Iniciación a Oracle8

José Manuel Navarro http://www.lawebdejm.com

Edición de febrero de 2004

Índice

Introducción a este manual

¡Bienvenido a este pequeño manual de Oracle!

Ya han pasado más de tres años desde que escribí por primera vez esta introducción, y nunca me imaginé que este manual llegaría a tantas manos como ha llegado. Desde que publiqué el manual en Internet, me han llegado comentarios desde todas las partes del mundo hispano-hablante, y de personas de todas tipo de empresas, gobiernos, estudiantes, etc. Gracias a todos los que lo habéis leído con atención, y por todas vuestras felicitaciones, comentarios y dudas que me habéis enviado. Por vosotros me he animado a seguir ampliando este manual.

Tratando sobre la versión 8 de Oracle, cualquiera podría decir que se ha quedado obsoleto, y la verdad es que no le falta razón. Sin embargo, la mayoría de los capítulos no tratan sobre temas específicos y avanzados de Oracle, sino sobre conceptos sobre bases de datos relacionales, la arquitectura interna o el uso de SQL, y la verdad es que sobre estos tres aspectos no hay cambios muy a menudo. De todas formas, algunos detalles de lo que explico han podido quedar obsoletos, así que os recomiendo que los contrastéis con los manuales de Oracle para aseguraros sobre cómo funcionan en las versiones actuales.

Yo no soy ningún experto en Oracle (aunque algunos me tratáis como si lo fuera), sino que simplemente soy una persona con cierta experiencia y con ganas de compartir lo que sabe. Es por eso que este manual no da los detalles más ocultos de Oracle, ni los trucos más avanzados para optimizar bases de datos, ni siquiera pretende ser un texto de referencia para consultar la sintaxis de las instrucciones, sino que te ayudará a introducirte en el mundo de Oracle, empezando desde cero, y llegando a un nivel que te permitirá seguir aprendiendo por ti mismo. También puede ser útil para los que habéis trabajado algo con SQL en otras bases de datos, y queréis comprender la arquitectura interna de Oracle y los conceptos de base.

Y por último, como ya sabéis, estoy disponible en la dirección de correo jm@lawebdejm.com para cualquier duda, sugerencia o corrección, y os animo a que visitéis mi página web, http://www.lawebdejm.com, en la que voy publicando distinto material sobre Oracle y otras tecnologías (Win32, C/C++, Delphi, UML, etc.)

Un saludo y espero que nos volvamos a ver en el próximo manual.

José Manuel Navarro

Introducción a las bases de datos

¿Qué es una base de datos?

Una base de datos es un programa residente en memoria, que se encarga de gestionar todo el tratamiento de entrada, salida, protección y elaboración de la información que almacena.
Aunque aparentemente podamos pensar que una base de datos son ficheros donde se almacena información, en realidad esto no es así. El corazón de una base de datos es el *motor*, que es el programa que debe estar ejecutándose en una máquina para gestionar los datos. Además de este programa y los archivos con datos, existen otras utilidades auxiliares, como programas para realizar copias de seguridad, intérpretes SQL, etc.

Tipos de bases de datos

Desde el punto de vista de la organización lógica:

a) Jerárquicas. (Progress)
b) Relacionales. (Oracle, Access, Sybase…)

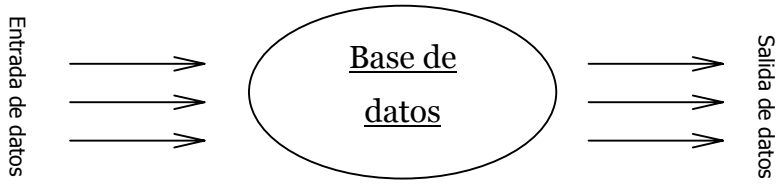
Desde el punto de vista de la arquitectura y el número de usuarios:

a) De escritorio (dBase, Access, Paradox…)
b) Cliente/servidor (Oracle, Sybase…)

Oracle es una base de datos relacional para entornos cliente/servidor, es decir, que aplica las normas del álgebra relacional (conjuntos, uniones, intersecciones…) y que utiliza la arquitectura cliente/servidor, donde en un lado de la red está el servidor con los datos y en el otro lado están los clientes que *interrogan* al servidor.
Todo lo que hablemos a partir de ahora será aplicable sólo a bases de datos Relacionales cliente/servidor, concretamente para bases de datos Oracle7 y Oracle8

Funciones de las bases de datos

a) Permitir la introducción de datos por parte de los usuarios (o programadores).
b) Salida de datos.
c) Almacenamiento de datos.
d) Protección de datos (seguridad e integridad).
e) Elaboración de datos.



Básicamente, la comunicación del usuario-programador con la base de datos se hace a través de un lenguaje denominado **SQL**: Structured Query Laguage (Lenguaje estructurado de consultas).

Conocimientos necesarios

Para un programador de bases de datos el conocimiento mínimo debe ser de:

- Conocimiento básico de las estructuras internas de Oracle.
- Lenguaje SQL
- Utilidades básicas: (SQL*Plus, Export, Import…)
- Lenguaje de programación PL/SQL
- Tareas simples de administración
- *Tunning* básico de SQL.

Tareas más propias de un administrador de bases de datos pueden ser:

- Los conocimientos propios de un programador de bases de datos.
- Conocimiento profundo de estructuras internas de Oracle.
- Conocimiento profundo de los catálogos de sistema.
- Utilidades de administración (SQL*DBA, Server Manager…)
- *Tunning* avanzado de SQL, red, memoria, discos, CPU…

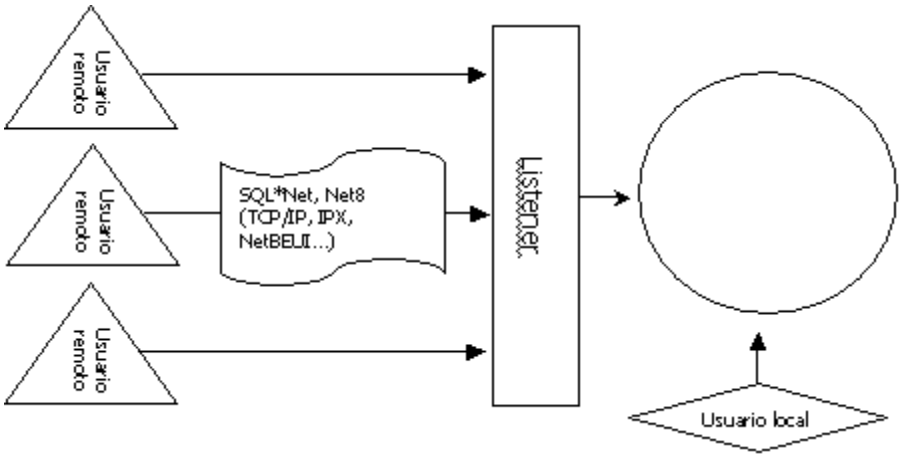
Componentes de las bases de datos

Una base de datos consta de varios componentes:

a) Motor: el programa ejecutable que debe estar en memoria para manejar la base de datos. Cuando este programa está ejecutándose se dice que la base de datos está levantada (*startup*), en caso contrario se dice que la base de datos está bajada (*shutdown*).

b) Servicio de red: Es un programa que se encarga de establecer las conexiones y transmitir datos entre cliente y servidor o entre servidores.

c) *Listener* (escuchador): Es un programa residente en memoria que se encarga de recibir las llamadas que llegan a la base de datos desde la red, y de pasárselas a esta. Una base de datos que no tenga un *listener* cargado, no podrá recibir llamadas remotas. El *listener* se comunica con el servicio de red.



d) Utilidades: Programas de utilidad como pueden ser:

- · Intérpretes de consultas.
- · Programas de administración de base de datos.
- · Programas de copia de seguridad.
- · Monitores de rendimiento.

A todo el conjunto de la base de datos se le denomina RDBMS: Relational DataBase Manager System, decir: Sistema de gestión de bases de datos relacionales.

El primer fabricante en diseñar un RDBMS fue IBM, aunque fue Oracle, en 1979, la primera empresa hacer una implementación comercial de un sistema de bases de datos relacionales.

El modelo relacional

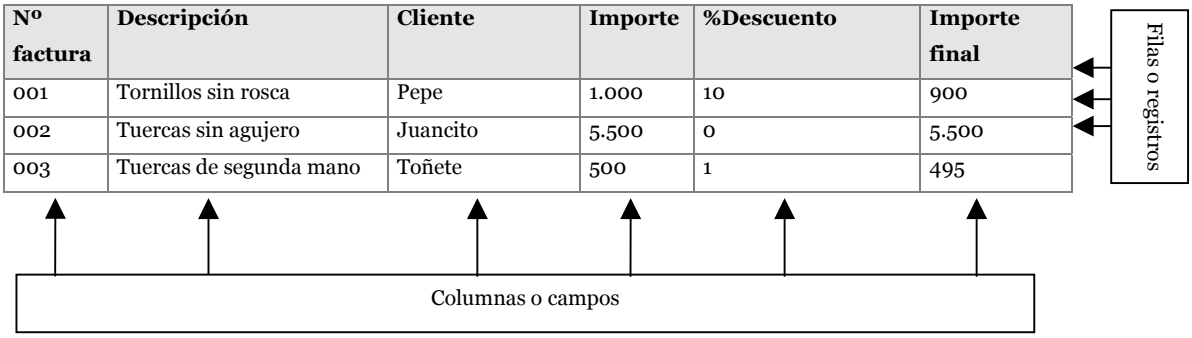
Concepto de tabla

Una tabla es una estructura lógica que sirve para almacenar los datos de un mismo tipo (desde el punto de vista conceptual). Almacenar los datos de un mismo tipo no significa que se almacenen sólo datos numéricos, o sólo datos alfanuméricos. Desde el punto de vista conceptual esto significa que cada entidad se almacena en estructuras separadas.
Por ejemplo: la entidad factura se almacena en estructuras diseñadas para ese tipo de entidad: la tabla FACTURA, la tabla FACTURA_COMPRA etc. Así, cada entidad, tendrá una estructura (tabla) pensada y diseñada para ese tipo de entidad.
Cada elemento almacenado dentro de la tabla recibe el nombre de *registro* o *fila*. Así, si la tabla FACTURA almacena 1.000 facturas, se dice que la tabla FACTURA contiene 1.000 registros o filas.

Una tabla se compone de *campos* o *columnas,* que son conjuntos de datos del mismo tipo (desde el punto de vista físico). Ahora cuando decimos "del mismo tipo" queremos decir que los datos de una columna son de todos del mismo tipo: numéricos, alfanuméricos, fechas…

Con lo que hemos dicho la estructura de una tabla es esta:

Nº factura	**Descripción**	**Cliente**	**Importe**	**%Descuento**	**Importe final**
001	Tornillos sin rosca	Pepe	1.000	10	900
002	Tuercas sin agujero	Juancito	5.500	0	5.500
003	Tuercas de segunda mano	Toñete	500	1	495



En este esquema se puede ver que cada fila almacena los datos de una factura (es decir, la entidad factura en sí), y cada columna almacena los datos de un mismo tipo (las descripciones, los clientes, etc).
De este modo se pude crear una tabla para cada tipo de entidad, y almacenar en ella los valores correspondientes.

Concepto de índice

Antes de ver qué es un índice tenemos que entender cómo se almacenan los datos.
Los registros de una tabla se almacenan uno detrás de otro, respetando las longitudes de cada columna. Esto es una norma general pero en la actualidad no cumple en todas las bases de datos, ya que este sistema es poco eficiente. De todas formas vamos a imaginar que es así, ya que va a servir para entender qué pasa *entre bastidores*.
Los tabla de FACTURA que hemos visto antes tiene la siguiente estructura:

Columna	Tipo	Ocupación (bytes)
Nº factura	N(3)	3+1
Descripción	A(50)	50+1
Cliente	A(20)	20+1
Importe	N(12)	12+1
Descuento	N(3)	3+1
Importe final	N(10)	10+2

La ocupación se incrementa en uno para incluir una marca de fin de columna, y en la última columna una marca de fin de registro.

La forma de almacenar en el disco duro los registros el ejemplo anterior sería la siguiente:

```
BOF||001|Tornillos sin
rosca···························|Pepe·············00000000
1000|010|0000000900||002|Tuercas·sin·agujero·····················
··|Juancito···········000000005500|000|0000005500||003|Tuercas·de·segun
da·mano·····················|Toñete···········000000000500|001|0
000000495|EOF
```

Podemos ver que al principio de la tabla hay una marca **BOF** (Begin Of File), y al final de la tabla una marca **EOF** (End Of File). Esto delimita el rango en el que están los datos de una determinada tabla.
Hay que darse cuenta que aunque la descripción de la factura no ocupe los 50 caracteres, es base de datos se están almacenando los restantes con un carácter de relleno, en nuestro caso el carácter "·"

Si a la base de datos le damos la siguiente orden:

"Dime la descripción de aquellas facturas cuyo cliente sea Toñete"

Lo que hará es recorrerse el fichero de datos desde la marca BOF hasta la marca EOF, "dando saltos" de N caracteres para leer sólo el campo cliente.
Así dará saltos de 55 (3+1+50+1) bytes que es el espacio que hay entre el principio de registro y el principio de la columna "Cliente".
Una vez encontrada esta posición, sabemos que la descripción está 51 bytes anteriores al cliente, así que hay que posicionarse en ese byte, leer el valor y retornar el resultado.
El *pseudocódigo* que representa este algoritmo puede ser:

```
Abrir fichero;

Bucle mientras no se acabe el fichero

Dar salto de 54 bytes en el fichero;
Valor_campo = Leer 20 bytes de fichero;

Si valor_campo = "Toñete" entonces
      Posicion_cliente = Posicion actual del fichero;
      Dar salto de -posicion_cliente bytes;  // al principio del fichero
      Dar salto de posicion_cliente - 51 bytes;
      Valor = Leer 50 bytes;
      Retornar valor;

Fin-si;

Fin-bucle;

Retornar NO_ENCONTRADO;
```

En este *pseudocódigo* tenemos que tener en cuenta una de las características de la programación con ficheros: la instrucción de lectura es muy rápida (su tiempo es casi despreciable), mientras que el desplazamiento del cursor del fichero es una operación my lenta. Esto se traduce en que la instrucción **leer** no consume casi tiempo, sin embargo la instrucción **dar salto** es la que más tiempo consume y cuando mayor sea el número de bytes de desplazamiento, peor.
Esto es debido a que la operación más lenta en los soportes de disco es el posicionamiento de las cabezas lectoras en el cilindro y sector adecuados, y una vez que el posicionamiento ya está hecho, la lectura es prácticamente instantánea.
Así que en este algoritmo es muy lento porque hace demasiados saltos.
A este tipo de lectura se le denomina lectura secuencial o FULL SCAN y es el caso más desfavorable en una consulta a base de datos. Además, cuando mayor sea el volumen de datos, se consiguen peores tiempos con un FULL SCAN.

Ahora vamos a lo que nos ocupa: **un índice es una tabla paralela a otra principal que tan sólo contienen la(s) columna(s) indicada(s) durante su creación**. Estas columnas se las denomina *columnas indexadas*.

Podemos usar la analogía del índice de un libro.
Cuando nosotros necesitamos buscar un tema en un libro, tenemos dos posibilidades:

1.- Recorrernos todas las páginas del libro buscando la primera hoja de cada tema y comprobando si es el que necesitamos. En esta búsqueda perderemos la mayoría del tiempo pasando hojas y hojas, (lo que sería el posicionamiento de las cabezas lectoras), buscando el principio de cada tema. Cada vez que encontrásemos el principio de un nuevo tema, comprobaríamos si el es el que buscamos (lo que equivaldría a la lectura del dato), y esta operación de lectura no nos ocupará nada de tiempo.

2.- Podemos ir al índice en el que sólo están escritos los títulos de los temas y tan solo con pasar tres hojas (el posicionamiento de la cabezas lectoras es mínimo) ya hemos recorrido todo el temario. Después vamos a la página que nos indique el índice y consultamos lo que necesitemos. Este último desplazamiento sería el equivalente al salto que damos utilizando un puntero en un lenguaje de programación con el C.

Los índices en las tablas de BD son equivalentes a los índices de los libros. Siempre que exista índice, debe consultarse porque si no las búsquedas se dispararán en tiempo.

Formas normales

El análisis de un sistema de base de datos consiste en la investigación para decidir qué tablas nos hacen falta para resolver un problema. Existen muchos métodos para realizar este análisis, aunque quizá el más conocido sea el Entidad/Relación. Para completar el análisis de bases de datos, es necesario pasar por varias etapas, entre ellas, las principales son:

· Análisis conceptual (o lógico, o relacional): es un análisis abstracto de aquellas entidades que formarán la base de datos, así como las relaciones que establecen unas con otras y las restricciones que se aplican a cada una de ellas. El resultado de esta fase de análisis se ve reflejado en algo llamado Modelo Conceptual o lógico, que es una descripción de las entidades, atributos, relaciones y restricciones que compondrán la base de datos.
El análisis conceptual es abstracto, por lo que no depende de la base de datos que vayamos a utilizar, ni del sistema en que se vaya a implementar dicha base de datos.

· Análisis físico: consta de un análisis específico teniendo en cuenta que base de datos se va a utilizar (Oracle, Sybase…) y en qué arquitectura se va a implementar la base de datos (entornos multiusuario, plataformas NT…)

Las formas normales no son más que tres reglas que se deben tener un cuenta dentro del Análisis conceptual, utilizando concretamente el método Entidad/Relación.
El proceso de aplicar las tres formas normales se llama normalización. Un diseño de base de datos que no cumpla la primera forma normal no será correcto. Cuantas más formas normales cumpla el diseño de base de datos, significará que la base de datos está más correctamente analizada.

Primera forma normal
Identificar cada tabla con una clave primaria, y poner los datos en tablas separadas, de manera que los datos de cada tabla sean de un tipo similar (desde el punto de vista conceptual)

Segunda forma normal
Sacar las columnas que sólo dependen de una parte de la clave primaria a otra tabla.

Tercera forma normal
Incluir en cada tabla sólo datos que pertenezcan a la misma unidad lógica.

Estas tres normas las vamos a explicar con un ejemplo:
Supongamos que necesitamos una tabla para almacenar facturas, y los datos que nos interesan son los siguientes:

Dato	Tipo
Descripción	A(50)
Nombre del cliente	A(20)
Dirección del cliente	A(30)
Teléfono del cliente	A(10)
Importe	N(12)

Si el programador que está diseñando la tabla, no tiene mucha experiencia, lo primero que hará es definir la tabla tal y como aparece anteriormente. Si además metemos datos, la tabla se verá así:

Descripción	Nombre cliente	Dirección cliente	Teléfono cliente	Importe
Tornillos sin rosca	Federico Antóñez	C/ Alta, nº 2	555-123546	500
Servicios prestados	Juancito Pérez Pí	C/ del Abedul, s/n	555-131415	4.587
Compra de tuercas sin agujero	Federico Antóñez	C/ Baja, nº 2	555-123456	258.987
Atrasos	Juancito Pérez Pí	C/ del Abedul, s/n	555-131415	1.245.847
Tornillos sin rosca	Juancito Pérez Pí	C/ del Abedul, s/n	555-131415	500

Según la primera forma normal, tenemos la necesidad de identificar cada uno de los registros de esta tabla inequívocamente. No podemos utilizar la descripción porque es posible que haya dos facturas con la misma descripción (dos ventas de tornillos), tampoco el cliente porque un cliente suele tener más de una factura. Ni tampoco el importe porque es normal tener varias facturas con el mismo importe. Para ello tenemos que definir una nueva columna que nos identifique cada una de las facturas

Es posible (y bastante común) que no encontremos una columna que sea capaz de identificar a al registro completo, por ello se puede definir más de una columna dentro de la clave. En este caso es el conjunto de valores de las columna seleccionadas el que no se podrá repetir.

Esta columna (o conjunto de ellas) se denomina clave primaria (o *primary key*).

Columna	**Tipo**
(*) Referencia	A(10)
Descripción	A(50)
Nombre cliente	A(20)
Dirección cliente	A(30)
Teléfono cliente	A(10)
Importe	N(12)

Las columnas marcadas con (*) son las que componen la clave primaria. Según este nuevo esquema, nuestra tabla con datos quedará así:

Ref.	**Descripción**	**Nombre cliente**	**Dirección cliente**	**Teléfono cliente**	**Importe**
FR00123	Tornillos sin rosca	Federico Antóñez	C/ Alta, nº 2	555-111111	500
FR00124	Servicios prestados	Juancito Pérez Pí	C/ del Abedul, s/n	555-131415	4.587
FR00125	Compra de tuercas sin agujero	Federico Antóñez	C/ Baja, nº 2	555-111112	258.987
FR00126	Atrasos	Juancito Pérez Pí	C/ del Abedul, s/n	555-131415	1.245.847
FR00127	Tornillos sin rosca	Juancito Pérez Pí	C/ del Abedul, s/n	555-131415	500

Ahora podemos estar seguros de que no habrá dos facturas con la misma referencia por lo que podemos consultar la factura con referencia 'FFR00123' y estaremos seguros de que sólo habrá una.

