados con los Sistemas Gestores de Base de Datos. Nos centraremos en una arquitectura de referencia que ha generado un interés considerable, esta es la arquitectura ANSI/SPARC.

Al final de 1972, el Comité de Computadores y de Procesamiento de la Información (X3) del Instituto Nacional Americano de Estándares (ANSI) estableció un Grupo de Estudio en los Sistemas de Gestión de Base de Datos bajo los auspicios de su Comité de Planificación de Estándares y Requisitos (SPARC). La misión del grupo de estudio era estudiar la viabilidad de establecer estándares en este área, así como determinar que aspectos se deben estandarizar si ello es factible. El grupo de estudio editó su artículo íntegro en 1975 [SPARC 75], y su

artículo final en 1977 [Tsichritzis & Klug 78]. El marco arquitectónico propuesto en estos artículos se empezó a conocer como la "arquitectura ANSI/SPARC", siendo su título completo "Estructura del SGBD ANSI/X3/SPARC". El grupo de estudio propuso que se estandarizaran los interfaces, y definió un marco arquitectónico que contenía 43 interfaces, 14 de los cuales tratarían con el subsistema de almacenamiento físico del computador y por tanto este no se consideraba parte esencial de la arquitectura de SGBD.

La arquitectura ANSI/SPARC esta basada en la organización de los datos. Reconoce tres vistas de los datos: la vista externa, que es la del usuario, que debe ser un usuario programador; la vista interna, la del sistema o la máquina; y la vista conceptual, la de la empresa. Para cada una de estas vistas, se requiere una definición del esquema apropiada. La Figura representa la arquitectura ANSI/SPARC desde la perspectiva de la organización de los datos.

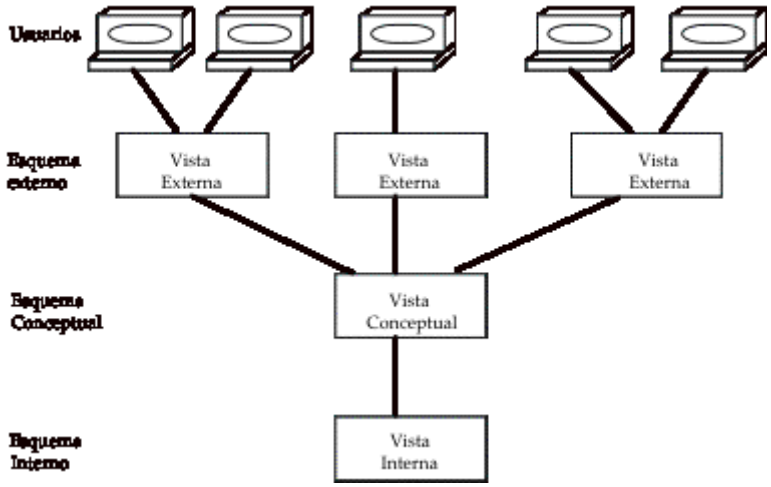


Figura 2. La Arquitectura ANSI/SPARC.

La vista interna está en el nivel más bajo de la arquitectura, que trata con la definición física y la organización de los datos. La localización de los datos en distintos periféricos de almacenamiento y los mecanismos de acceso usados para alcanzar y manipular los datos son los problemas que se tratan a este nivel.

En el otro extremo está la vista externa, que está relacionada con como los usuarios ven la BD. La vista de un usuario individual representa la porción de BD que será accedida por dicho usuario así como las relaciones entre los datos que le gustaría ver al usuario. Una vista puede compartirse entre un número de usuarios. La colección de vistas de usuario constituye el esquema externo.
Entre esos dos extremos está el esquema conceptual, que es una definición abstracta de la BD. Es la visión del "mundo real" que tiene la empresa modelada en la BD [Yormark 77]. Como tal, se supone que representa los datos y las relaciones entre los datos sin considerar los requisitos de las aplicaciones individuales o las restricciones del medio de almacenamiento físico.