El siguiente paso de la primera forma normal es poner los datos en tablas separadas, asegurándonos de que los datos de una tabla son datos correspondientes a aquello que almacena la tabla.

En este ejemplo podemos ver cómo en la tabla FACTURA se están guardando datos del cliente (dirección y teléfono). En caso de que un cliente tenga más de una factura (como en el ejemplo), estaremos repitiendo la dirección y el teléfono para cada una de las facturas.
Esto se denomina redundancia de datos y produce tres efectos negativos:

1.- Mayor ocupación en disco de los datos: el mismo dato se repite N veces, por lo que ocupan espacio innecesario. Podéis ver cómo se repite la dirección y el teléfono de "Juancito Pérez Pí" tres veces, desperdiciando espacio (con almacenarlo una sola vez sería suficiente).

2.- Posibles inconsistencias de datos: debido a la repetición de datos, es posible que por un error, los datos sean inconsistentes. Por ejemplo, podéis ver que el teléfono de "Federico Antóñez" es 555-111111 en un registro pero 555-111112 en el otro, así que... ¿cuál de los dos teléfonos es el correcto?

3.- Problemas a la hora de cambiar datos repetidos: si un cliente cambia de teléfono o dirección, tenemos que modificar todas sus facturas (o cualquier tabla donde aparezca el número de teléfono) para cambiarle el dato, siendo un proceso engorroso y pudiendo crear más inconsistencias si cometemos un error.

Hay casos muy especiales en los que la redundancia de datos puede ser recomendable por razones de rendimiento, aunque esta es la excepción que confirma la regla, y yo no lo habría sin pensármelo dos veces.

La solución que da la primera forma normal a este problema es poner los datos en tablas separadas, dependiendo del origen de la información: la información perteneciente a factura irá en la tabla FACTURA y la información perteneciente a clientes irá en la tabla CLIENTE.
Además, podemos encontrarnos con el problema que los clientes de países distintos tiene una codificación independiente, es decir, que pude existir el cliente 1 de España y el cliente 1 de Argentina a la vez.
Un diseño que cumpla la primera forma normal podría ser:

FACTURA	
Columna	**Tipo**
(*) Referencia	A(10)
Descripción	A(50)
Cód cliente	N(5)
País	A(20)
Importe	N(12)

CLIENTE	
Columna	**Tipo**
(*) Cód. Cliente	N(5)
(*) País	A(20)
Nombre	A(50)
Teléfono	A(10)
Dirección	A(50)

Tan sólo se almacena el código del cliente para cada una de sus facturas, y cuando se tenga que modificar la dirección, se modificará para todas las facturas de ese cliente.
Con esto ya hemos hecho que se cumpla la 3ª forma normal.
Siguiendo con nuestro ejemplo, los datos quedarían así:

FACTURA:

Ref.	Descripción	Cód cliente	País	Importe
FR00123	Tornillos sin rosca	1	Argentina	500
FR00124	Servicios prestados	2	España	4.587
FR00125	Compra de tuercas sin agujero	1	Argentina	258.987
FR00126	Atrasos	2	España	1.245.847
FR00127	Tornillos sin rosca	2	España	500

CLIENTE:

Cód. cliente	País	Nombre	Teléfono	Dirección
1	Argentina	Federico Antóñez	555-111111	C/ Alta, nº 2
1	España	Juancito Pérez Pí	555-131415	C/ del Abedul, s/n
2	España	Antoñito “el salao”	555-999888	C/ Marismas, 25

Y para que la tabla CLIENTE cumpla con la primera forma normal, hemos tenido que añadir una nueva columna (Código), que sirve para identificar a cada cliente con un código.
Como es posible que exista el mismo código de cliente varias veces (una vez por cada país), la columna País se ha tenido que incluir dentro de la clave primaria.
La segunda forma normal nos dice que hay que sacar las columnas descriptivas que pertenezcan a la clave a otra tabla.
Si embargo, la primera forma normal no nos dice que la tabla CLIENTE esté mal definida, ya que todos los campos son datos relacionados con el cliente. Pero vemos que el País (España, Argentina, etc.) se repetirá varias veces, volviendo a caer en el error de la redundancia.
Para ello hay que crear una tabla aparte en la que se incluya el código y la descripción del país y así, a la hora de almacenar el país en la tabla CLIENTE, sólo se almacenará un código y no su descripción completa que ocupa mucho más espacio. Además a la hora de modificar una descripción, sólo habrá que modificarla una vez.

El esquema en segunda forma normal quedaría así:

FACTURA	
Columna	**Tipo**
(*) Referencia	A(10)
Descripción	A(50)
Cód. Cliente	N(3)
Cód país	N(5)
Importe	N(12)

CLIENTE	
Columna	**Tipo**
(*) Cód cliente	N(3)
(*) Cód país	N(5)
Nombre	A(50)
Teléfono	A(10)
Dirección	A(50)

PAIS	
Columna	**Tipo**
(*) Cód. país	N(5)
Descripción	A(50)

Y los datos del siguiente modo:

FACTURA:

Ref.	**Descripción**	**Cód cliente**	**Cód. País**	**Importe**
FR00123	Tornillos sin rosca	1	22	500
FR00124	Servicios prestados	2	34	4.587
FR00125	Compra de tuercas sin agujero	1	22	258.987
FR00126	Atrasos	2	34	1.245.847
FR00127	Tornillos sin rosca	2	34	500

CLIENTE:

Cód. cliente	**Cód. País**	**Nombre**	**Teléfono**	**Dirección**
1	22	Federico Antóñez	555-111111	C/ Alta, nº 2
1	34	Juancito Pérez Pí	555-131415	C/ del Abedul, s/n
2	22	Antoñito “el salao”	555-999888	C/ Marismas, 25

PAÍS:

Cód. País	**Descripción**
22	Argentina
34	España

En este punto, aunque sólo hayamos aplicado la primera y segunda forma normal, ya tenemos la base de datos normalizada, ya que la tercera forma normal, se cumple en todas las tablas.

Una forma de abreviar las formas normales es aplicando directamente la tercera, ya que si un esquema de base de datos cumple la tercera forma normal, automáticamente está cumpliendo la primera y la segunda.

Concepto de relación

Se denomina relación a todo aquellos vínculos que establecen unas tablas con otras, debidos a la aplicación de las formas normales.

En el ejemplo anterior, hemos creado relaciones entre unas tablas y otras desde el momento en que se separan los datos en más de una tabla y se utiliza el código como enlace entre unas y otras.
Una relación que hemos creado ha sido la que se establece entre la tabla CLIENTE y la tabla PAIS. Ambas tablas están "intercomunicadas" por una de sus columnas: Cód Pais para CLIENTE y Código para PAIS. Con esta relación sabemos que todo campo Cód País de la tabla CLIENTE, tiene un registro equivalente en la tabla PAIS.

Relación 1-1

La relación 1-1 se establece cuando un registro de la tabla A tiene un solo registro relacionado en la tabla B.
Esta relación se podría establecer por ejemplo si creamos una tabla de Pagos de facturas. Suponiendo que una factura se paga una sola vez, podemos definir la siguiente tabla para almacenar cuando se pagan las facturas:

PAGOS_FACTURA	
Columna	**Tipo**
(*) Referencia	A(10)
Fecha pago	F
Importe original	N(12)
% Recargo por retraso	N(3)
Importe final	N(10)

Podemos ver que la clave de esta tabla sólo es la referencia de la factura. Esto es el dato relevante que nos dice que la relación establecida entre una tabla y otra es 1-1. En la tabla PAGOS_FACTURA sólo puede aparecer una vez cada referencia. Y el la tabla FACTURA sólo puede aparecer una vez cada referencia. Siguiendo el mismo ejemplo anterior, podemos ver los datos de esta tabla, que nos dicen que sólo han sido pagadas dos facturas:

Ref.	**Fecha pago**	**Imp. Original**	**recargo**	**Importe final**
FR00123	19/05/2001	500	5%	525
FR00126	29/02/2003	4.587	0%	4.587

En este ejemplo he introducido dos redundancias de datos, que como ya he dicho, pueden resultar útiles en algunos casos muy especiales. En este caso, se almacena tanto el importe original, como el importe después de aplicar el recargo. El segundo sobraría, ya que es posible calcularlo con los otros dos datos. Sin embargo, y para no tener que hacer este cálculo continuamente, almacenamos los dos importes y así conseguiremos acelerar los cálculos.

Desde el punto de vista conceptual, las relaciones 1-1 son necesarias y convenientes, para que se cumpla la tercera forma normal, y que en cada tabla sólo aparezcan datos correspondientes a su nivel lógico.
Sin embargo, desde el punto de vista productivo y práctico, una relación 1-1 se puede sustituir por más registros en la tabla principal. Ya que un registro de A solo puede tener un registro en B, entonces las columnas de B pueden entrar a formas parte de A. De todas formas, siempre es mejor separa los datos en distintas tablas, ya que ahorraremos espacio y organizaremos mejor la información.
En un análisis más práctico que exhaustivo podríamos haber definido facturas de la siguiente manera:

FACTURA	
Columna	**Tipo**
(*) Referencia	A(10)
Descripción	A(50)
Cód país	N(3)
Cód. Cliente	N(5)
Importe	N(12)
Fecha pago	F
%Recargo por retraso	N(3)
Importe final	N(10)

Relación 1-N

Una relación 1-N es más común que 1-1, ya que, tanto desde el punto de vista conceptual, como desde el práctico, es necesario hacerlas.
Volviendo al caso de los pagos de las facturas, podemos permitir que una factura se pague fraccionada, por ejemplo a 30, 60 y 90 días. Para este caso necesitamos que una referencia aparezca más de una vez en la tabla de PAGOS, por lo que la clave primaria debe ser cambiada.

Si la definición de la tabla para una relación 1-N hubiese sido la siguiente:

PAGOS_FRACCIONADOS_FACTURA	
Columna	**Tipo**
(*) Referencia	A(10)
(*) Fecha pago	F
Importe original	N(12)
% Recargo por retraso	N(3)
Importe final	N(10)

Entonces una misma referencia de factura puede aparecer N veces, una por fecha distinta introducida. Así podemos pagar la segunda factura en tres fracciones, y la primera en una sola fracción.

PAGOS_FRACCIONADOS_FACTURA				
Referencia	**Fecha**	**Importe orig.**	**% Recargo**	**Importe final**
FR00123	19/05/2001	500	5%	525
FR00126	29/02/2003	2.000	0%	2.000
FR00126	29/03/2003	2.000	5%	2.100
FR00126	29/04/2003	587	10%	646

Si la clave se hubiese definido sólo con el campo "Referencia", no podríamos haber insertado más de una fecha para la misma referencia. Sin embargo al definirla con los campos "Referencia, Fecha", podemos introducir tantas parejas Referencia-Fecha como queramos.

Las relaciones 1-N también son llamadas normalmente maestro-detalle, donde el maestro es la tabla A (el 1 en la relación) y el detalle es la tabla B (el N en la relación).
En nuestro ejemplo FACTURA es el maestro y PAGOS_FRACCIONADOS_FACTURA un detalle de FACTURA.

Como norma general (lógicamente tiene sus excepciones) podemos decir que las columnas de la clave primaria de una tabla detalle tienen que ser las mismas que su maestro, añadiendo una (o varias) columnas a la clave (que marcan la diferencia entre el maestro y el detalle).
Esta norma se cumple para nuestro ejemplo.

Las relaciones 1-N pueden ser optativas u obligatorias en ambos sentidos.
Es decir, la tabla A puede estar obligada (o no) a tener registros relacionados en la tabla B,
La tabla B puede estar obligada (o no) a tener registros relacionados en la tabla A.

Una relación típica maestro-detalle, es optativa en sentido A-B pero obligatoria en sentido B-A. Significa que un maestro puede tener o no tener detalles, pero el detalle tiene que tener maestro obligatoriamente. Es decir: la factura puede que tenga pagos (si ya ha sido pagada) o puede que no

los tenga (si todavía no se ha pagado). Sin embargo, un pago tiene que tener obligatoriamente factura, porque si no hay factura ¡no hay nada que pagar!
El concepto de relación es muy común dentro de las bases de datos relacionales (de ahí viene su nombre) y está presente continuamente. Es muy importante manejarlo con soltura a la hora de definir las claves primarias para cada una de las tablas.

Claves foráneas

Una vez establecidas las relaciones entre las tablas, debemos estar seguros de que éstas se cumplen siempre. En nuestro ejemplo anterior debemos de asegurarnos de que si hay un registro en PAGOS_FRACCIONADOS_FACTURA, debe existir la correspondiente factura, y que si ésta es borrada, se haga lo mismo con sus pagos.
Las bases de datos nos ofrecen esta posibilidad a través de las claves foráneas, que no son más que un tipo de clave (como la primaria) que hace referencia a otras tablas.
Así la tabla PAGOS_FRACCIONADOS_FACTURA debe definir una clave que compruebe que siempre que se inserte un pago, exista la factura correspondiente (en la tabla FACTURA).
Además la base de datos se encarga de que si queremos borrar una factura (en la tabla FACTURA) no nos deje si existen pagos o bien borre todos los pagos correspondientes.
Las claves foráneas deben crearse sobre las tablas "hijo", o las tablas B en cada relación, normalmente en los detalles.

En nuestro ejemplo de las tablas FACTURA, CLIENTE y PAIS se deben aplicar las siguientes claves foráneas (restricciones).

CLIENTE: Comprobar que no se puede dar insertar un cliente en un país que no exista. Además no se podrá borrar un país siempre que existan clientes dados a ese país.

Clave foránea en CLIENTE(cod_pais) hace referencia sobre PAIS(codigo)

FACTURA: Comprobar que no se puede dar de alta una factura a un cliente (país, cliente) que no exista. También se comprobará que no se pueda borrar un cliente que tenga facturas.

Clave foránea en FACTURA(cod_pais,cod_cliente) hace referencia sobre CLIENTE(cod_pais,cod_cliente)

Se puede ver como la tabla PAIS no tiene ninguna clave foránea, ya que es "padre" o tabla A en todas las relaciones establecidas.

Como norma general, cada relación debe tener una clave foránea, y debe crearse sobre la tabla B de la relación que representa.

Normas básicas de codificación

A la hora de definir una tabla hay que tener en cuenta ciertos aspectos en la codificación:

- Sólo deben ser numéricas aquellas columnas que sean susceptibles de operaciones aritméticas. Es decir, un código de factura, no debe ser numérico ya que nunca se van a sumar los códigos.

- A la hora de codificar columnas alfanuméricas, hay que tener en cuenta el sistema de ordenación:

  Dada la siguiente lista de valores (de distinto tipo de dato):

Alfanumérico	Numérico
'50'	50
'41'	41
'21'	21
'1'	1
'5'	5
'20'	20
'100'	100
'13'	13
'10'	10
'2'	2

  La lista ordenada será la siguiente:

Alfanumérico	Numérico
'1'	1
'10'	2
'100'	5
'13'	10
'2'	13
'20'	20
'21'	21
'41'	41
'5'	50
'50'	100

  El orden, como vemos, difiere mucho uno de otro.
  Sin embargo, dada la siguiente lista de valores (de distinto tipo de dato):

Alfanumérico	Numérico
'050'	50
'041'	41
'021'	21
'001'	1
'005'	5
'020'	20
'100'	100
'013'	13
'010'	10
'002'	2

La lista ordenada será la siguiente:

Alfanumérico	Numérico
'001'	1
'002'	2
'005'	5
'010'	10
'013'	13
'020'	20
'021'	21
'041'	41
'050'	50
'100'	100

La diferencia está en que el método alfanumérico ordena por posiciones, no por valores absolutos.

- Las descripciones deben ser lo suficientemente largas como para almacenar el caso más desfavorable para la columna, aunque tampoco se deben crear columnas demasiado largas.
  Por ejemplo: para albergar nombre y apellidos nos valdrá un 2 nombres de 10 caracteres cada uno, más dos apellidos de 15 caracteres cada uno. Total 35 caracteres. Para darnos un margen de error podemos poner 40 caracteres. Más de esta estimación será una longitud exagerada para el campo.

Codificación compuesta o "claves inteligentes"

En bases de datos antiguas había una práctica muy común que consistía en utilizar una sola columna con varios significados. El significado de la columna dependía de las posiciones de los dígitos.
Por ejemplo, se podría definir la siguiente regla para almacenar las referencias de las facturas:

Dígitos 1-2	Día de emisión.
Dígitos 3-4	Mes de emisión.
Dígitos 5-8	Año de emisión.
Dígitos 9-14	Código de cliente.
Dígitos 14-20	Número de factura.

Así la referencia de la factura número 1, emitida a 23/8/1999, para el cliente código 567 sería:

23081999000567000001

Esto no tiene ningún sentido, ya que queda mucho más claro separar cada valor a su columna correspondiente, y si es necesario, definir todas las columnas necesarias como clave.

Fecha	Cliente	Número
23/8/1999	567	1

El origen de esta práctica viene de la época de las bases de datos jerárquicas en las que la clave sólo podía estar formada por un campo. Entonces era la única manera de definir una clave compuesta.

Estándar de nomenclatura de objetos

Cuando un equipo de desarrollo da los primeros pasos en un proyecto informático (de bases de datos o de cualquier otro tipo), lo primero que se debe definir es qué estándar de nomenclatura de objetos se va a utilizar.
El objetivo principal de esta tarea es que el esquema sea consistente y homogéneo, además de permitir una memorización más rápida de los objetos.

El estándar debe responder a las siguientes preguntas:

- ¿Los nombres de objetos van en mayúsculas o minúsculas?
- ¿Debo utilizar nombres lo más descriptivos posibles o sin embargo nombres muy cortos?
- ¿Puedo usar abreviaturas?
- ¿Los nombres deben ir en singular o en plural?

El estándar debe ser un documento que tengan presente en todo momento el equipo de desarrollo, y siempre debe aplicarse salvo contadas excepciones.

A continuación tienes ciertas normas, que aunque no pretenden ser un estándar, si que pueden resultarte de utilidad. Tú eres el que tienes que decidir si quieres aplicarlas, o bien crear tus propio estándar de nomenclatura:

1. Los nombres de objetos (tablas, índices, claves primarias, claves foráneas…) deben ir en mayúscula. Oracle interpreta por defecto todos los objetos en mayúscula a no ser que se escriba su nombre entre comillas dobles:

Nombres	**Interpretación de Oracle**
Factura, factura y FACTURA	Equivalente.
"FACTURA", "factura", "Factura"	Distintos objetos.

2. Los nombres de objetos deben ir en singular ya que el nombre representa a la entidad que almacena, y no las entidades que almacena. Una razón práctica es que con los nombres en minúscula se ahorra 1 ó 2 letras, lo cual no es despreciable.

Entidad	**Nombre recomendado**
Facturas	FACTURA
Facturas de proveedores	FACTURA_PROVEEDOR
Facturas que no han sido pagadas	FACTURA_PENDIENTE_PAGO
Facturas caducadas	FACTURA_CADUCADA

3. Los nombres a utilizar deben ser descriptivos, aunque no deben ser demasiado largos. Oracle admite hasta un máximo de 30 caracteres para los identificadores, aunque no es recomendable llegar hasta el límite.
   Nunca se deben utilizar nombres de objetos crípticos, como XP34TY ó 3456RRDW7E2. con esto lo único que conseguimos es complicar el esquema con nombres de tablas que son difíciles de recordar.

Entidad	**Nombre recomendado**
Empresas pertenecientes al sector de la construcción	EMPRESA_CONSTRUCCION
Clientes que han tenido algún impago	CLIENTE_MOROSO
Proveedores que se suelen retrasar en sus entregas	PROVEEDOR_LENTO, PROVEEDOR_RETRASO
Facturas caducadas de empresas del sector de la construcción	FACTURA_CONSTRUCCION_CADUCADA

4. Es recomendable utilizar abreviaturas, sobre todo si el nombre más descriptivo es demasiado largo. Como veremos más adelante, Oracle sólo permite 30 caracteres para el nombre de las tablas, por lo que muchas veces las abreviaturas se convierten en una obligación. Para nombres cortos no es necesario utilizar abreviaturas.

Entidad	**Nombre recomendado**
Empresas pertenecientes al sector de la construcción que han tenido alguna demolición	EMPRESA_CONSTR_DEMOLICION
Clientes que han tenido algún impago	CLIENTE_MOROSO
Proveedores que se suelen retrasar en sus entregas de empresas del sector de la construcción	PROVEEDOR_CONSTR_LENTO, PROVEEDOR_ CONSTR _RETRASO
Facturas caducadas de empresas del sector de la construcción	FACTURA_CONSTR_CADUCADA
Almacén de productos terminados para empresas del sector de la construcción	ALMACEN_ CONSTR _PROD_TERM

5. A la hora de nombrar tablas relacionadas entre si, es recomendable que el nombre empiece por el sufijo que representa la entidad principal.

Entidad	**Nombre recomendado**
Facturas	FACTURA
Líneas de Factura (detalle de FACTURA)	FACTURA_LINEA
Desglose de las líneas de factura (detalle de FACTURA_LINEA)	FACTURA_LINEA_DESGLOSE
Factura impagadas (Relación 1-1 con FACTURA)	FACTURA_IMPAGADA, FACTURA_IMPAGO

6. Se pueden establecer ciertas abreviaturas para los nombres de columnas:

Columnas típicas	**Abreviatura**
Código de…	C_xxx
Descripción de…	D_xxx
Referencia de …	REF_xxx
Importe de …	IMP_xxx
Precio de …	PRC_xxx
Porcentaje de …	PCT_xxx
Unidades de …	UDS_xxx
Tipo de …	TIP_xxx
Número de …	NUM_xxx

7. Los nombres de clave primaria deben ir precedidos del prefijo PK_ (Primary key), los de índices por IND_, y los de clave foránea por FK_ (Foreing key).
   El nombre restante será el de la propia tabla para las claves primarias, el de la tabla referenciada para las claves foráneas y para los índices una o dos palabras descriptivas que indiquen la razón por la que se crea ese índice (si es posible).

Conceptos de almacenamiento en Oracle

Concepto de Tablespace (espacio de tablas)

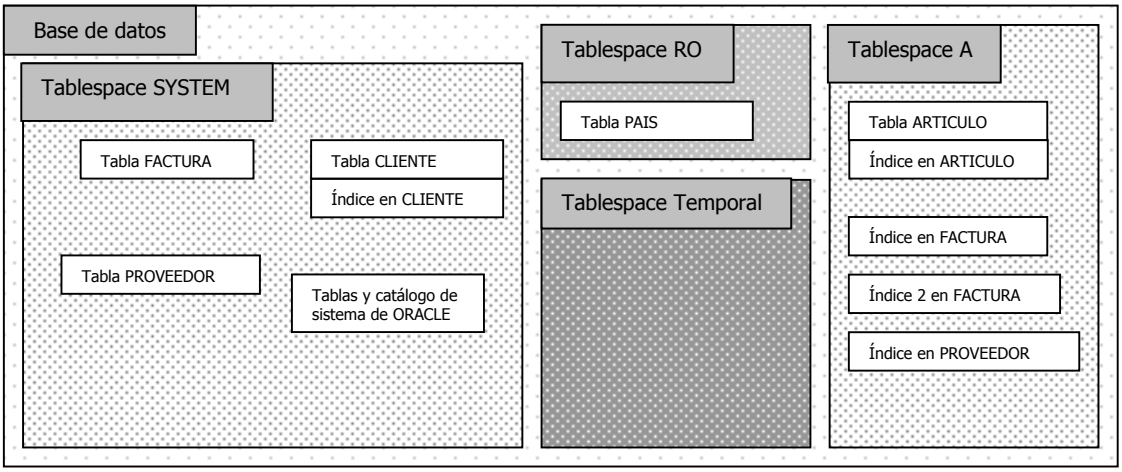
Una base de datos se divide en unidades lógicas denominadas TABLESPACES.
Un *tablespace* no es un fichero físico en el disco, simplemente es el nombre que tiene un conjunto de propiedades de almacenamiento que se aplican a los objetos (tablas, secuencias…) que se van a crear en la base de datos bajo el *tablespace* indicado (tablas, secuencias…).

Un objeto en base de datos debe estar almacenado obligatoriamente dentro de un tablespace.

Las propiedades que se asocian a un *tablespace* son:

- Localización de los ficheros de datos.
- Especificación de máximas cuotas de consumo de disco.
- Control de la disponibilidad de los datos (en línea o fuera de línea).
- Backup de datos.

Cuando un objeto se crea dentro de un cierto *tablespace*, este objeto adquiere todas las propiedades antes descritas del *tablespace* utilizado.



En este esquema podemos ver que, por ejemplo, la tabla ARTICULO se almacena dentro del *tablespace* A, y que por lo tanto tendrá todas las propiedades del *tablespace* A que pueden ser:

- Sus ficheros de datos están en $ORACLE_HOME/datos/datos_tablespace_A
- Los objetos no pueden ocupar más de 10Mb de espacio de base de datos.

- En cualquier momento se puede poner fuera de línea todos los objeto de un cierto *tablespace*.
- Se pueden hacer copiar de seguridad sólo de ciertos *tablespaces*.

Si nos fijamos, se puede apreciar que es posible tener una tabla en un *tablespace*, y los índices de esa tabla en otro. Esto es debido a que los índices no son más que objetos independientes dentro de la base de datos, como lo son las tablas. Y al ser objetos independientes, pueden ir en *tablespaces* independientes.

El *tablespace* SYSTEM es uno de los que se crear por defecto en todas las bases de datos Oracle. En él se almacenan todos los datos de sistema, el catálogo y todo el código fuente y compilado de procedimientos PL/SQL. También es posible utilizar el mismo *tablespace* para guardar datos de usuario.

En el esquema también vemos que hay un *tablespace* Temporal (en gris oscuro). Este representa las propiedades que tendrán los objetos que la base de datos cree temporalmente para sus cálculos internos (normalmente para ordenaciones y agrupaciones). Su creación difiere en una de sus cláusulas de creación.

El *tablespace* RO (en gris claro) difiere de los demás en que es de solo lectura (Read Only), y que por lo tanto todos los objetos en él contenidos pueden recibir órdenes de consulta de datos, pero no de modificación de datos. Estos puede residir en soportes de sólo lectura, como pueden ser CDROMs, DVDs, etc. Cuando se crea un *tablespace*, éste se crea de lectura/escritura. Después se puede modificar para que sea de solo lectura.

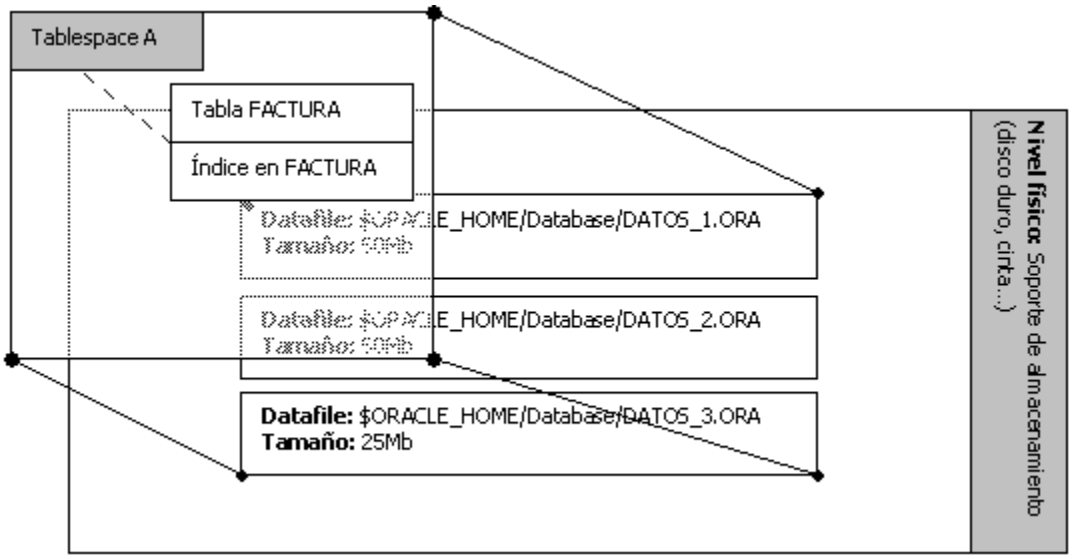
Un *tablespace* puede estar en línea o fuera de ella (Online o OffLine), esto es que todos los objetos contenidos en él están a disposición de los usuarios o están inhabilitados para restringir su uso.

Cualquier objeto almacenado dentro de un *tablespace* no podrá ser accedido si este está fuera de línea.

Concepto de Datafile (fichero de datos)

Un *datafile* es la representación física de un *tablespace*. Son los "ficheros de datos" donde se almacena la información físicamente.
Un *datafile* puede tener cualquier nombre y extensión (siempre dentro de las limitaciones del sistema operativo), y puede estar localizado en cualquier directorio del disco duro, aunque su localización típica suele ser $ORACLE_HOME/Database.
Un *datafile* tiene un tamaño predefinido en su creación (por ejemplo 100Mb) y este puede ser alterado en cualquier momento.
Cuando creemos un *datafile*, este ocupará tanto espacio en disco como hayamos indicado en su creación, aunque internamente esté vacío. Oracle hace esto para reservar espacio continuo en disco y evitar así la fragmentación. Conforme se vayan creando objetos en ese *tablespace*, se irá ocupando el espacio que creó inicialmente.

Un *datafile* está asociado a un solo *tablespace* y, a su vez, un *tablespace* está asociado a uno o varios *datafiles*. Es decir, la relación lógica entre *tablespaces* y *datafiles* es de 1-N, maestro-detalle.



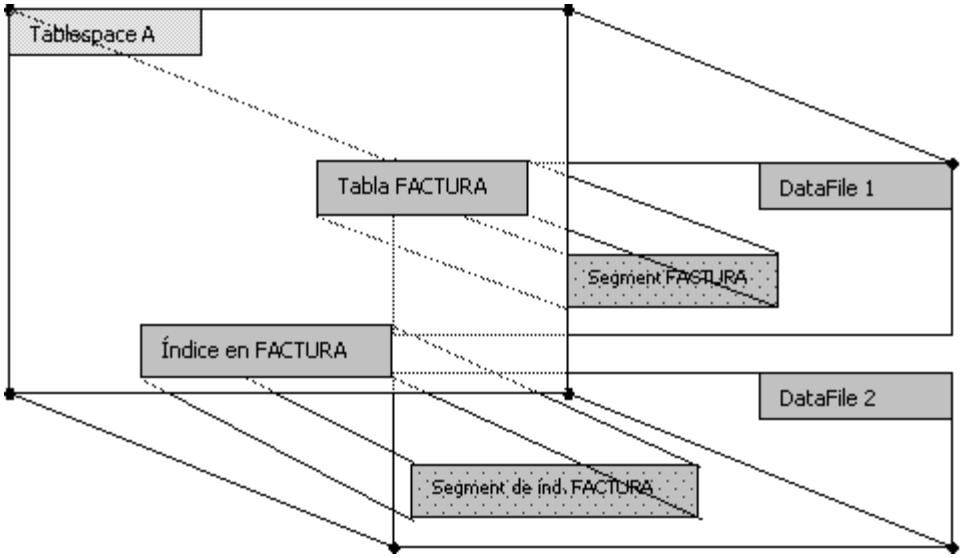
En el esquema podemos ver como el "Tablespace A" está compuesto (físicamente) por tres *datafiles* (DATOS_1.ORA, DATOS_2.ORA y DATOS_3.ORA). Estos tres *datafiles* son los ficheros físicos que soportan los objetos contenidos dentro del *tablespace* A.
Aunque siempre se dice que los objetos están dentro del *tablespace*, en realidad las tablas están dentro del *datafile*, pero tienen la propiedades asociadas al *tablespace*.

Cada uno de los *datafiles* utilizados está ocupando su tamaño en disco (50 Mb los dos primeros y 25 Mb el último) aunque en realidad sólo contengan dos objetos y estos objetos no llenen el espacio que está asignado para los *datafiles*.

Los *datafiles* tienen una propiedad llamada AUTOEXTEND, que se si está activa, se encarga de que el *datafile* crezca automáticamente (según un tamaño indicado) cada vez que se necesite espacio y no exista.
Al igual que los *tablespaces*, los *datafiles* también puede estar en línea o fuera de ella.

Concepto de Segment (segmento, trozo, sección)

Un *segment* es aquel espacio reservado por la base de datos, dentro de un *datafile*, para ser utilizado por un solo objeto.
Así una tabla (o cualquier otro objeto) está dentro de su segmento, y nunca podrá salir de él, ya que si la tabla crece, el segmento también crece con ella.
Físicamente, todo objeto en base de datos no es más que un segmento (segmento, trozo, sección) dentro de un *datafile*.
Se puede decir que, un segmento es a un objeto de base de datos, lo que un *datafile* a un *tablespace*: el segmento es la representación física del objeto en base de datos (el objeto no es más que una definición lógica).



Podemos ver cómo el espacio que realmente se ocupa dentro del *datafile* es el *segment* y que cada segmento pertenece a un objeto.

Existen cuatro tipos de segmentos (principalmente):

Segmentos de TABLE: aquellos que contienen tablas

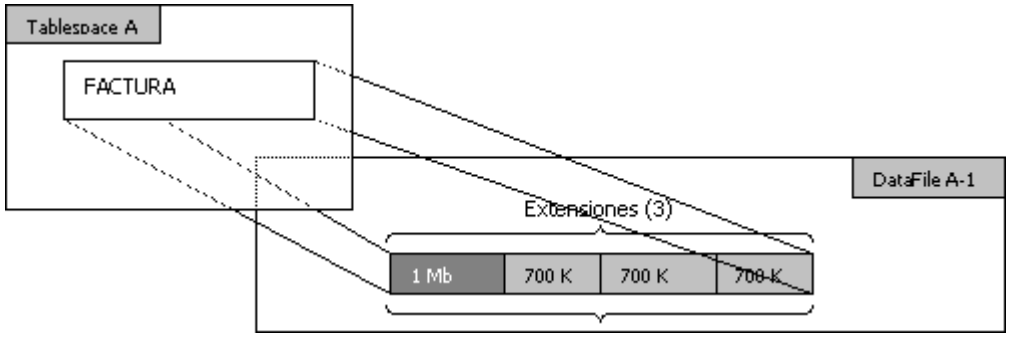
Segmentos de INDEX: aquellos que contienen índices

Segmentos de ROLLBACK: aquellos se usan para almacenar información de la transacción activa.

Segmentos TEMPORALES: aquellos que se usan para realizar operaciones temporales que no pueden realizarse en memoria, tales como ordenaciones o agrupaciones de conjuntos grandes de datos.

Concepto de Extent (extensión)

Para cualquier objeto de base de datos que tenga cierta ocupación en disco, es decir, cualquier objeto que tenga un *segment* relacionado, existe el concepto de *extent*.
Extent es un espacio de disco que se reserva de una sola vez, un segmento que se reserva en un momento determinado de tiempo. El concepto de *extent* es un concepto físico, unos están separados de otros dentro del disco.
Ya dijimos que todo objeto tiene su segmento asociado, pero lo que no dijimos es que este segmento, a su vez, se compone de distintas extensiones. Un segmento, puede ser reservado de una sola vez (10 Mb de golpe), o de varias veces (5 Mb hoy y 5 Mb mañana). Cada una de las veces que se reserva espacio se denomina "extensión".



En el esquema vemos como el objeto (tabla) FACTURA tiene un segmento en el *datafile* A-1, y este segmento está compuesto de 3 extensiones.
Una de estas extensiones tiene un color distinto. Esto es porque existen dos tipos de extensiones:

- INITIAL (extensiones iniciales): estas son las extensiones que se reservan durante la creación del objeto. Una vez que un objeto está creado, no se puede modificar su extensión inicial.

- NEXT (siguientes o subsiguientes extensiones): toda extensión reservada después de la creación del objeto. Si el INITIAL EXTENT de una tabla está llena y se está intentando insertar más filas, se intentará crear un NEXT EXTENT (siempre y cuando el *datafile* tenga espacio libre y tengamos cuota de ocupación suficiente).

Sabiendo que las extensiones se crean en momentos distintos de tiempo, es lógico pensar que unas extensiones pueden estar fragmentadas de otras. Un objeto de base de datos no reside todo junto dentro del bloque, sino que residirá en tantos bloque como extensiones tenga. Por eso es crítico definir un buen tamaño de extensión inicial, ya que, si es lo suficientemente grande, el objeto nunca estará fragmentado.

Si el objeto tiene muchas extensiones y éstas están muy separadas en disco, las consultas pueden retardarse considerablemente, ya que las cabezas lectoras tienes que dar saltos constantemente.

El tamaño de las extensiones (tanto las INITIAL como las NEXT), se definen durante la creación del objeto y no puede ser modificado después de la creación.
Oracle recomienda que el tamaño del INITIAL EXTENT sea igual al tamaño del NEXT EXTENT.

La mejor solución es calcular el tamaño que tendrá el objeto (tabla o índice), multiplicando el tamaño de cada fila por una estimación del número de filas. Cuando hemos hecho este cálculo, debemos utilizar este tamaño como extensión INITIAL y NEXT, y tendremos prácticamente la certeza de que no se va a producir fragmentación en ese objeto.
En caso de detectar más de 10 extensiones en un objeto (consultando el catálogo de Oracle, como veremos), debemos recrear el objeto desde cero (aplicando el cálculo anterior) e importar de nuevo los datos.

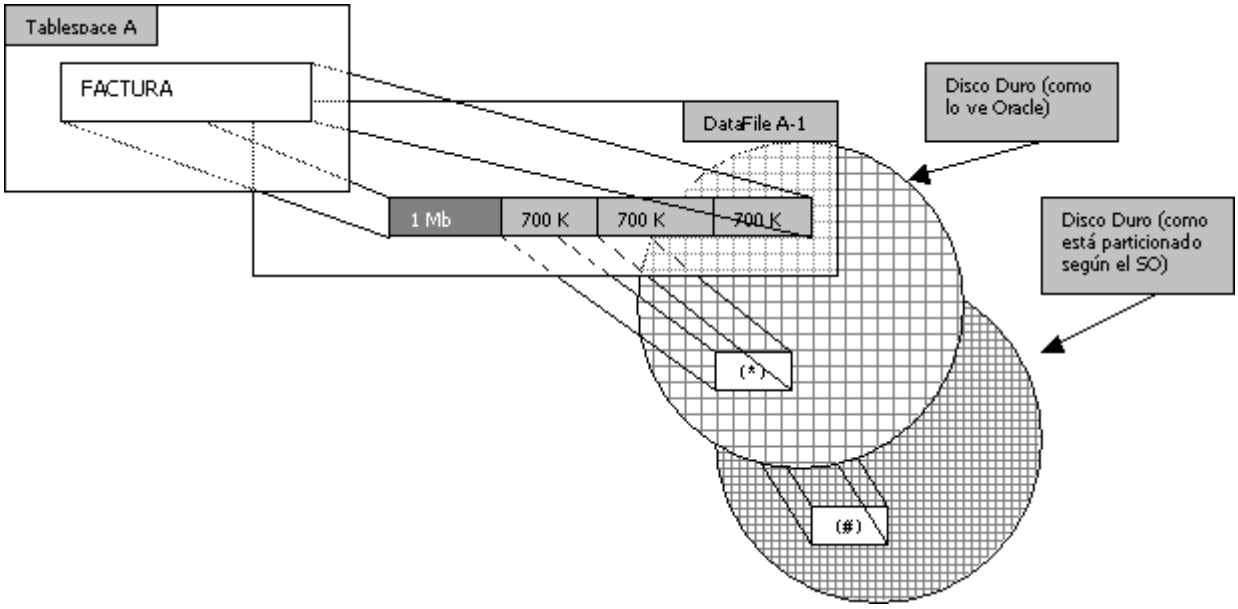
Ciertas operaciones, necesitan de espacio en disco para poder realizarse. El espacio reservado se denomina "segmentos temporales". Se pueden crear segmentos temporales cuando:

- Se crea un índice
- Se utiliza ORDER BY, DISTINTC o GROUP BY en un SELECT.
- Se utilizan los operadores UNION, INTERSECT o MINUS.
- Se utilizan joins entre tablas.
- Se utilizan subconsultas.

Concepto de Data block (bloque de datos)

Un *data block* es el último eslabón dentro de la cadena de almacenamiento.
El concepto de *Data block* es un concepto físico, ya que representa la mínima unidad de almacenamiento que es capaz de manejar Oracle.
Igual que la mínima unidad de almacenamiento de un disco duro es la unidad de asignación, la mínima unidad de almacenamiento de Oracle es el *data block*.
En un disco duro no es posible que un fichero pequeño ocupe menos de lo que indique la unidad de asignación, así si la unidad de asignación es de 4 Kb, un fichero que ocupe 1 Kb, en realidad ocupa 4 Kb.