La transformación entre estos tres niveles se acompaña de correspondencias que especifican como se puede obtener una definición en un nivel a partir de una definición en otro nivel.

Teniendo en cuenta su arquitectura se pueden distinguir tres generaciones de sistemas. La primera generación de sistemas de gestión de Bases de Datos se caracteriza por tener una arquitectura de un único nivel. Ya se había conseguido la descripción centralizada de los datos con la eliminación de la redundancia, pero no existía separación entre la estructura lógica y física de los datos. Estos sistemas tampoco tenían la posibilidad de definir vistas parciales. Algunos sistemas con estas características son el IDS de HONEYWELL y el TOTAL de CINCOM .
La segunda generación de sistemas aparece con las primeras especificaciones de CODASYL en 1971. En ellas se habla de esquema (descripción global, tanto a nivel lógico como físico) y de subesquemas (vistas parciales de los usuarios). Esta arquitectura a dos niveles ya tiende a lograr la independencia de los datos. Son sistemas típicos de esta generación el IMS de IBM, basado en el modelo jerárquico, y el IDMS de CULET, basado en las especificaciones de CODASYL.

La tercera generación tiene su origen en la propuesta de 1977 de ANSI/SPARC, y posteriormente de CODASYL 1978 de una arquitectura a tres niveles para los sistemas de gestión de Bases de Datos. Con esta propuesta, ya se consigue la independencia entre la estructura lógica y física de los datos, de forma que los cambios tienen una mínima repercusión en los programas.
En la propuesta de ANSI/SPARC se definen tres niveles de representación de la información y se hacen corresponder con tres tipos de esquemas: conceptual, interno y esquemas externos. En el estado actual de la tecnología de Bases de Datos no existen implementaciones de Sistemas Gestores de Base de Datos que asuman modelos de Datos lo suficientemente expresivos y generales. Esto hace imposible implementar directamente el esquema conceptual ya que este es una descripción de muy alto nivel de los datos y sus interrelaciones. Esto hace que muchos autores prefieran desdoblar el esquema conceptual de la arquitectura ANSI/SPARC en otros dos, con lo que surge lo que algunos de estos autores consideran una arquitectura con cuatro niveles:

- **Esquema conceptual**: visión desde un punto de vista organizativo independiente del SGBD que se utilice. Describe la información de la organización (entidades y relaciones) sin considerar ningún tema relacionado con el soporte informático.
- **Esquema lógico**: visión expresada en términos del Modelo de Datos concreto que asume el SGBD sobre el cual se va a desarrollar el sistema de información. Se representan las entidades y relaciones de acuerdo a las características del SGBD, sin considerar todavía detalles de su representación física
- **Esquema interno**: descripción de la representación en la memoria masiva del sistema informático de los datos del esquema lógico, sus interrelaciones y los mecanismos ofrecidos para acceder a ellos.
- **Esquemas externos**: describen los datos y relaciones relevantes para cada aplicación. Son subconjuntos del esquema lógico.

En la descripción del esquema conceptual se emplea un Modelo de Datos muy expresivo y de muy alto nivel (con conceptos muy cercanos a los del mundo real que intenta modelar). En la descripción del esquema lógico o canónico de la BD se emplea un Modelo de Datos para la implementación, es decir, el Modelo de Datos (de mucho más bajo nivel) asumido por el SGBD sobre el que se vaya a realizar la implementación.

Sin embargo, debe quedar claro que la inclusión de este nuevo nivel de descripción de la información no es, en ningún caso, una modificación o aportación a la arquitectura ANSI/SPARC. Este desdoblamiento de la vista conceptual de la BD en esquema conceptual y esquema lógico no es más que una solución a las carencias expresivas de las que adolecen los modelos de Datos que son asumidos por los Sistemas Gestores de Base de Datos disponibles. Por los tanto, se trata de algo más relacionado con el proceso de diseño de una BD que con su arquitectura.