Siguiendo con la cadena, cada segmento (o cada extensión) se almacena en uno o varios bloques de datos, dependiendo del tamaño definido para el extensión, y del tamaño definido para el *data block*.



(*) Espacio ocupado en el *data block* por la primera NEXT EXTENSION.
(#) Espacio ocupado en unidades de asignación del sistema operativo por los *data blocks* anteriores.

El esquema muestra toda la cadena de almacenamiento de Oracle.

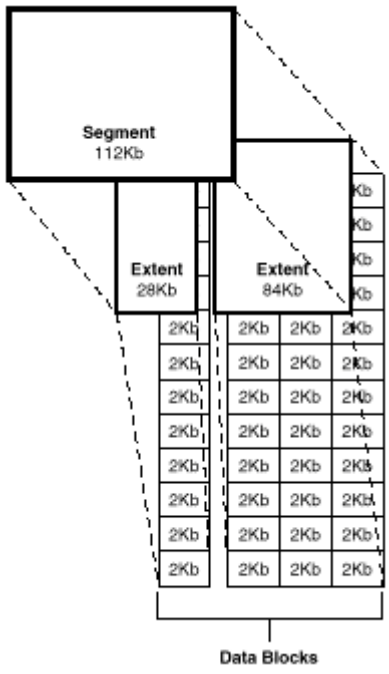
Desde el nivel más físico al más lógico:

· Unidades de asignación del sistema operativo (El más físico. No depende de Oracle)
· Data blocks de Oracle
· *Extents*
· *Segments*
· *DataFiles*
· *Tablespaces* (El más lógico)

El tamaño de las unidades de asignación del sistema operativo se define durante el particionado del disco duro (FDISK, FIPS…), y el espacio de los *data blocks* de Oracle se define durante la instalación y no puede ser cambiado.

Como es lógico, el tamaño de un *data block* tiene que ser múltiplo del tamaño de una unidad de asignación, es decir, si cada unidad de asignación ocupa 4 K, los *data blocks* pueden ser de 4K, 8K, 12K… para que en el sistema operativo ocupen 1, 2, 3… unidades de asignación.

Esquema extraído del Oracle8 Concepts



Estructuras de memoria

Todas las estructura que hemos visto se refieren a cómo se almacenan los datos en el disco. Sin embargo, y como es lógico, Oracle también utiliza la memoria del servidor para su funcionamiento. Oracle utiliza dos tipos de memoria

- Memoria local y privada para cada uno de los procesos: PGA (Process Global Area o Program Global Area).
- Memoria común y compartida por todos los procesos SGA (System Global Area o Shared Global Area).

Cada vez que se conecta un cliente al servidor, se ejecuta un subproceso que atenderá sus peticiones (a través del *fork* en Unix o con CreateThread en el mundo Windows), y este subproceso creará un nuevo bloque de memoria de tipo PGA.
El tamaño de este bloque de memoria dependerá del sistema operativo, y permanece invariable, aunque se puede configurar cambiando el valor de la variable SORT_AREA_SIZE del archivo de inicialización INIT.ORA.

Por cada instancia de base de datos, tendremos una zona de memoria global, el SGA, donde se almacenan aquellos datos y estructuras que deben se compartidos entre distintos procesos de la base de datos, como los procesos propios de Oracle y cada uno de los subprocesos que gestionan la conexión. El tamaño del SGA es uno de los puntos más críticos a la hora de mejorar el rendimiento de una base de datos, ya que, cuanto mayor memoria se reserve (mientras no sea memoria virtual), más rápidas se realizarán ciertas operaciones. Por ejemplo, las ordenaciones (una de las operaciones que más rápido deben hacerse) se realizan en el SGA si hay espacio suficiente. En caso contrario, se realizarán directamente en el disco, utilizando segmentos temporales.

El SGA se divide en cuatro grandes zonas:

- ***Database buffer cache****:* almacena los bloques que se han leído de los *datafiles*. Cada vez que es necesario acceder a un bloque, se busca el bloque en esta zona, y en caso de no existir, se lee de nuevo del *datafile* correspondiente. Cuantos más bloques quepan en esta zona de memoria, mejor será el rendimiento.

- ***SQL Area***: es la zona de memoria se almacenan compiladas las últimas sentencias SQL (y bloques PL/SQL) ejecutadas. Además se almacenan las variables acopladas (bind), el árbol de *parsing*, los buffer de ejecución y el plan de ejecución. Es importante que siempre que se utilice la misma sentencia, sea exactamente igual, para poder aprovechar sentencias previas almacenadas en el SQL Area.
  Es decir, las siguientes sentencias:

```
"SELECT * FROM TABLA"
```

```
"select * from tabla"
"SELECT  *  FROM   TABLA"
"SELECT *
FROM tabla"
```

se consideran distintas y no se aprovecha el SQL Area. Debe coincidir el texto exactamente, considerando mayúsculas y minúsculas, espacios, retornos de carro, nombre de parámetros, etc. Esto es debido a que se buscan dentro del SQL Area utilizando un *hash* de la sentencia, y un simple espacio (o cambiar una letra a mayúsculas) hace que el *hash* resultante sea distinto, por lo que no encontrará la sentencia dentro del SQL Area.
Cuanto mayor sea el espacio del *SQL Area*, se realizarán menos compilaciones, planes de ejecución y análisis léxicos, por lo que la ejecución de las consultas será más rápida.

- ***Redo cache***: almacena los registros de *redo* de las últimas operaciones realizadas. Estos registros se almacenan en los archivos de *redo*, que sirven para recomponer la base de datos en caso de error.

- ***Dictionary cache***: almacena datos del diccionario de Oracle, para utilizarlos en los planes de ejecución, optimización de consultas, etc. Cuantos más datos quepan en esta zona, mayor probabilidad habrá de que el dato que necesitamos ya esté en memoria, y no sea necesario acceder a las tablas del diccionario para leerlo.

Archivos de inicialización

Además de estructuras de disco y de memoria, un servidor Oracle necesita ciertos archivos para poder ejecutarse. Estos archivos se establecen durante la creación de la base de datos, y se consultarán cada vez que se arranque la base de datos, por lo que deben estar disponibles.
Básicamente podemos diferencias los tipos de archivos:

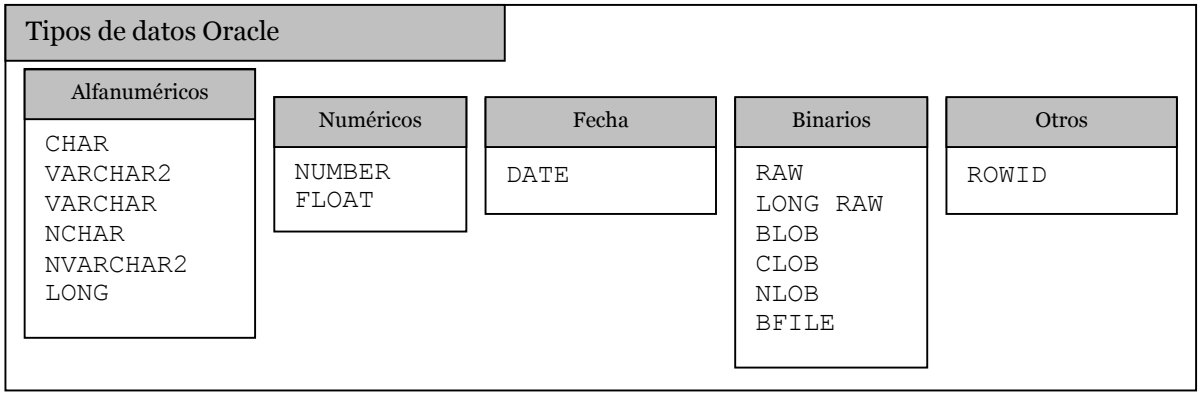
1. **Control files**: son archivos de control que se consultan cada vez que se arranca la base de datos. Indica datos como la localización de los datafiles, nombre de la base de datos.

2. **Init file:** es el archivo que contiene los parámetro de inicio de la base de datos (tamaño del bloque, tamaño del SGA, etc.). Normalmente tiene el nombre INIT.ORA

3. **Redo logs**: estos archivos contienen un historial de todas las instrucciones que han sido lanzadas a la base de datos, para poder recuperarla en caso de fallo. No se utilizan durante la inicialización, sino durante toda la ejecución de la base de datos.

Tipos de datos en Oracle

Los tipos de datos soportados por Oracle se agrupan en los siguientes conjuntos.



Los valores alfanuméricos van encerrados entre comilla simple: 'Alfanumérico'

Los valores numéricos son número simples: 123

Las fechas van encerradas entre comillas simples: '1/12/2000'

Los valores binarios no pueden ser representados (son fotos, videos...)

Tipo de dato CHAR(b)

Almacena cadenas de caracteres de longitud fija, desde 1 a 2.000 bytes de ocupación. El número de caracteres que se pueden almacenar se rige según la siguiente fórmula.

nº caracteres = bytes / character set

Para ASCII, el conjunto de caracteres ocupa un byte, por lo que coincide el número de caracteres máximos con la ocupación del tipo de dato.

Si se introduce un valor de 10 caracteres en un campo de CHAR(100), se rellenará con espacios las 90 posiciones restantes.

Así la siguiente expresión es cierta:

'Hola pepe' = 'Hola pepe '

Si se intenta introducir un valor demasiado grande para el campo, se intentará eliminar los espacios finales, y si cabe sin espacios, se introduce. Si aún así no cabe, se retorna un error.

Tipo de dato VARCHAR2(b)

Almacena cadenas de caracteres de longitud variable.

Si se define una columna de longitud 100 bytes, y se introduce en ella un valor de 10 bytes, la columna ocupará 10 y no 100 como hacía con el tipo de dato CHAR.

Tipo de dato VARCHAR(b)

En ***Oracle8*** es equivalente a VARCHAR2, en futuras versiones permitirá distintos criterios de comparación.

Tipo de dato NCHAR(b)

Almacena un valor alfanumérico de longitud fija con posibilidad de cambio de juego de caracteres. Puede almacenar tanto caracteres ASCII, EBCDIC, UNICODE…

Tipo de dato NVARCHAR2(b)

Almacena un valor alfanumérico de longitud variable con posibilidad de cambio de juego de caracteres. Puede almacenar tanto caracteres ASCII, EBCDIC, UNICODE…

Tipo de dato NUMBER(p,s)

Almacena valores numéricos en punto flotante que pueden ir desde 1.0 x 10^{-130} hasta 9.9…(38 nueves)… 9 x 10^{125}.

El almacenamiento interno de los valores numéricos en notación científica:

$$\text{Mantisa} \times 10^{\text{exponente}}$$

La mantisa puede contener cualquier número, entero o decimal, positivo o negativo.

El exponente podrá contener cualquier número entero, positivo o negativo.

El parámetro “p” indica la precisión (número de dígitos contando los decimales) que contendrá el número como máximo. Oracle garantiza los datos con precisiones de 1 a 38.

El parámetro “s” indica la escala, esto es, el máximo de dígitos decimales. Hay que tener en cuenta que una columna definida NUMBER(10,5), podrá contener como máximo cualquier número siempre y cuando el número de dígitos enteros más el número de dígitos decimales no supere 10 (y no 15).

La escala puede ir de -84 a 127. Para definir número enteros, se puede omitir el parámetro s o bien poner un 0 en su lugar.

Se puede especificar una escala negativa, esto lo que hace es redondear el número indicado a las posiciones indicadas en la escala. Por ejemplo un número definido como NUMBER(5,-2), redondeará siempre a centenas. Así si intentamos introducir el valor 1355, en realidad se almacenará 1400.

Tipo de dato FLOAT(b)

Almacena un número en punto decimal sin restricción de dígitos decimales.
El parámetro b indica la precisión binaria máxima que puede moverse en el rango 1 a 126. Si se emite el defecto será 126. Una columna FLOAT(126) es equivalente a una columna NUMBER(38), aunque la diferencia está en que la columna NUMBER no podrá contener decimales y la columna FLOAT si y en con cualquier escala.

Tipo de dato DATE

Almacena un valor de fecha y hora.

Para un tipo de dato DATE, Oracle almacena internamente los siguiente datos:

- Siglo
- Año
- Mes
- Día
- Hora
- Minuto
- Segundo

El formato por defecto de las fechas es:

'DD-MON-YYYY'

Esto es:

Dos dígitos para el día
Las tres primeras siglas del año (depende del idioma instalado).
Cuatro dígitos para el año.

Por ejemplo:

'1-JAN-2001' ó '2-DEC-1943'

Este formato puede ser alterado en cualquier momento.

Internamente un fecha se almacena como el número de días desde cierto punto de inicio (por ejemplo el año 0). Esto permite que las fechas puedan ser tratadas en operaciones aritméticas normales:

```
'1-JAN-2001' + 10 = '11-JAN-2001'
'1-JAN-2000' - 1 = '31-DEC-1999'
'10-MAY-2000' - '1-MAY-2000' = 9
```

Tipos de datos binarios

Permiten almacenar información en formato "crudo", valores binarios tal y como se almacenan en el disco duro o como residen en memoria.

Estas columnas se pueden utilizar tanto para almacenar grandes cantidades de datos (hasta 4Gb.), como para almacenar directamente cualquier tipo de fichero (ejecutables, sonidos, videos, fotos, documentos Word, librerías...) o para transportar datos de una base de datos a otra, ya que el formato binario es el único formato común entre cualquier sistema informático.

Tipo de dato LONG

Almacena caracteres de longitud variable hasta 2 Gb.

Este tipo de dato se soporta para compatibilidad con versiones anteriores. En ***Oracle8*** y siguientes versiones se deben usar los tipos de datos CLOB y NLOB para almacenar grandes cantidades de datos alfanuméricos.

Tipo de dato ROWID

Representa una dirección de la base de datos, ocupada por una única fila. El ROWID de una fila es un identificador único para una fila dentro de una base de datos. No hay dos filas con el mismo ROWID. Este tipo de dato sirve para guardar punteros a filas concretas.

El ROWID se compone de:

- Número de *datafile* donde se almacena la fila (se pueden ver en DBA_DATA_FILES)
- Dirección del bloque donde está la fila
- Posición dentro del bloque

Siempre que queramos obtener una fila de la firma más rápida posible, debemos hacerlo a través de su ROWID.

Un uso típico suele ser obtener un listado de ROWIDs con un SELECT, y después acceder a cada una de las filas directamente con la condición del ROWID.

Lenguaje estructurado de consultas SQL

SQL es un conjunto de sentencias u órdenes que todos los programas y usuarios deben utilizar para acceder a bases de datos Oracle. No hay otra manera de comunicarse con Oracle si no es a través de SQL.
Dado que SQL es un estándar, todas las bases de datos comerciales de la actualidad utilizan SQL como puente de comunicación entre la base de datos y el usuario.

Historia

SQL nació como a partir de una publicación de 1970 escrita por E.F. Cood, y titulada "A relational model of data for large shared data banks" (El modelo de datos relacionales para grandes bancos de datos compartidos). IBM utilizó el modelo planteado por Codd para desarrollar un lenguaje capaz de soportar el recién nacido modelo relacional y así apareció SEQUEL (Structured English QUEry Language). SEQUEL más tarde se convirtió en SQL (Structured Query Language) que continuó pronunciándose en inglés como su predecesor: SEQUEL. En 1979, una desconocida empresa llamada Relational Software, sacó por sorpresa al mercado la primera implementación comercial de SQL. Relational Software más tarde pasó a llamarse Oracle.
Después de 20 años, SQL todavía es (y será) siendo el estándar en lenguajes de acceso a base de datos relacionales.

En 1992, ANSI e ISO (organizaciones que se encargan de establecer estándares de todo tipo), completaron la estandarización de SQL y se definió un conjunto de sentencias básicas que debía tener toda implementación para ser llamada estándar. Este SQL se le denominó ANSI-SQL o SQL92.

Hoy en día todas las bases de datos comerciales cumplen el estándar ANSI, aunque cada fabricante añade sus mejoras al lenguaje SQL.

SQL como lenguaje estructurado

En realidad SQL no es un lenguaje en si, como podría ser un lenguaje de programación de 3ª generación (C, Pascal…), sino que en un sublenguaje orientado a acceso y manipulación de base de datos relacionales. Con SQL como única herramienta sólo podemos acceder a las bases de datos, pero no tenemos las estructuras típicas de un lenguaje de programación.
Una buena analogía podría ser un sistema operativo. El interfaz de comandos de un SO nos da todo lo que necesitamos para acceder al sistema de ficheros, pero sólo podemos hacer eso, acceder a ficheros.

SQL actúa de la misma manera, nos da todo lo que necesitamos para acceder a bases de datos, pero no podemos hacer más.

Se dice que SQL es estructurado porque trabaja con conjuntos de resultados (*result set*) abstractos como unidades completas.
Un conjunto de resultados es el esquema básico de una tabla: N filas x N columnas. Este esquema se trata como un todo y es la idea principal de SQL.
A la hora de recuperar un conjunto de resultados, éste se trata de la misma forma tenga el número de filas que tenga (0-N) y tenga el número de columnas que tenga (1-N).
Además SQL es consistente, esto significa que los "estilos" de las distintas sentencias son uniformes, por lo que el aprendizaje es rápido.

Operadores SQL

Ya hemos visto anteriormente qué tipos de datos se pueden utilizar en Oracle. Y siempre que haya datos, habrá operaciones entre ellos, así que ahora se describirán qué operaciones y con qué operadores se realizan:

Los operadores se pueden dividir en dos conjuntos:

· Aritméticos: utilizan valores numéricos
· Lógicos (o booleanos o de comparación): utilizan valores booleanos o lógicos.
· Concatenación: para unir cadenas de caracteres.

Operadores arítméticos
Retornan un valor numérico:

Símbolo	Significado	Ejemplo
+	Operación suma	1 + 2
-	Operación resta	1 - 2
*	Operación multiplicación	1 * 2
/	Operador división	1 / 2

Operadores lógicos

Retornan un valor lógico (verdadero o falso)

Símbolo	Significado	Ejemplo
=	Igualdad	1 = 2
!= <> ^=	Desigualdad	1 != 2 1 <> 2 1 ^= 2
>	Mayor que	1 > 2
<	Menor que	1 < 2
>=	Mayor o igual que	1 >= 2
<=	Menor o igual que	1 <= 2
IN (RS)	Igual a algún elemento del result set.	1 IN (1,2) [TRUE]
<op.> ANY <op.> SOME	<op> a algún elemento del result set (derecha). Debe ser estar precedido por =, !=, <, <=, >, >= Hace un OR lógico entre todos los elementos.	10 >= ANY (1,2,3,10) [TRUE]
<op.> ALL	<op> a todos los elementos del result set (derecha), Debe ser estar precedido por =, !=, <, <=, >, >= Hace un AND lógico entre todos los elementos.	10 <= ALL (1,2,3,10) [TRUE]
BEETWEEN x AND y	Operando de la izquierda entre x e y. Equivalente a op >= x AND op <= y	10 BETWEEN 1 AND 100
EXISTS <subconsulta>	Si la <subconsulta> retorna al menos una fila	EXISTS(SELECT 1 FROM DUAL)
LIKE(*)	Es como	'pepe' LIKE 'pe%'
IS NULL	Si es nulo	1 IS NULL
IS NOT NULL	Si es No nulo	1 IS NOT NULL
NOT cond.	Niega la condición posterios	NOT EXISTS… NOT BETWEEN NOT IN NOT =
cond AND cond	Hace un AND lógico entre dos condiciones	1=1 AND 2 IS NULL
Cond OR cond	Hace un OR lógico entre dos condiciones	1=1 OR 2 IS NULL

(*) El operador LIKE sirve para hacer igualdades con comodines, al estilo * y ? de MS-DOS.
Existen los siguientes comodines:

%: Conjunto de N caracteres (de 0 a ∞)
_: Un solo carácter

Ejemplo:
Las siguientes condiciones retornarán el valor "verdadero" (TRUE)

'significado LIKE 's_gn%fi%d_'
'pepe' LIKE 'pep%' (los que empiecen por 'pep')
'pepote' LIKE 'pep%'
'pepote' LIKE 'pe%te' (los que empiecen por 'pe' y terminen por 'te')
'pedrote' LIKE 'pe%te'

Operador de concatenación
Retornan una cadena de caracteres

Símbolo	Significado	Ejemplo
||	Concatena una cadena a otra	'juan' || 'cito' ['Juancito']

Oracle puede hacer una conversión automática cuando se utilice este operador con valores numéricos:

10 || 20 = '1020'

Este proceso de denomina CASTING y se puede aplicar en todos aquellos casos en que se utiliza valores numéricos en puesto de valores alfanuméricos o incluso viceversa.

La ausencia de valor: NULL

Todo valor (sea del tipo que sea) puede contener el valor NULL que no es más que la ausencia de valor.
Así que cualquier columna (NUMBER, VARCHAR2, DATE…) puede contener el valor NULL, con lo que se dice que la columna *está a NULL*.
Una operación retorna NULL si cualquiera de los operandos es NULL.
Para comprobar si una valor es NULL se utiliza el operador IS NULL o IS NOT NULL.

Lenguaje de manipulación de datos: DML

El DML (*Data Manipulation Language*) es el conjunto de sentencias que está 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.
Básicamente consta de cuatro sentencias: SELECT, INSERT, DELETE, UPDATE.

Instrucción SELECT

La sentencia SELECT es la encargada de la recuperación (selección) de datos, con cualquier tipo de condición, agrupación u ordenación.
Una sentencia SELECT retorna un único conjunto de resultados, por lo que podrá ser aplicada en cualquier lugar donde se espere un conjunto de resultados.

La sintaxis básica es:

```
  SELECT columnas
    FROM tablas
   WHERE condición
GROUP BY columnas de agrupación
  HAVING condición agrupada
ORDER BY columnas de ordenación;
```

Todas las cláusulas son opcionales excepto SELECT y FROM.

A continuación vamos a hacer una descripción breve de cada cláusula:

SELECT

Se deben indicar las columnas que se desean mostrar en el resultado. Las distintas columnas deben aparecer separadas por coma (",").
Opcionalmente puede ser cualificadas con el nombre de su tabla utilizando la sintaxis:

TABLA.COLUMNA

Si se quieren introducir todas las columnas se podrá incluir el carácter *, o bien TABLA.*
Existe la posibilidad de sustituir los nombres de columnas por constantes (1, 'pepe' o '1-may-2000'), expresiones, pseudocolumnas o funciones SQL.

A toda columna, constante, pseudocolumna o función SQL, se le puede cualificar con un nombre adicional:

COLUMNA NOMBRE	CONSTANTE NOMBRE
PSEUDOCOLUMNA NOMBRE	FUNCION SQL NOMBRE

Si se incluye la cláusula DISTINCT después de SELECT, se suprimirán aquellas filas del resultado que tenga igual valor que otras.

Así

```
SELECT C_CLIENTE FROM FACTURA;
```

Puede retornar 1, 3, 5, 5, 1, 7, 3, 2 y 9

Sin embargo, para el mismo caso:

```
SELECT DISTINCT C_CLIENTE FROM FACTURA;
```

Retornará (suprimiendo las repeticiones): 1, 3, 5, 7, 2 y 9

Ejemplos:

```
SELECT REFERENCIA REF, DESCRIPCION
SELECT FACTURA.REFERENCIA, DESCRIPCION
SELECT *
SELECT FACTURA.*
SELECT 1 UN_NUMERO_CTE_CUALIFICADO, REFERENCIA
SELECT 1+1-3*5/5.4 UNA_EXPRESION_SIN_CUALIFICADA
SELECT DESCRIPCION, ROWNUM UNA_PSEUDOCOLUMNA_CUALIFICADA
SELECT TRUNC( '1-JAN-2001'+1, 'MON' ) FUNCION_CUALIFICADA
SELECT DISTINCT *
SELECT DISTINCT DESCRIPCION, IMPORTE
SELECT REFERENCIA||DESCRIPCION
```

FROM

se indican el(los) conjunto(s) de resultado(s) que interviene(n) en la consulta. Normalmente se utilizan tablas, pero se admite cualquier tipo de conjunto (tabla, select, vista…).
Si apareciese más de una tabla, deben ir separadas por coma.
Las tablas deben existir y si no existiera alguna aparecería el siguiente error:

```
ORA-00942: table or view does not exist
```

Al igual que a las columnas, también se puede cualificar a las tablas

TABLA NOMBRE

Oracle tiene definida una tabla especial, llamada DUAL, que se utiliza para consultar valores que no dependen de ninguna tabla.

```
SELECT (1+1.1*3/5)-1-2 FROM DUAL;
```

Ejemplos:

```
FROM FACTURA FAC
FROM FACTURA FAC, CLIENTE CLI
FROM DUAL
FROM ( SELECT C_CLIENTE FROM FACTURA ) CLIENTE_FAC
```

WHERE

Indica qué condiciones debe cumplirse para que una fila entre dentro del conjunto de resultados retornado.

Para construir las condiciones se podrán utilizar todos los operadores lógicos vistos anteriormente.

Es posible construir condiciones complejas uniendo dos o más condiciones simples a través de los operadores lógicos AND y OR.

Ejemplos:

```
WHERE FACTURA.REFERENCIA = 'AA3455'
WHERE FACTURA.C_CLIENTE IS NULL
WHERE C_CLIENTE BETWEEN '12' AND '20'
WHERE C_CLIENTE IS NULL AND
      REFERENCIA IN ('AA23344', 'BB23345')
WHERE C_CLIENTE != 55 OR
      REFERENCIA LIKE 'AA%5_'
```

GROUP BY

La expresión GROUP BY se utiliza para agrupar valores que es necesario procesar como un grupo.

Por ejemplo, puede darse el caso de necesitar procesar todas las facturas de cada cliente para ver su total, o para contarlas, o para incrementarles un 10%... Para estos casos se haría un SELECT agrupando por C_CLIENTE.

Un SELECT con GRUOP BY es equivalente a un SELECT DISTINCT, siempre y cuando en el SELECT no aparezcan consultas sumarias (ver apartado Funciones SQL).

Trataremos con más profundidad este tipo de consultas en el apartado "Consultas agrupadas".

HAVING

Se utiliza para aplicar condiciones sobre agrupaciones. Sólo puede aparecer si se ha incluido la cláusula GROUP BY.
Trataremos con más profundidad este tipo de consultas en el apartado "Consultas agrupadas".

ORDER BY

Se utiliza para ordenar las filas del conjunto de resultados final.
Dentro de esta cláusula podrá aparecer cualquier expresión que pueda aparecer en el SELECT, es decir, pueden aparecer columnas, pseudocolumnas, constantes (no tiene sentido, aunque está permitido), expresiones y funciones SQL. Como característica adicional, se pueden incluir números en la ordenación, que serán sustituidos por la columna correspondiente del SELECT en el orden que indique el número.

La ordenación es el último paso en la ejecución de una consulta SQL, y para ello Oracle suele necesitar crear objetos temporales que son creados en el *tablespace* Temporal. Por eso es recomendable hacer las ordenaciones del lado de cliente (siempre que sea posible), ya que el servidor puede cargarse bastante si tiene que hacer, por ejemplo, 300 ordenaciones de tablas de 2 millones de registros.
Después de cada columna de ordenación se puede incluir una de las palabras reservadas ASC o DESC, para hacer ordenaciones ASCendentes o DESCendentes. Por defecto, si no se pone nada se hará ASC.

Ejemplos:

```
ORDER BY REFERENCIA ASC
ORDER BY REFERENCIA DESC, C_CLIENTE DES, IMPORTE ASC
ORDER BY C_CLIENTE
ORDER BY 1, C_CLIENTE, 2
ORDER BY TRUNC( '1-JAN-2001'+1, 'MON' )
ORDER BY 1.1+3-5/44.3 -- no tiene sentido ordenar por una cte.
```

Consultas agrupadas

Una consulta agrupada se utiliza para considerar los registros cuyos ciertos campos tienen el mismo valor, y procesarlos de la misma manera, para contarlos, sumarlos, hacer la media…
Las consultas típicas son para contar los registros de cierto tipos, sumar los importes de cierto cliente, etc.
Por ejemplo, vamos a sacar el total del importe de las factura, *por cliente:*

```
SELECT C_CLIENTE, SUM(IMPORTE)
FROM   FACTURA
GROUP BY C_CLIENTE;
```

Esto nos sumará (la función SUM suma su parámetro) los registro agrupando por cliente.
Internamente Oracle tiene que hacer una ordenación interna de los registros, según las columnas incluidas en el GROUP BY, así que todo lo dicho para el ORDER BY (sobre la sorbecarga del servidor) se puede aplicar para el GROUP BY.

Cuando en la cláusula SELECT no se incluyen funciones SQL (para más información ver el apartado Funciones SQL), una consulta GROUP BY es equivalente a una consulta SELECT DISTINCT.

Un error muy común cuando se construyen consultas agrupadas, es el siguiente:

```
ORA-00979: not a GROUP BY expression
```

Esto es debido al modo que tiene Oracle de analizar las consultas agrupadas:
Lo que hace es comprobar que todos las columnas incluidos en la cláusula SELECT *fuera de funciones sumarias*, estén dentro de la cláusula GROUP BY, aunque pueden estar en cualquier orden y en el GROUP BY pueden aparecer columnas que no estén en el SELECT.
Si encuentra alguna columna en el SELECT (que no esté dentro de una función sumaria) que no aparezca en el GROUP BY, entonces nos retorna el error anterior.
Si pensamos la situación, es lógico que nos retorne un error, porque no podemos agrupar por la columna C_CLIENTE, si luego queremos mostrar otras columnas que estén sin agrupar. O agrupamos por todo, o mostramos sin agrupar, pero ambas a la vez no es posible.

Ejemplos de consultas agrupadas:

```
SELECT C_CLIENTE, SUM( IMPORTE )
FROM FACTURA
GROUP BY C_CLIENTE;

SELECT C_PAIS, SUM( IMPORTE )
FROM FACTURA
GROUP BY C_PAIS;
```

```
SELECT C_CLIENTE, COUNT(*)
FROM FACTURA
GROUP BY C_CLIENTE;

SELECT C_CLIENTE, SUM(1)
FROM FACTURA
GROUP BY C_CLIENTE;

SELECT C_PAIS, AVG( IMPORTE )
FROM FACTURA
GROUP BY C_PAIS;

SELECT C_PAIS, COUNT(*)
FROM   CLIENTE
GROUP BY C_PAIS,

SELECT C_CLIENTE + AVG( IMPORTE )
FROM   FACTURA;
```

Consultas multitabla

El posible que para consultas sencillas, todos los datos que necesitemos estén en una sola tabla.
Pero… ¿y si están repartidos por una, dos o muchas tablas?

Es posible hacer consultas que incluyan más de una tabla (o conjunto de resultados) dentro de la cláusula FROM, como ya vimos anteriormente.
Pero en estas consultas hay que tener en cuenta ciertos factores.

Veamos lo que hacer Oracle para esta consulta:

```
SELECT F.REFERENCIA, F.C_CLIENTE, C.C_CLIENTE, C.D_CLIENTE
FROM  FACTURA F, CLIENTE C;
```

Suponiendo que tenemos los siguientes datos:

FACTURA	
Referencia	**C_Cliente**
A111	1
A112	2
A113	1
A114	5
A115	2

CLIENTE	
C_Cliente	**D_Cliente**
1	Pepote
2	Juancito
5	Toñete

El *select* anterior nos retornará el siguiente conjunto de resultados:

F.REFERENCIA	F.C_CLIENTE	C.C_CLIENTE	C.D_CLIENTE
A111	**1**	**1**	**Pepote**
A111	1	2	Juancito
A111	1	5	Toñete
A112	2	1	Pepote
A112	**2**	**2**	**Juancito**
A112	2	5	Toñete
A113	**1**	**1**	**Pepote**
A113	1	2	Juancito
A113	1	5	Toñete
A114	5	1	Pepote
A114	5	2	Juancito
A114	**5**	**5**	**Toñete**
A115	2	1	Pepote
A115	**2**	**2**	**Juancito**
A115	2	5	Toñete

Podemos ver que el resultado es el producto cartesiano de una tabla por otra tabla, es decir, todas las combinaciones posibles de la tabla FACTURA con la tabla CLIENTE.

Pero en realidad lo que a nosotros nos interesa es mostrar todas las facturas, pero con la descripción del cliente de cada factura, es decir, que cada factura seleccione sólo su registro correspondiente de la tabla CLIENTE.

Los registros que a nosotros nos interesan están marcados en negrita en el esquema anterior, y en todos ellos se cumple que F.C_CLIENTE = C.C_CLIENTE. O dicho de otro modo, los campos que componen la relación igualados.

Entonces, del conjunto de resultados anterior, sólo nos interesan los registros marcados en negrita, y el *select* que nos retorna ese resultados es:

```
 SELECT F.REFERENCIA, F.C_CLIENTE, C.C_CLIENTE, C.D_CLIENTE
FROM  FACTURA F, CLIENTE C
WHERE F.C_CLIENTE = C.C_CLIENTE;
```

El resultado final es:

F.REFERENCIA	F.C_CLIENTE	C.C_CLIENTE	C.D_CLIENTE
A111	1	1	Pepote
A112	2	2	Juancito
A113	1	1	Pepote
A114	5	5	Toñete
A115	2	2	Juancito

Con la descripción del cliente.

Como norma general se puede decir que para combinar dos o más tablas hay que poner como condición la igualdad entre las claves de una tabla y el enlace de la otra.
Las condiciones dentro del WHERE que sirven para hacer el enlace entre tablas se denominan **JOIN** (unión, enlace).

Nota: en el ejemplo utilizado hemos omitido por simplicidad la columna C_PAIS que también forma parte de la clave, así que el join debería hacerse con las columnas C_PAIS y C_CLIENTE.

Existe un caso especial cuando se establece un *join* entre tablas: el ***outer-join***.
Este caso se da cuando los valores de los campos enlazados en alguna de las tablas, contiene el valor NULL.
Al realizar un *join*, si algún campo enlazado contiene el valor NULL, es registro quedará automáticamente excluido, ya que una condición en la que un operando sea NULL siempre se evalúa como falso.

Supongamos que las tablas utilizadas en el ejemplo anterior ahora tienen los siguientes datos:

FACTURA	
Referencia	**C_Cliente**
A111	1
A112	NULL
A113	1
A114	NULL
A115	7

CLIENTE	
C_Cliente	**D_Cliente**
1	Pepote
2	Juancito
5	Toñete

Si realizamos la misma consulta (las facturas con la descripción de cliente), no aparecerán las facturas "A112" y "A114", ya que su campo C_CLIENTE contiene un NULL, y al evaluar la condición de *join* (WHERE FACTURA.C_CLIENTE = CLIENTE.C_CLIENTE), no se evaluará como verdadero.
Además, tampoco aparecerá la factura "A115", porque el cliente "7" no existe en la tabla de clientes.

Sin embargo, puedes ser que necesitemos mostrar **todas** las facturas de la base de datos, independientemente de si el cliente existe o si el campo está a NULL.

Para ello debemos utilizar un *outer-join*, que no es más que un JOIN con un modificador (+), indicando que queremos considerar aquellos registros que se descarten por existencia de nulos.

El *select* final sería así:

```
SELECT F.REFERENCIA, F.C_CLIENTE, C.C_CLIENTE, C.D_CLIENTE
FROM  FACTURA F, CLIENTE C
WHERE F.C_CLIENTE = C.C_CLIENTE(+);
```

El resultado de ejecutar este *select* es:

F.REFERENCIA	F.C_CLIENTE	C.C_CLIENTE	C.D_CLIENTE
A111	1	1	Pepote
A113	1	1	Pepote
A115	2	7	NULL
A112	NULL	NULL	NULL
A114	NULL	NULL	NULL

Esta consulta podría leerse con el siguiente enunciado:

> "Selecionar las facturas que tengan cliente (el *join*) y aquellas que no encuentren su referencia en la tabla cliente (el *outer-join*)".

Es importante fijarse en la posición en que se ha colocado el modificador (+). Si se sitúa detrás del campo de la tabla cliente, significa que se recuperen las **todas las facturas, aunque no encuentren referencia al cliente**, sin embargo, si lo ponemos detrás del campo de la tabla factura:

```
SELECT F.REFERENCIA, F.C_CLIENTE, C.C_CLIENTE, C.D_CLIENTE
FROM  FACTURA F, CLIENTE C
WHERE F.C_CLIENTE(+) = C.C_CLIENTE;
```

Significaría que recupere **todos los clientes, aunque no encuentre la referencia de la factura**.

Sólo queda por comentar que si el *join* entre las tablas es de varios campos, debe indicarse el símbolo del *outer* (+) en todos los campos, y en la misma posición en todos ellos.

Pseudocolumnas

Una pseudocolumna es una columna válida para poner en cualquier cláusula SELECT, independientemente de las tablas incluidas en la cláusula FROM.
Las pseudocolumnas válidas para *Oracle8* son:

· CURRVAL y NEXTVAL: sólo válidas si el objeto del FROM es una secuencia. Permiten recuperar el valor actual y siguiente (respectivamente) de una secuencia. Para más información sobre las secuencias ir a al apartado CREATE SEQUENCE.

· LEVEL: Retorna el nivel para consultas jerárquicas. Las consultas jerárquicas se realizan utilizando las cláusulas START WITH y CONNECT BY de la sentencia SELECT. Para más información sobre consultas jerárquicas, dirigirse a la ayuda se la sentencia SELECT en el *Oracle8 SQL Reference*.

· ROWID: Retorna una dirección de disco donde se encuentra la fila seleccionada. Es un valor único para cada fila de la base de datos.

· ROWNUM: Es un valor consecutivo para cada fila retornada por una consulta. La primera fila tendrá un 1, la segunda un 2, etc.
Se suele utilizar para restringir el tamaño del conjunto de resultados, por ejemplo, si queremos que sólo retorne las 5 primeras facturas:

```
SELECT *
FROM FACTURA
WHERE ROWNUM <= 5;
```

Hay que tener en cuenta que una consulta de este estilo:

```
SELECT *
FROM FACTURA
WHRE ROWNUM > 1;
```

Nunca retornará resultado, porque siempre habrá una fila que sea la primera. Después de que el WHERE elimine la primera fila, el ROWNUM de todas las filas restantes de recalculará y volverá a haber otra nueva primera fila. Así se seguirá aplicando la condición de filtro hasta que no queden filas. Por eso no retorna ninguna fila.
El valor de ROWNUM se aplica antes de que se ordene el conjunto de resultados (por la cláusula ORDER BY).

· SYSDATE: Nos retorna un tipo de dato DATE con la fecha y hora del sistema (según el reloj del servidor de base de datos).

· USER, UID: Nos retorna el nombre e identificador de usuarios de la sesión activa.

Instrucción INSERT

La sentencia INSERT nos permite introducir nuevas filas en una tabla de base de datos.

La sintaxis básica es:

```
INSERT INTO tabla{( campos )}
VALUES( lista de valores );
```

Los nombres de los campos detrás del nombre de tabla son opcionales y si no se ponen se supondrá todos los campos de la tabla en su orden original. Si se ponen, se podrán indicar cualquier número de columnas en cualquier orden.

La lista de valores es el registro que se insertará en la tabla. Los tipos de datos deben coincidir con la definición dada en la cláusula INTO o con la definición de la tabla si omitimos dicha cláusula.
Las columnas que no se incluyan en el INTO, de inicializarán con NULL, (si no se ha definido valor en el DEFAULT).

Existe otra sintaxis que se denomina INSERT masivo:

```
INSERT INTO tabla{( campos )}
SELECT . . .
```

Este tipo de INSERT permite introducir un gran número de registros en una sola sentencia.
Al igual que con el INSERT normal, los tipos de datos del SELECT deben coincidir con la definición de la cláusula INTO.

Ejemplos:

```
INSERT INTO FACTURA
VALUES('A111', 'Factura nueva', 1, 5, 50000);

INSERT INTO FACTURA(C_PAIS, REFERENCIA, IMPORTE, C_CLIENTE)
VALUES(1, 'A111', 50000, 5);

INSERT INTO FACTURA(REFERENCIA, IMPORTE)
VALUES('A111', 50000);

INSERT INTO FACTURA(C_PAIS, C_CLIENTE)
SELECT C_PAIS, C_CLIENTE
FROM CLIENTE;
```

Instrucción DELETE

La sentencia DELETE nos permite eliminar nuevas filas en una tabla de base de datos conforme a una condición.
Es equivalente al SELECT, pero en vez de mostrar las filas que cumplan la condición, las elimina.

Su sintaxis es:

```
DELETE {FROM} tabla
{WHERE condición};
```

Si se omite la cláusula WHERE se borrarán todas las filas de la tabla.
Las condiciones pueden ser las mismas que las aplicadas en una sentencia SELECT.

En la cláusula FROM no puede haber más de una tabla, por lo que no es posible hacer joins en un DELETE.
Para hacer un "pseudojoin" hay que utilizar el operador IN comparando los campos clave de la tabla a borrar con el subselect de la tabla con la que se quiere hacer el join.

Ejemplos:

```
DELETE FROM FACTURA
WHERE REFERENCIA = 'A111';
```

```
DELETE FACTURA;

DELETE FACTURA
WHERE C_PAIS = 1 AND C_CLIENTE = 5;

DELETE FROM FACTURA
WHERE REFERENCIA NOT IN ( SELECT REFERENCIA
                          FROM   FACTURA
                          WHERE  C_CLIENTE = 4 );

DELETE FROM FACTURA
WHERE C_CLIENTE <> 4;

DELETE FROM FACTURA
WHERE (C_PAIS,C_CLIENTE) IN ( SELECT C_PAIS, C_CLIENTE
                          FROM   CLIENTE
                          WHERE  D_CLIENTE LIKE '%Fernández%' );

DELETE FROM FACTURA
WHERE (C_PAIS,C_CLIENTE) IN ( SELECT C_PAIS, C_CLIENTE
                 FROM   CLIENTE
                 WHERE  CLIENTE.C_PAIS = FACTURA.C_PAIS AND
                 CLIENTE.C_CLIENTE = FACTURA.C_CLIENTE AND
                 D_CLIENTE LIKE '%Fernández%' );
```

Instrucción UPDATE

La sentencia UPDATE se encarga de modificar registros ya existentes en una tabla.
Es equivalente a la sentencia DELETE, pero en vez de borrar, actualiza las columnas indicadas que cumplan la condición impuesta.

Sintaxis:

```
UPDATE tabla
SET   campo = valor,
      campo = valor,
      . . .
{WHERE condición};
```

El valor puede ser tanto un valor discreto (1, 'pepe', '1-jan-2000', etc), un valor dependiente de otra una columna (IMPORTE*10) o un subselect que retorne un conjunto de resultados de 1x1 (1 fila y 1 una columna). Si se utiliza un subselect se puede hacer *join* entre este subselect y la tabla del UPDATE.
Si se omite la cláusula WHERE, se actualizarán todas las filas de la tabla.

Ejemplos:

```
UPDATE FACTURA
SET   IMPORTE = 1000
WHERE C_PAIS = 1 AND C_CLIENTE = 5;

UPDATE FACTURA
SET   IMPORTE = IMPORTE * 0.5
WHERE C_PAIS = 1 AND C_CLIENTE = 5;

UPDATE FACTURA F1
SET   IMPORTE = ( SELECT AVG(IMPORTE) * 1.10
               FROM  FACTURA F2
               WHERE F1.C_PAIS    = F2.C_PAIS AND
                      F1.C_CLIENTE = F2.C_CLIENTE );

UPDATE FACTURA F1
SET   IMPORTE = ( SELECT AVG(F2.IMPORTE) + F1.IMPORTE
               FROM   FACTURA F2
               WHERE  F1.C_PAIS     = F2.C_PAIS AND
                       F1.C_CLIENTE = F2.C_CLIENTE );
```

Lenguaje de definición de datos: DDL

El DDL (Data Definition Language) es el conjunto de sentencias que está orientadas a la creación, modificación y configuración de objetos en base de datos.
El DDL es el subconjunto más extenso dentro de SQL así que sólo vamos a hacer una referencia rápida a algunas sentencias.
Se puede encontrar una descripción detallada del todo el DDL dentro del ***Oracle8 SQL Reference.***

CREATE TABLE

Crea una tabla en base de datos.

La sintaxis básica es:

```
CREATE TABLE nombre_tabla(
      COLUMNA     TIPO  [NOT NULL],
      COLUMNA     TIPO  [NOT NULL],
      . . .
{CONSTRAINT nombre_clave_primaria PRIMARY KEY (columnas_clave)}
{CONSTRAINT nombre_clave_foránea
    FOREIGN    KEY(columnas_clave)    REFERENCES    tabla_detalle(
columnas_clave )
 {ON DELETE CASCADE} }  )
{TABLESPACE tablespace_de_creación}
{STORAGE( INITIAL XX{K|M} NEXT XX{K|M} )}
```

La creación de la tabla FACTURA, definida en el capítulo “Concepto de índice”, sería la siguiente:

```
CREATE TABLE FACTURA(
      REFERENCIA  VARCHAR2(10)      NOT NULL,
      DESCRIPCION VARCHAR2(50),
      C_PAIS      NUMBER(3),
      C_CLIENTE   NUMBER(5),
      IMPORTE     NUMBER(12),
CONSTRAINT PK_FACTURA PRIMARY KEY( REFERENCIA )
CONSTRAINT FK_CLIENTE(C_PAIS,C_CLIENTE)
            REFERENCES CLIENTE(C_PAIS, C_CLIENTE)
ON DELETE CASCADE
```

```
TABLESPACE tab_facturas
STORAGE( INITIAL 1M NEXT 500K );
```

Los campos que van a formar parte de la clave se tienen que definir como NOT NULL ya que no tiene sentido que éstas columnas carezcan de valor.
Además se crea la clave primaria y la clave foránea que hace referencia a CLIENTE (como ya dijimos en el apartado de claves foráneas). Con la cláusula ON DELETE CASCADE hacemos que si se borra un cliente, se borren automáticamente todas sus facturas.
Si no la incluyésemos, al borrar el cliente, nos daría el siguiente error:

```
ORA-02292: integrity constraint (FK_CLIENTE) violated - child record found
```

Hay que tener en cuenta que aunque la clave foránea se crea sobre la tabla FACTURA, también actúa cuando se hacen operaciones sobre CLIENTE.

Al intentar insertar una factura a un cliente inexistente nos dará el siguiente error:

```
ORA-02291: integrity constraint (FK_CLIENTE) violated - parent key not found
```

Se puede encontrar una descripción detallada de todos los errores en el *Oracle8 Error Messages*.

Sobre la cláusula STORAGE hablaremos al final de este capítulo.

CREATE INDEX

Crea un índice sobre una tabla de base de datos.
La sintaxis básica es:

```
CREATE {UNIQUE} INDEX nombre_índice
ON tabla( columnas_indexadas )
{TABLESPACE tab_indices}
{STORAGE( INITIAL XX{K|M} NEXT XX{K|M} )}
```

La cláusula UNIQUE actúa como si los campos indexados fuesen clave primaria, es decir, no permite que el conjunto de campos indexados se repita en la tabla.
Ya dijimos que un índice es como una tabla auxiliar que sólo contiene ciertas columnas de búsqueda. Por eso también es posible (y recomendable) indicar tanto el *tablespace* como las

cláusula STORAGE para las características de almacenamiento de disco. Si no se incluyera, se utilizará el STORAGE indicado en la creación del *tablespace* sobre el que se crea el índice.

Además, Oracle recomienda que los índices residan en un *tablespace* y *datafile* separado al de las tablas. Esto es debido a la siguiente razón:
Dos *tablespaces* distintos pueden están almacenados físicamente por, al menos, un *datafile* cada uno. Si nuestro servidor de base de datos tiene más de un disco duro (algo muy normal), es posible crear un *tablespace* con sus *datafiles* en un disco y otro *tablespace* con los *datafiles* en otro disco. Esto permite que se puedan hacer lecturas de disco simultáneamente sobre dos disco físicos, ya que cada disco tiene su propio *bus* de datos. Al crear los índices en un disco físico y los datos en otro, se facilita que se puedan hacer lecturas simultaneas.
Este proceso (de poner los índices y datos en discos separados), se denomina *balanceado*.

Oracle crea automáticamente un índice cuando se define la clave primaria. Esto es debido a que la condición más habitual en una consulta a cualquier tabla es a través de los campos de su clave primaria. De esta forma se aceleran la gran mayoría de las consultas (recordar que el índice de una tabla actúa del mismo modo que el de un libro).
Pero pueden darse casos en los que se hagan gran cantidad de consultas por campos distintos a los de la clave primaria. En este caso es necesario crear un índice por los campos por lo que se accede.
Por ejemplo, puede ser que en nuestra tabla FACTURA sea muy común recuperar aquellas facturas de un cierto cliente.
En este caso la consulta SELECT ha realizar sería la siguiente:

```
SELECT *
FROM FACTURA
WHERE C_PAIS = 1 AND
      C_CLIENTE = 'A111';
```

En este caso se está accediendo la tabla FACTURA por campos distintos a la clave primaria (que es Referencia). Si este tipo de consultas son muy habituales es necesario crear un índice por estos campos:

```
CREATE INDEX ind_factura_cliente
ON FACTURA( C_PAIS, C_CLIENTE )
TABLESPACE tab_factura_ind
STORAGE( INITIAL 500K NEXT 500K );
```

No podemos poner la cláusula UNIQUE porque si no, no podríamos insertar más de una factura por cliente.

CREATE VIEW

Una vista (*view*) es una consulta SELECT almacenada en base de datos con un cierto nombre.
Si tenemos la siguiente consulta:

```
SELECT C.D_CLIENTE, SUM( F.IMPORTE )
FROM   FACTURA F, CLIENTE C
WHERE  F.C_PAIS    = C.C_PAIS AND
       F.C_CLIENTE = C.C_CLIENTE
GROUP BY F.C_PAIS, F.C_CLIENTE, C.D_CLIENTE;
```

Si esta consulta se repite es necesaria muchas veces, entonces podemos guardar esta definición en base de datos con un nombre (crear una vista), y después hacer la consulta sobre la vista.

```
CREATE VIEW TOTAL_FACTURA_CLIENTE AS
SELECT C.D_CLIENTE, SUM( F.IMPORTE )
FROM   FACTURA F, CLIENTE C
WHERE  F.C_PAIS    = C.C_PAIS AND
       F.C_CLIENTE = C.C_CLIENTE
GROUP BY F.C_PAIS, F.C_CLIENTE, C.D_CLIENTE;
```

Y después hacer la consulta sobre la vista:

```
SELECT *
FROM TOTAL_FACTURA_CLIENTE;
```

La sintaxis de creación de vista es:

```
CREATE {OR REPLACE} {FORCE} VIEW nombre_vista AS
Subconsulta;
```

La cláusula OR REPLACE permite sobrescribir una definición existente con otra nueva definición.
La cláusula FORCE permite crear una vista aunque las tablas de la subconsulta no existan.

Ejemplos:

```
CREATE OR REPLACE FORCE VIEW VISTA_INCORRECTA AS
SELECT * FROM FACTURA_ADICIONAL;

CREATE OR REPLACE VIEW FACTURA_CLIENTE_A111 AS
SELECT * FROM FACTURA
WHERE C_PAIS = 1 AND C_CLIENTE = 'A111';
```

CREATE SYNONYM

Un sinónimo (*synonym*) es una redefinición de nombre de un objeto en base de datos.
Así al objeto FACTURA podemos crearle un sinónimo y entonces se podrá acceder a él tanto con el nombre de FACTURA como con el nombre del sinónimo.

Sintaxis:

```
CREATE {PUBLIC} SYNONYM nombre_sinónimo
FOR {usuario.}objeto;
```

La cláusula PUBLIC permite que todos los usuarios de base de datos puedan acceder al sinónimo creado. Para más información sobre los usuario de base de datos ir a la sección *Administración básica y seguridad en Oracle.*

Ejemplos:

```
CREATE SYNONYM FACTURAS
FOR FACTURA;

CREATE SYNONYM FACTURA_SCOTT
FOR SCOTT.FACTURA;

CREATE PUBLIC SYNONYM BILL
FOR FACTURA;
```

CREATE SEQUENCE

Una secuencia (*sequence*) es un objeto de base de datos que genera números secuenciales. Se suele utilizar para asignar valores a campos autonuméricos.

En realidad una secuencia no es más que una tabla con una columna numérica en la que se almacena un valor. Cada vez que se consulta la secuencia se incrementa el número para la siguiente consulta.

Sintaxis:

```
CREATE SEQUENCE nombre_secuencia
{START WITH entero}
{INCREMENT BY entero}
{MAXVALUE entero | NOMAXVALUE}
{MINVALUE entero | NOMINVALUE }
{CYCLE | NOCYCLE};
```

- La cláusula START WITH define el valor desde el que empezará la generación de números. Si no se incluye, se empezará a partir de MINVALUE.
- La cláusula INCREMENT BY indica la diferencia que habrá entre un número y el siguiente. Puede ser cualquier número entero (positivo o negativo) distinto de 0.
- La cláusula MAXVALUE indica el valor máximo que podrá alcanzar la secuencia. Se podrá incluir la cláusula NOMAXVALUE para no definir máximo de 10^{27}.
- La cláusula MINVALUE indica el valor mínimo de la secuencia. Se podrá incluir la cláusula NOMINVALUE para definir un mínimo de -10^{26}.
- La cláusula CYCLE permite que se empiece a contar en MINVALUE cuando se llegue a MAXVALUE. Por defecto las secuencias se crean NOCYCLE.

Ejemplos:

```
CREATE SEQUENCE REF_FACTURA
START WITH 1
INCREMENT BY 1
MAXVALUE 999999
MINVALUE 1;

CREATE SEQUENCE COD_CLIENTE
INCREMENT BY 10;

CREATE SEQUENCE COD_PAIS
INCREMENT BY 10
CYCLE;
```

Acceso a secuencias

Las secuencias al ser tablas se acceden a través de consultas SELECT. La única diferencia es que se utilizan pseudocolumnas para recuperar tanto el valor actual como el siguiente de la secuencia. Al ser pseudocolumnas se puede incluir en el FROM cualquier tabla o bien la tabla DUAL.

Nombre_secuencia.CURRVAL: retorna el valor actual de la secuencia.
Nombre_secuencia.NEXTVAL: incrementa la secuencia y retorna el nuevo valor.

Ejemplos:

```
SELECT REF_FACTURA.CURRVAL FROM DUAL;
```

```
SELECT COD_CLIENTE.NEXTVAL
FROM DUAL;

SELECT COD_CLIENTE.NEXTVAL, D_CLIENTE
FROM CLIENTE;

UPDATE CLIENTE
SET CODIGO = SECUENCIA_CLIENTE.NEXTVAL;

INSERT INTO CLIENTE
VALUES( SECUECIA_CLIENTE.NEXTVAL, 'Juancito Pérez Pí' );
```

CREATE TABLESPACE

Ya hemos visto el concepto de *tablespace* y su importancia en el acceso a datos en Oracle. Ahora vamos a ver cómo se crean los *tablespaces*.

La sintaxis básica es:

```
CREATE TABLESPACE nombre_tablespace
DATAFILE 'ruta\fichero_datafile.ext' SIZE XX{K|M}
{DEFAULT STORAGE( INITIAL XX{K|M} NEXT XX{K|M} )}
{ONLINE | OFFLINE}
{PERMANENT | TEMPORARY};
```

La cláusula DATAFILE indica la localización del fichero de datos (*datafile*) que soportará el *tablespace*. Para añadir más *datafiles* al *tablespace* lo podremos hacer a través de la sentencia ALTER TABLESPACE.

Cuando indicamos el tamaño a través de la cláusula SIZE, le estamos diciendo al *datafile* que reserve cierto espacio en disco, aunque inicialmente ese espacio esté vacío. Conforme vayamos creando objetos sobre este *tablespace*, ese espacio reservado se irá llenando.

La cláusula DEFAULT STORAGE indica qué características de almacenamiento por defecto se aplicará a los objetos creados sobre el *tablespace*. Si no incluimos la cláusula STORAGE en la creación de un objeto (por ejemplo en el CREATE TABLE), se aplicarán las características definidas en la creación del *tablespace*.

El *tablespace* inicialmente podrá estar en línea o fuera de línea a través de las cláusulas ONLINE y OFFLINE. Por defecto el *tablespace* se creará en estado ONLINE.

Para *tablespaces* temporales se deberá incluir la cláusula TEMPORARY, para los de datos la cláusula PERMANENT. Por defecto el *tablespace* será PERMANENT.

Ejemplos:

```
CREATE TABLESPACE tab_factura
DATAFILE 'C:\ORANT\DATABASE\tablespace_facturas.dat'
SIZE 100M
DEFAULT STORATE( INITIAL 100K NEXT 100K )
ONLINE
PERMANENT;

CREATE TABLESPACE tab_temporal
DATAFILE 'C:\ORANT\DATABASE\tablespace_tmp.ora' SIZE 50M
OFFLINE
TEMPORARY;

CREATE TABLESPACE tab_indices
DATAFILE 'C:\ORANT\DATABASE\tab_indices.tab' SIZE 10M;
```

Sentencias DROP

Toda sentencia de creación CREATE tiene su equivalente para eliminar el objeto creado.
Todas estas sentencias tienen la misma sintaxis:

```
DROP tipo_objeto objeto_a_borrar.
```

Por ejemplo:

```
DROP TABLE FACTURA;

DROP SEQUENCE COD_CLIENTE;

DROP SYNONYM BILL;

DROP VIEW TOTAL_FACTURA_CLIENTE;

DROP TABLESPACE tab_indices;
```

Ciertas sentencias DROP (como DROP TABLE o DROP TABLESPACE) tienen cláusulas adicionales para ciertas situaciones especiales. Para más información buscar la ayuda de la sentencia necesitada en el *Oracle8 SQL Reference*.

Sentencias ALTER

Al igual que existe una sentencia DROP para cada objeto creado, también existe una sentencia ALTER para cada objeto de base de datos.
Con estos tres grupos de sentencias se hace la gestión completa de los objeto: creación, modificación y borrado.
La sintaxis básica de las sentencias ALTER es:

```
ALTER tipo_objeto nombre_objeto
Cláusulas específicas de cada tipo de ALTER;
```

Las cláusulas propias de cada sentencia ALTER son muchas y variadas, por lo que aquí no se citarán más que ciertos ejemplos. Para más información dirigirse la ayuda de la sentencia necesitada en el *Oracle8 SQL Reference*.

Ejemplos:

```
ALTER TABLE FACTURA ADD(
      NUEVA_COLUMNA  VARCHAR2(10) NOT NULL );

ALTER VIEW BILL COMPILE;

ALTER TABLESPACE tab_indices ADD(
DATAFILE 'C:\ORANT\DATABASE\otro_datafile.ora' SIZE 5M;

ALTER TABLESPACE tab_indices
RENAME DATAFILE
'C:\ORANT\DATABASE\nombre.ora' TO 'C:\ORANT\DATABASE\otro_nombre.ora'

ALTER TABLESPACE tab_indices COALESCE;

ALTER SEQUENCE NOCYCLE;
```

La sentencia TRUNCATE

La sentencia TRUNCATE pertenece al conjunto de las sentencias DDL, y permite vaciar todos los registros de una tabla.
Aparentemente es equivalente a hacer un DELETE sin condición, pero en realidad no es igual, ya que DELETE pertenece al subconjunto de DDL y TRUNCATE al DML.
La sintaxis básica es:

```
TRUNCATE nombre_tabla {DROP|REUSE STORAGE}
```

La cláusula DROP STORAGE eliminará todas las extents creadas durante la vida de la tabla.

Ejemplos:

```
TRUNCATE FACTURA DROP STORAGE;

TRUNCATE CLIENTE;
```

Cláusula STORAGE

Todo objeto que tenga ocupación física en la base de datos, tendrá una cláusula *storage* en su sintaxis de creación.
El objetivo de esta cláusula es definir ciertas propiedades de almacenamiento para el objeto creado, como puede ser tamaño de la extensión inicial, tamaño de las siguientes extensiones…

Sintaxis:

```
STORAGE( INITIAL entero{K|M} NEXT entero{K|M}
      {MINEXTENTS entero}
      {MAXEXTENTS entero|UNLIMITED}
      {PCTINCREASE %entero} )
```

La cláusula INITIAL define el tamaño que tendrá al extensión inicial y NEXT el tamaño de las siguientes extensiones.
La cláusula MINEXTENTS indica el número mínimo de extensiones para el objeto, y MAXEXTENTS indica el máximo número de extensiones (puede ser UNLIMITED, aunque no es aconsejable).
PCTINCREASE indica el porcentaje en que se aumentará el tamaño de un "next extent". Para que todas las extensiones adicionales sean del mismo tamaño se debe indicar 0.

Funciones SQL

Las funciones SQL permiten mostrar columnas calculadas dentro de sentencias DML (SELECT, INSERT, DELETE y UPDATE).

Funciones de tratamiento numérico

Función	Descripción
ABS(n)	Retorna el valor absoluto del parámetro.
CEIL(n)	Retorna el entero mayor del parámetro.
FLOOR(n)	Retorna el entero menor del parámetro.
MOD(m,n)	Retorna el resto de la división m/n
POWER(m,n)	Retorna m^n
ROUND(m[,n])	Retorna m, redondeado a n decimales. Si m se omite es 0.
SIGN(n)	Retorna 1 si n es positivo, -1 si negativo y 0 si es 0.
TRUNC(n[,m])	Trunca un número a m decimales. Si m se omite es 0.

Funciones de tratamiento alfanumérico

Función	Descripción
CHR(n)	Retorna el carácter equivalente al código n en la tabla de conjunto de caracteres utilizado (ASCII, UNICODE...)
CONCAT(s1, s2)	Concatena dos cadenas de caracteres. Equivalente al operador ||
INITCAP(s)	Pasa el mayúscula la primera letra de cada palabra
LOWER(s)	Pasa a minúsculas toda la cadena de caracteres
LPAD(s, n)	Retorna los n primeros caracteres de la cadena s.
RPAD(s, n)	Retorna los n últimos caracteres de la cadena s.
LTRIM(s1[, s2])	Elimina todas las ocurrencias de s2 en s1 por la izquierda. Si se omite s2, se eliminarán los espacios.
RTRIM(s1[, s2])	Elimina todas las ocurrencias de s2 en s1 por la derecha. Si se omite s2, se eliminarán los espacios.
REPLACE(s1, s2, s3)	Retorna s1 con cada ocurrencia de s2 reemplazada por s3.
SUBSTR(s, m, n)	Retorna los n caracteres de s desde la posición m.
UPPER(s)	Pasa a mayúsculas toda la cadena de caracteres
LENGTH(s)	Retorna la longitud (en caracteres) de la cadena pasada.

Funciones de tratamiento de fechas

Función	Descripción
ADD_MONTHS(d, n)	Suma un número (positivo o negativo) de meses a una fecha.
LAST_DAY(d)	Retorna el ultimo día de mes de la fecha pasada.
MONTHS_BETWEEN(d1, d2)	Retorna la diferencia en meses entre dos fechas.
ROUND(d, s)	Redondea la fecha d según el formato indicado en s. (*)
TRUNC(d, s)	Trunca la fecha d según el formato indicado en s. (*)

Formatos para ROUND y TRUNC para fechas:

Formato	Descripción
'MONTH', 'MON', 'MM'	Principio de mes
'DAY', 'DY', 'D'	Principio de semana
'YEAR', 'YYYY', 'Y'	Principio de año

Funciones de grupo

Estas funciones actúan sobre un conjunto de valores, retornando sólo un registro.

Función	Descripción
SUM(valores)	Retorna la suma.
AVG(valores)	Retorna la media aritmética
MAX(valores)	Retorna el máximo.
MIN(valores)	Retorna el mínimo
COUNT(valores|*)	Retorna la cuenta.
Todas estas funciones permite incluir el modificador DISTINCT delante de la lista de valores para que omita los repetidos.	

Funciones de conversión

Función	Descripción
CHARTOROWID(s)	Convierte una cadena en tipo de dato ROWID.
ROWIDTOCHAR(rowid)	Convierte un tipo de dato ROWID en cadena de caracteres.
TO_CHAR(*[, s])	Convierte el tipo de dato * en cadena de caracteres. Si * es una fecha, se podrá utilizar la cadena s como formato de conversión.
TO_DATE(s1[, s2])	Convierte la cadena s1 en fecha, conforme al formato de convesión s2.
TO_NUMBER(s)	Convierte una cadena de caracteres en valor numérico.

Otras funciones

Función	Descripción
DUMP(columna)	Retorna información de almacenamiento para la columna indicada.
GREATEST(expr1, expr2, ... exprN)	Retorna la expresión mayor.
LEAST(expr1, expr2, ... exprN)	Retorna la expresión menor.
NVL(expr1, expr2)	Retorna expr2 si expr1 es NULL, sino retorna expr1.
USEREVN(s)	Retorna opciones de entorno de la sesión activa: Los valores para s pueden ser: · 'ISDBA': Retorna 'TRUE' si el usuario activo tiene el rol DBA. · 'LANGUAGE': Idioma activo. · 'TERMINAL': Nombre de terminal donde se realiza la consulta. · 'SESSIONID': Número de sesión activa.
DECODE(expr_ev, Caso_1, ret_1, Caso_2, ret_2, ... Caso_N, ret_N, Caso_else)	Permite hace una evaluación de valores discretos. Es similar a la estructura switch de C/C++ o case of de Pascal. Ejemplo: DECODE(COLOR, 'R', 'Rojo', 'V', 'Verde', 'A', 'Azul', 'Color desconocido')

Control de transacciones

Debido a que en las operaciones normales con la base de datos puede corromper la información, todas las bases de datos tiene un sistema de control de transacción.

Se denomina transacción al espacio de tiempo que transcurre desde la primera sentencia DML que no sea SELECT (INSERT, UPDATE, DELETE), hasta que damos por finalizada la transacción explícitamente (con las sentencias apropiadas) o implícitamente (terminando la sesión).

Durante la transacción, todas las modificaciones que hagamos sobre base de datos, no son definitivas, más concretamente, se realizan sobre un *tablespace* especial que se denomina *tablespace* de ROLLBACK, o RBS (RollBack Segment). Este *tablespace* tiene reservado un espacio para cada sesión activa en el servidor, y es en ese espacio donde se almacenan todas las modificaciones de cada transacción. Una vez que la transacción se ha finalizado, las modificaciones temporales almacenadas en el RBS, se vuelcan al *tablespace* original, donde está almacenada nuestra tabla. Esto permite que ciertas modificaciones que se realizan en varias sentencias, se puedan validar todas a la vez, o rechazar todas a la vez.

Las sentencias de finalización de transacción son:

COMMIT: la transacción termina correctamente, se vuelcan los datos al *tablespace* original y se vacía el RBS.

ROLLBACK: se rechaza la transacción y el vacía el RBS.

Opcionalmente existe la sentencia SAVEPOINT para establecer puntos de transacción.
La sintaxis de SAVEPOINT es:

```
SAVEPOINT nombre_de_punto;
```

A la hora de hacer un ROLLBACK o un COMMIT se podrá hacer *hasta* cierto punto con la sintaxis:

```
COMMIT TO nombre_de_punto;
ROLLBACK TO nombre_de_punto;
```

Ejemplo de transacción:

Histórico de sentencias:

```
SELECT;
SELECT;
SELECT;
UPDATE;
SELECT;
INSERT;
UPDATE;
SELECT;
UPDATE;
COMMIT;
```

Transacción

```
UPDATE;
SELECT;
SAVEPOINT pepe
INSERT;
UPDATE;
SELECT;
UPDATE;
ROLLBACK TO pepe;
UPDATE;
INSERT;
DELETE;
COMMIT TO pepe;
SELECT;
COMMIT;
```

Transacción

Si nuestro número de sentencias es tan grande que el RBS se llena, Oracle hará un ROLLBACK, por lo que perderemos todos los datos. Así que es recomendable hacer COMMIT cada vez que el estado de la base de datos sea consistente.

Si terminamos la sesión con una transacción pendiente, Oracle consultará el parámetro AUTOCOMMIT, y si éste está a TRUE, se hará COMMIT, si está FALSE se hará ROLLBACK;

Administración básica y seguridad en Oracle

Concepto de usuario, privilegio y rol:

A la hora de establecer una conexión con un servidor Oracle, es necesario que utilicemos un modo de acceso, el cual describa de qué permisos dispondremos durante nuestra conexión. Estos permisos se definen sobre un nombre de usuario.
Un usuario no es más que un conjunto de permisos que se aplican a una conexión de base de datos.
Así mismo, el usuario también tiene otras funciones:

· Ser el propietario de ciertos objetos.
· Definición del *tablespace* por defecto para los objetos de un usuario.
· Copias de seguridad.
· Cuotas de almacenamiento.

Un privilegio no es más que un permiso dado a un usuario para que realice cierta operación. Estas operaciones pueden ser de dos tipos:

· Operación de sistema: necesita el permiso de sistema correspondiente.
· Operación sobre objeto: necesita el permiso sobre el objeto en cuestión.

Y por último un rol de base de datos no es más que una agrupación de permisos de sistema y de objeto.

Creación de usuarios

La creación de usuarios se hace a través de la sentencia SQL CREATE USER
Su sintaxis básica es:

```
CREATE USER nombre_usuario
IDENTIFIED [ BY clave | EXTERNALLY ]
{ DEFAULT TABLESPACE tablespace_por_defecto }
{ TEMPORARY TABLESPACE tablespace_temporal }
{ DEFAULT ROLE [ roles, ALL [EXCEPT roles], NONE ] };
```

La cláusula IDENTIFIED BY permite indicar el tipo de autorización que se utilizará:

· Interna de Oracle: una clave para cada usuario de base de datos.

· Interna del SO: utilizando la seguridad del SO.

La cláusula DEFAULT TABLESPACE será el *tablespace* por defecto en la creación de objetos del usuario que estamos creando. Si se omite se utilizará el *tablespace* SYSTEM.

La cláusula TEMPORARY TABLESPACE indica el *tablespace* que se utilizará para la creación de objetos temporales en la operaciones internas de Oracle. Si se omite se utilizará el *tablespace* SYSTEM.

La cláusula DEFAULT ROLE permite asignar roles de permisos durante la creación del usuario.

Ejemplos:

```
CREATE USER ADMINISTRADOR
IDENTIFIED BY MANAGER
DEFAULT TABLESPACE SYSTEM
TEMPORARY TABLESPACE TEMPORARY_DATA
DEFAULT ROLE DBA;

CREATE USER PEPOTE
IDENTIFIED BY TORO;

CREATE USER JUANCITO
IDENTIFIED BY PEREZ
DEFAULT TABLESPACE DATOS_CONTABILIDAD
TEMPORARY TABLESPACE TEMPORARY_DATA;
```

Creación de roles

La creación de roles permite asignar un grupo de permisos a un usuario, y poder modificar este grupo de permisos sin tener que ir modificando todos los usuarios.
Si asignamos un rol con 10 permisos a 300 usuarios, y posteriormente añadimos un permiso nuevo al rol, no será necesario ir añadiendo este nuevo permiso a los 300 usuarios, ya que el rol se encarga automáticamente de propagarlo.
La sintaxis básica es:

```
CREATE ROLE nombre_rol
{ [NOT IDENTIFIED | IDENTIFIED [BY clave | EXTERNALLY]] };
```

Una vez que el rol ha sido creado será necesario añadirle permisos a través de instrucción GRANT.

Inicialmente Oracle tiene predefinidos los siguiente roles (entre otros):

Rol predefinido	Descripción
CONNECT	Todos los permisos necesarios para iniciar una sesión en Oracle
RESOURCE	Todos los permisos necesarios para tener recursos para la creación de objetos
DBA	Todos los permisos para un administrador de base de datos (DBA)
EXP_FULL_DATABASE	Permisos para poder exportar toda la base de datos.
IMP_FULL_DATABASE	Permisos para poder importar toda la base de datos.

Podemos decir que un usuarios normal, debe tener al menos los permisos de CONNECT (para conectarse) y de RESOURCE (para poder crear objetos).

Ejemplos:

```
CREATE ROL CONTROL_TOTAL;

CREATE ROL BASICO;

CREATE ROL ACCESO_CONTABILIDAD;
```

Privilegios de sistema

Ya hemos dicho que los privilegios de sistema son permisos para realizar ciertas operaciones en la base de datos.
El modo de asignar un privilegio es a través de la instrucción GRANT y el modo de cancelar un privilegio es a través de la instrucción REVOKE.
La sintaxis básica de ambas instrucciones es:

Instrucción GRANT

```
GRANT [privilegios_de_sistema | roles]
TO [usuarios | roles |PUBLIC]
{ WITH ADMIN OPTION };
```

Es posible dar más de un privilegio de sistema o rol, separándolos por comas.
También es posible asignarle uno (o varios) privilegios a varios usuarios, separándolos por comas.
Si se le asigna el privilegio a un rol, se asignará a todos los usuarios que tengan ese rol.
Si se asigna el privilegio a PUBLIC, se asignará a todos los usuarios actuales y futuros de la base de datos.
La cláusula WITH ADMIN OPTION permite que el privilegio/rol que hemos concedido, pueda ser concedido a otros usuarios por el usuario al que estamos asignando.
La lista de los privilegios de sistema existentes se puede encontrar en el *Oracle8 SQL Reference* en la sección GRANT (System privileges and roles).

Ejemplos:

```
GRANT DBA TO ADMINISTRADOR;

GRANT CREATE USER TO PEPOTE WITH ADMIN OPTION;

GRANT DROP USER TO JUANCITO;

GRANT CONNECT, RESOURCE TO PEPOTE, JUANCITO;

GRANT CONNECT, RESOURCE, DBA, EXP_FULL_DATABASE, IMP_FULL_DATABASE
TO CONTROL_TOTAL;

GRANT CONTROL_TOTAL TO ADMINISTRADOR;
```

Instrucción REVOKE

```
REVOKE [privilegios_de_sistema | roles]
FROM [usuarios | roles |PUBLIC];
```

Es posible eliminar más de un privilegio de sistema o rol, separándolos por comas.
También es posible eliminar uno (o varios) privilegios a varios usuarios, separándolos por comas.
Si se le elimina el privilegio de un rol, se eliminará de todos los usuarios que tengan ese rol.
Si se elimina el privilegio de PUBLIC, se eliminará de todos los usuarios actuales y futuros de la base de datos.
La lista de los privilegios de sistema existentes se puede encontrar en el *Oracle8 SQL Reference* en la sección GRANT (System privileges and roles).
Como es lógico, sólo se podrá eliminar un privilegio/rol, si previamente ha sido concedido a través de la instrucción GRANT.

Ejemplos:

```
REVOKE DBA FROM ADMINISTRADOR;

REVOKE CREATE USER FROM PEPOTE;

REVOKE DROP USER FROM JUANCITO;

RECOKE CONNECT, RESOURCE FROM PEPOTE, JUANCITO;

REVOKE CONNECT, RESOURCE, DBA, EXP_FULL_DATABASE, IMP_FULL_DATABASE
FROM CONTROL_TOTAL;

REVOKE CONTROL_TOTAL FROM ADMINISTRADOR;
```

Privilegios sobre objetos

Los privilegios sobre objetos permiten que cierto objeto (creado por un usuario) pueda ser accedido por otros usuarios. El nivel de acceso depende del permiso que le demos: podemos darle permiso de SELECT, de UPDATE, de DELETE, de INSERT o de todos ellos.
La sintaxis básica es:

```
GRANT [ALL {PRIVILEGES} | SELECT | INSERT | UPDATE | DELETE]
ON objeto
TO [usuario | rol | PUBLIC]
{WITH ADMIN OPTION};
```

Al igual que con los permisos de sistema, es posible asignar un permiso de objeto sobre uno o varios (separados por comas) usuario y/o roles. Si se asigna a PUBLIC será accesible en toda la base de datos.

Si se incluye la cláusula WITH ADMIN OPTION, este permiso podrá ser concedido por el usuario al que se le ha asignado.

Ejemplos:

```
GRANT ALL ON FACTURA TO CONTROL_TOTAL;

GRANT SELECT, UPDATE ON ALUMNO TO PEPOTE, JUANCITO
WITH ADMIN OPTION;

GRANT SELECT ON PROFESOR TO PUBLIC;

GRANT SELECT ON APUNTE TO ACCESO_CONTABILIDAD;
```

El modo de eliminar permisos de objeto es con la instrucción REVOKE:

```
REVOKE [ALL {PRIVILEGES} | SELECT | INSERT | UPDATE | DELETE]
ON objeto
FROM [usuario | rol | PUBLIC]
{WITH ADMIN OPTION};
```

Al igual que con los permisos de sistema, es posible asignar un permiso de objeto sobre uno o varios (separados por comas) usuario y/o roles. Si se asigna a PUBLIC será accesible en toda la base de datos.

Si se incluye la cláusula WITH ADMIN OPTION, este permiso podrá ser concedido por el usuario al que se le ha asignado.

Ejemplos:

```
GRANT ALL ON FACTURA TO CONTROL_TOTAL;

GRANT SELECT, UPDATE ON ALUMNO TO PEPOTE, JUANCITO
WITH ADMIN OPTION;

GRANT SELECT ON PROFESOR TO PUBLIC;

GRANT SELECT ON APUNTE TO ACCESO_CONTABILIDAD;
```

Eliminación de usuarios

La eliminación de usuarios se hace a través de la instrucción DROP USER.
Su sintaxis es:

```
DROP USER usuario {CASCADE};
```

La cláusula CASCADE permite borrar el usuario y todos los objetos que posea.

Programación PL/SQL

PL: El lenguaje de programación para SQL

Ya dijimos en los primeros capítulos que SQL es un lenguaje de comandos, no un lenguaje de programación con todas las estructuras de control típicas. Así, SQL sólo contempla instrucciones, más o menos simples, pero no tiene ningún tipo de instrucciones de control de flujo o de otro tipo más propias de los lenguajes de programación 3GL.
Para subsanar esta carencia, Oracle definió un lenguaje de programación de tercera generación, que admitía sentencias SQL embebidas. Este lenguaje se llama PL/SQL (Programming Language/SQL)

La idea básica sobre la que se sustenta el PL/SQL es aplicar las estructuras típicas de un lenguaje de programación (bifurcaciones, bucles, funciones…) a las sentencias SQL típicas.
Así podemos tener el siguiente pseudocódigo:

```
Sentencia SELECT que recupera el total de sueldos

Si el total de sueldos > 1.000.000
   Sentencia UPDATE que incrementa un 10% los sueldos
Si no
   Sentencias UPDATE que incrementa un 5% los sueldos
Fin-si
```

Estructura básica en PL/SQL: El bloque de código

Cuando se escribe código en PL/SQL, este debe estar agrupado en unidades denominadas “bloques de código”. Un bloque de código puede contener otros sub-bloques de código y así sucesivamente.

Un bloque de código queda delimitado por las palabras reservadas BEGIN y END.
Por ejemplo:

```
BEGIN
      Sentencias . . .
      Sentencias . . .
      Sentencias . . .
```

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

            Sentencias . . .
            Sentencias . . .
            Sentencias . . .
      END;
```

En este ejemplo podemos ver que hay un bloque de código externo que contiene un bloque de código interno.

Un bloque de código opcionalmente puede contar con las siguientes secciones:

```
DECLARE
      Declaración de variables
BEGIN
      Sentencias SQL y PL/SQL
EXCEPTION
      Manejadores de excepciones
END;
```

La única sección obligatoria es la contenida dentro de BEGIN y END;

Comentarios

Los comentarios pueden ser multilínea encerrados entre /* y */ o monolínea, que comienzan por –

Declaración de variables

Las variables deben declararse dentro de la sección DECLARE y deben seguir la siguiente sintaxis:

```
      Nombre_de_variable   {CONSTANT}   TIPO {:= inicialización};
```

Los tipos posibles son todos aquellos válidos para SQL añadiendo algunos propios de PL/SQL. Para más información sobre los tipos propios de PL/SQL consultar el *PL/SQL User's Guide and Reference*

Ejemplos:

```
Interes       NUMBER(5,3);
Descripcion   VARCHAR2(50) := 'inicial';
Fecha_max     DATE;
Contabilizado BOOLEAN := TRUE;
PI            CONSTANT   REAL := 3.14159
```

Estructuras básicas de control

Como PL/SQL es un lenguaje 3GL, cuenta con las estructuras típicas de control de flujo: bifurcaciones condicionales y bucles:

Bifurcaciones condicionales

La sintaxis básica es:

```
IF condición_1 THEN
   Se ejecuta si se cumple condicion_1
ELSIF condicion_2 THEN -- ojo a 'ELSIF' y no ELSEIF
   Se ejecuta si no se cumple condicion_1 y se cumple condicion_2
ELSE
   Se ejecuta si no se cumple condicion_1 ni condicion_2
END IF;
```

Como en cualquier lenguaje de programación, las estructuras IF se pueden anidar unas dentro de otras.

Bucles

Existen varias variantes de la estructura bucle.
La más sencilla es la siguiente:

```
LOOP
   sentencias
END LOOP;
```

Las sentencias de dentro del bucle se ejecutarán durante un número indefinido de vueltas, hasta que aparezca la instrucción EXIT; que finalizará el bucle. Este tipo de bucle se denomina bucle incondicional.

Otra opción es incluir la estructura EXIT WHEN condición, se terminará el bucle cuando la condición se cumpla:

```
LOOP
   Sentencias
   EXIT WHEN condicion;
   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 condicion LOOP
   Sentencias
END LOOP;
```

Y por último el bucle FOR:

```
FOR contador IN {REVERSE} limite_inferior..limite_superior LOOP
    sentencias
END LOOP;
```

Contador deberá ser una variable de tipo numérico que sea capaz de contener los valores comprendidos entre limite_inferior y limite_superior.
Limite_inferior y limite_superior deberán ser expresiones numéricas, ya sean constantes (1,10…) o funciones (ROUND(max,0), ASCII('A')…)

Si la variable contador no está definida, PL/SQL definirá una variable de tipo INTEGER al iniciar el bucle, y la liberará al finalizar el bucle.

Registros y tablas

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 en un 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 nombre_registro IS RECORD(
   Campo1  tipo,
   Campo2  tipo,
   Campo3  tipo );
```

Por ejemplo:

```
TYPE alumno IS RECORD(
    n_alumno    VARCHAR2(5),
    nombre      VARCHAR2(25),
    apellido_1  VARCHAR2(25),
    apellido_2  VARCHAR2(25),
    tlf         VARCHAR2(15) );
```

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_tabla IS TABLE OF tipo_de_elementos;
```

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

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

Por ejemplo:

```
DECLARE
      TYPE array_enteros IS TABLE OF INTEGER;

      Un_array  array_enteros := array_enteros( 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 4 elementos (todos con 0).

Excepciones

Anteriormente dijimos que un bloque de código puede contener una sección denominada EXCEPTION. Esta sección es la encargada de recoger todas las anomalías que se puedan producir dentro del bloque de código.

Una excepción es una situación especial dentro de la ejecución de un programa, que puede ser capturada para asignar un nuevo comportamiento. Una excepción puede ser un error de ejecución (una división entre 0) o cualquier otro tipo de suceso.

Las excepciones deben ser declaradas dentro de la sección DECLARE, como si de una variable se tratasen:

```
DECLARE
    e_sin_alumnos  EXCEPTION;
```

Una vez que la excepción está definida, ésta debe ser lanzada, ya sea automáticamente por Oracle, o lanzada manualmente a través de la instrucción RAISE.

```
   SELECT COUNT(*)
   INTO num_alumnos;

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

Una vez que la excepción ha sido lanzada, la ejecución continua en la sección EXCEPTION, concretamente en el manejador apropiado (o el manejador WHEN OTHERS cuando no exista el específico).

Un manejador de excepciones es una sub-sección dentro de la sección EXCEPTION que se encarga de capturar una excepción concreta.

La sintaxis para escribir manejadores es:

```
EXCEPTION
     WHEN <excepción> THEN
          . . .
     WHEN <otra_excepción> THEN
         . . .
     WHEN OTHERS THEN
         . . .
END;
```

Las líneas de código debajo del manejador específico se ejecutarán cuando esa excepción se produzca.

Un ejemplo completo:

```
DECLARE
    e_sin_alumnos  EXCEPTION;
    num_alumnos    NUMBER(5);
BEGIN
   SELECT COUNT(*)
   INTO num_alumnos;

   IF num_alumnos = 0 THEN
      RAISE e_sin_alumnos;
   END IF;

EXCEPTION
   WHEN e_sin_alumno
      INSERT INTO ERROR( FECHA, DESCRIPCION )
      VALUES( SYSDATE, 'No se han encontrado alumnos en la tabla ALUMNO.');

   WHEN OTHERS
      Raise_application_error( -20000, 'Error en bloque de codigo PL/SQL' );
-- este error se transmite a la aplicación que llame a este bloque
-- de código (PL/SQL, Java, C++, etc.)
END;
```

Anteriormente habíamos dicho que las excepciones puede lanzarse automáticamente o manualmente a través de la instrucción RAISE.

Algunas excepciones se lanzarán automáticamente cuando se produzcan ciertos tipos de errores en la ejecución del bloque de código. Cada excepción automática tiene asociado un código de error ORA-XXXX el cual si se produce, hará que se lance la excepción correspondiente.

A continuación se muestra una lista de las excepciones automáticas predefinidas por Oracle:

Excepción	Error Oracle
ACCESS_INTO_NULL	ORA-06530
COLLECTION_IS_NULL	ORA-06531
CURSOR_ALREADY_OPEN	ORA-06511
DUP_VAL_ON_INDEX	ORA-00001
INVALID_CURSOR	ORA-01001
INVALID_NUMBER	ORA-01722
LOGIN_DENIED	ORA-01017
NO_DATA_FOUND	ORA-01403
NOT_LOGGED_ON	ORA-01012

Excepción	Error Oracle
PROGRAM_ERROR	ORA-06501
ROWTYPE_MISMATCH	ORA-06504
STORAGE_ERROR	ORA-06500
SUBSCRIPT_BEYOND_COUNT	ORA-06533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532
TIMEOUT_ON_RESOURCE	ORA-00051
TOO_MANY_ROWS	ORA-01422
VALUE_ERROR	ORA-06502
ZERO_DIVIDE	ORA-01476

Cursores

Cuando dentro de un intérprete SQL escribimos una consulta SELECT, el intérprete nos muestra las distintas filas del resultados para que podamos verlas. Sin embargo, dentro de un lenguaje de programación tenemos un problema, ya que lo más común no es mostrar el resultado, sino almacenarlo en variables para su posterior tratamiento.

Ahora tenemos que dividir el problema en dos partes, dependiendo del número de filas que nos retorna la consulta SELECT:

· Si retorna cero o una fila: El valor se podrá almacenar en tantas variables como columnas consultadas. Es decir, si escribimos un SELECT de tres columnas, y sólo retorna una fila (matriz 1x3), podremos almacenar el valor dentro de tres variables definidas para este uso. El modo de hacer esto en PL/SQL es:

```
SELECT col1, col2, col3
INTO   var1, var2, var3
FROM   TABLA;
```

De este modo se almacenará en las variables var1, var2 y var3 los valores recuperados por la consulta SELECT o NULL si esta consulta no retorna ninguna fila.

· Si retorna más de una fila: En este caso no es posible almacenar directamente los valores en variables. Para ello existen los cursores, que no son más que consultas SELECT que se recuperar fila a fila y no todo su conjunto de resultados a la vez.

Para utilizar 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.

Sin embargo para realizar una consulta a través de un cursor hay que realizar los siguientes pasos:

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

1.- Declarar el cursor

En este paso se define el nombre que tendrá el cursor y qué consulta SELECT ejecutará. No es más que una declaración. 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 en el servidor

Un cursor internamente es una sentencia SELECT cuyo resultado se guarda en el servidor en tablas temporales y que se va retornando cada una de las filas según se va pidiendo desde el cliente.
El primer paso es ejecutar la sentencia SELECT y guardar su resultado dentro de las tablas temporales. Este paso se denomina Abrir el cursor.
La apertura del cursor debe realizarse sólo una vez.
La sintaxis de apertura de un cursor es:

```
OPEN nombre_cursor;
```

Una vez que el cursor está abierto, se podrá empezar a pedir los resultados al servidor.

3.- Recuperar cada una de sus filas

Una vez que el cursor está abierto en el servidor se podrá hacer la petición de recuperación de fila. Este paso es equivalente a hacer una consulta SELECT de una sola fila, ya que estamos seguros de que no vamos a recuperar más de una fila.
La sintaxis de recuperación de fila de un cursor es:

```
FETCH nombre_cursor INTO variables;
```

Podremos recuperar filas mientras la consulta SELECT tenga filas pendientes de recuperar. Para saber cuándo no hay más filas podemos consultar los siguientes atributos de un cursor:

Nombre de atributo	Retorna	Descripción
Nombre_cursor%FOUND	BOOLEAN	Retorna si la última fila recuperada fue válida
Nombre_cursor%ISOPEN	BOOLEAN	Retorna si el cursor está abierto
Nombre_cursor%NOTFOUND	BOOLEAN	Retorna si la última fila fue inválida
Nombre_cursor%ROWCOUNT	NUMBER	Retorna el número de filas recuperadas

Así, la acción más típica es recuperar filas mientras queden alguna por recuperar en el servidor.

Esto lo podremos hacer a través del siguiente bloque de código:

```
LOOP
  FETCH nombre_cursor INTO variables;
  EXIT  WHEN nombre_cursor%NOTFOUND;

  <procesar cada una de las filas>
END LOOP;
```

4.- Cerrar el cursor

Una vez que se han recuperado todas las filas del cursor, hay que cerrarlo para que se liberen de la memoria del servidor los objetos temporales creados. Si no cerrásemos el cursor, la tabla temporal quedaría en el servidor almacenada con el nombre dado al cursor y la siguiente vez ejecutásemos ese bloque de código, nos daría la excepción CURSOR_ALREADY_OPEN (cursor ya abierto) cuando intentásemos abrir el cursor.

Para cerrar el cursor se utiliza la siguiente sintaxis:

```
CLOSE nombre_cursor;
```

Funciones, procedimientos y paquetes

Una vez que tenemos escrito un bloque de código, podemos guardarlo en un fichero .SQL para su posterior uso, o bien guardarlo en base de datos para que pueda ser ejecutado por cualquier aplicación.

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

1.- Palabra reservada DECLARE desaparece

2.- 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.

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

```
CREATE {OR REPLACE} PROCEDURE nombre(tipo_dato p1, tipo_dato p2...) IS
<Sección DECLARE>
BEGIN

{EXCEPTION}
END;
```

Para crear una función usaremos la siguiente sintaxis:

```
CREATE {OR REPLACE} FUNCTION nombre(tipo_dato p1, tipo dato p2...)
RETURN tipo_dato IS
<Sección DECLARE>
BEGIN

{EXCEPTION}
END;
```

Una vez que tenemos creados nuestros procedimientos y funciones, podemos almancenarlos dentro de librerías de funciones. Estas librerías tienen el nombre de paquetes o packages.

Un paquete puede contener procedimientos, funciones, 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 pasa por dos fases:

1.- Crear la cabecera del paquete donde se definen que procedimientos, funciones, variables, etc. Contendrá el paquete.

2.- Crear el cuerpo del paquete, donde se definen los bloques de código de las funciones y procedimientos definidos en la cabecera del paquete.

Para crear la cabecera del paquete utilizaremos la siguiente instrucción:

```
CREATE {OR REPLACE} PACKAGE nombre_de_paquete IS

<Declaraciones>

END;
```

Para crear el cuerpo del paquete utilizaremos la siguiente instrucción:

```
CREATE {OR REPLACE} PACKAGE BODY nombre_paquete IS

<Bloques de código>

END;
```

Hay que tener en cuenta que toda declaración de función o procedimiento debe estar dentro del cuerpo del paquete, y que todo bloque de código contenido dentro del cuerpo debe estar declarado dentro de la cabecera de paquete.

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

```
Nombre_paquete.función(x)
Nombre_paquete.procedimiento(x)
Nombre_paquete.variable
```

Oracle8 define los siguientes paquetes de funciones predefinidos:

DBMS_ALERT	DBMS_PIPE
DBMS_APPLICATION_INFO	DBMS_REFRESH
DBMS_AQ	DBMS_REPCAT
DBMS_AQADM	DBMS_REPCAT_ADMIN
DBMS_DDL	DBMS_REPCAT_AUTH
DBMS_DEFER	DBMS_ROWID
DBMS_DEFER_QUERY	DBMS_SESSION
DBMS_DEFER_SYS	DBMS_SHARED_POOL
DBMS_DESCRIBE	DBMS_SNAPSHOT
DBMS_JOB	DBMS_SQL
DBMS_LOB	DBMS_UTILITY
DBMS_LOCK	UTL_FILE
DBMS_OUTPUT	

Para más información sobre PL/SQL consultar el *Oracle8 PL/SQL User's Guide and Reference*

Disparadores

Los disparadores (o *triggers*) son bloques de código almacenados en base de datos y que se ejecutan automáticamente. Un disparador está asociado a una tabla y a una operación DML específica (INSERT, UPDATE o DELETE).
En definitiva, los disparadores son eventos a nivel de tabla que se ejecutan automáticamente cuando se realizan ciertas operaciones sobre la tabla.

Para crear un disparador utilizaremos la siguiente instrucción:

```
CREATE {OR REPLACE} TRIGGER nombre_disp
[BEFORE|AFTER|INSTEAD OF] [DELETE|INSERT|UPDATE {OF columnas}] ON tabla
{DECLARE}
BEGIN

{EXCEPTION}
END;
```

Para más información sobre los disparadores consultar el *Oracle8 SQL Reference*

El catálogo de Oracle

Oracle cuenta con una serie de tabla y vistas que conforman una estructura denominada catálogo. La principal función del catálogo de Oracle es almacenar toda la información de la estructura lógica y física de la base de datos, desde los objetos existentes, la situación de los *datafiles*, la configuración de los usuarios, etc.

El catálogo sigue un estándar de nomenclatura para que su memorización sea más fácil:

Prefijos:

Prefijo	**Descripción**
DBA_	Objetos con información de administrador. Sólo accesibles por usuarios con permisos DBA
USER_	Objetos con información propia del usuario al que estamos conectado. Accesible desde todos los usuarios. Proporcionan menos información que los objetos DBA_
ALL_	Objetos con información de todos los objetos en base de datos.
V_$ ó V$	Vistas dinámicas sobre datos de rendimiento

Existe una tabla de catálogo para cada tipo de objeto posible. Su nombre aparecerá en plural

TABLES, VIEWS, SEQUENCES, TABLESPACES…

Sabiendo qué objetos existen, y qué prefijos podemos utilizar, ya podemos acceder a los objetos del catálogo de Oracle.

Ejemplos:

Objeto	**Descripción**
DBA_TABLES	Información para administradores de las tablas en base de datos.
USER_VIEWS	Información de las vistas creadas por el usuario desde el que accedemos.
ALL_SEQUENCES	Información de todas las secuencias existentes en base de datos.
DBA_TABLESPACES	Información de administración sobre los *tablespaces*.
USER_TAB_COLUMNS	Todas las columnas de tabla en el usuario activo.

Los objetos de catálogo también guardan relaciones entre ellos. Por ejemplo, el objeto ALL_TABLES guarda una relación 1-N con el objeto ALL_TAB_COLUMNS: Una tabla tiene N columnas.

Existe un pseudo-usuario llamado PUBLIC el cual tiene acceso a todas las tablas del catálogo público. Si se quiere que todos los usuarios tengan algún tipo de acceso a un objeto, debe darse ese privilegio a PUBLIC y todo el mundo dispondrá de los permisos correspondientes.

El catálogo público son aquellas tablas (USER_ y ALL_) que son accesibles por todos los usuarios. Normalmente dan información sobre los objetos creados en la base de datos.

El catálogo de sistema (DBA_ y V_$) es accesible sólo desde usuarios DBA y contiene tanto información de objetos en base de datos, como información específica de la base de datos en sí (versión, parámetros, procesos ejecutándose…)

Ciertos datos del catálogo de Oracle están continuamente actualizados, como por ejemplo las columnas de una tabla o las vistas dinámicas (V$). De hecho, en las vistas dinámicas, sus datos no se almacenan en disco, sino que son tablas sobre datos contenidos en la memoria del servidor, por lo que almacenan datos actualizados en tiempo real. Algunas de las principales son:

- V$DB_OBJECT_CACHE: contiene información sobre los objetos que están en el caché del SGA
- V$FILESTAT: contiene el total de lecturas y escrituras físicas sobre un *data file* de la base de datos.
- V$ROLLSTAT: contienen información acerca de los segmentos de *rollback*.

Sin embargo hay otros datos que no pueden actualizarse en tiempo real porque penalizarían mucho el rendimiento general de la base de datos, como por ejemplo el número de registros de una tabla, el tamaño de los objetos, etc.
Para actualizar el catálogo de este tipo de datos es necesario ejecutar una sentencia especial que se encarga de volcar la información recopilada al catálogo:

```
ANALYZE [TABLE|INDEX] nombre
[COMPUTE|ESTIMATE|DELETE] STATISTICS;
```

La cláusula COMPUTE hace un cálculo exacto de la estadísticas (tarda más en realizarse en ANALYZE), la cláusula ESTIMATE hace una estimación partiendo del anterior valor calculado y de un posible factor de variación y la cláusula DELETE borra las anteriores estadísticas.

La sentencia COMMENT

El catálogo público contiene ciertas tablas encargadas de almacenar información adicional sobre tablas, vistas y columnas. La información que se suele almacenar es información de análisis, valores posibles para las columnas y en general todo aquello que se haya concluido durante la etapa de análisis.

Las tablas existentes son:

Tabla	Descripción
ALL_TAB_COMMENTS	Contiene los comentarios para tablas y vistas.
ALL_COL_COMMENTS	Contiene los comentarios para las columnas de tablas y vistas.

La información se debe almacenar en base de datos según la siguiente sintaxis:

```
COMMENT ON TABLE [tabla|vista] IS 'texto';

COMMENT ON COLUMN [tabla|vista].columna IS 'texto';
```

Una vez que esta información está en base de datos, se puede escribir procedimientos o scripts SQL que muestren la información para sacar informes de documentación de base de datos.

Optimización básica de SQL

Una de las tareas más importantes de las propias de un desarrollador de bases de datos es la de optimización, ajuste, puesta a punto o *tunning*. Hay que tener en cuenta que las sentencias SQL pueden llegar a ser muy complejas y conforme el esquema de base de datos va creciendo, las sentencias son más complejas y confusas. Por es difícil escribir la sentencia correcta a la primera.
Por todo ello después de tener cada uno de los procesos escrito, hay que pasar por una etapa de *tunning* en la que se revisan todas las sentencias SQL para poder optimizarlas conforme a la experiencia adquirida.

Tanto por cantidad como por complejidad, la mayoría de las optimizaciones deben hacerse sobre sentencias SELECT, ya que son (por regla general) las responsables de la mayor pérdida de tiempos.

Normas básicas de optimización

A continuación se dan unas normas básicas para escribir sentencias SELECT optimizadas.

- Las condiciones (tanto de filtro como de *join*) deben ir siempre en el orden en que esté definido el índice. Si no hubiese índice por las columnas utilizadas, se puede estudiar la posibilidad de añadirlo, ya que tener índices extra sólo penaliza los tiempos de inserción, actualización y borrado, pero no de consulta.

- Al crear un restricción de tipo PRIMARY KEY o UNIQUE, se crea automáticamente un índice sobre esa columna.

- Para chequeos, siempre es mejor crear restricciones (*constraints)* que disparadores (*triggers)*.

- Hay que optimizar dos tipos de instrucciones: las que consumen mucho tiempo en ejecutarse, o aquellas que no consumen mucho tiempo, pero que son ejecutadas muchas veces.

- Generar un plan para todas las consultas de la aplicación, poniendo especial cuidado en los planes de las vistas, ya que estos serán incluidos en todas las consultas que hagan referencia a la vista.

- Generar y optimizar al máximo el plan de las vistas. Esto es importante porque el SQL de una vista, no se ejecuta mientras que la vista no es utilizada en una consulta, así que

todas las consultas de esa vista se ven afectadas por su plan. Hay que tener especial cuidado de hacer *joins* entre vistas.

- Si una aplicación que funcionaba rápido, se vuelve lenta, hay que parar y analizar los factores que han podido cambiar. Si el rendimiento se degrada con el tiempo, es posible que sea un problema de volumen de datos, y sean necesarios nuevos índices para acelerar las búsquedas. En otras ocasiones, añadir un índice equivocado puede ralentizar ciertas búsquedas. Cuantos más índices tenga una tabla, más se tardará en realizar inserciones y actualizaciones sobre la tabla, aunque más rápidas serán las consultas. Hay que buscar un equilibrio entre el número de índices y su efectividad, de tal modo que creemos el menos número posible, pero sean utilizados el mayor número de veces posible.

- Utilizar siempre que sea posible las mismas consultas. La segunda vez que se ejecuta una consulta, se ahorrará mucho tiempo de *parsing* y optimización, así que se debe intentar utilizar las mismas consultas repetidas veces.

- Las consultas más utilizadas deben encapsularse en procedimientos almacenados. Esto es debido a que el procedimiento almacenado se compila y analiza una sola vez, mientras que una consulta (o bloque PL/SQL) lanzado a la base de datos debe ser analizado, optimizado y compilado cada vez que se lanza.

- Los filtros de las consultas deben ser lo más específicos y concretos posibles. Es decir: es mucho más específico poner WHERE campo = 'a' que WHERE campo LIKE '%a%'. Es muy recomendable utilizar siempre consultas que filtren por la clave primaria u otros campos indexados.

- Hay que tener cuidado con lanzar demasiadas consultas de forma repetida, como por ejemplo dentro de un bucle, cambiando una de las condiciones de filtrado. Siempre que sea posible, se debe consultar a la base de datos una sola vez, almacenar los resultados en la memoria del cliente, y procesar estos resultados después. También se pueden evitar estas situaciones con procedimientos almacenados, o con consultas con parámetros acoplados (*bind*).

- Evitar la condiciones IN (SELECT…) sustituyéndolas por joins: cuando se utiliza un conjunto de valores en la clausula IN, se traduce por una condición compuesta con el operador OR. Esto es lento, ya que por cada fila debe comprobar cada una de las condiciones simples. Suele ser mucho más rápido mantener una tabla con los valores que están dentro del IN, y hacer un join normal. Por ejemplo, esta consulta:

```
SELECT *
FROM datos
WHERE campo IN ('a', 'b', 'c', 'd', ... , 'x', 'y', 'z');
```

se puede sustituir por la siguiente consulta, siempre que la tabla "letras" contenga una fila por cada valor contenido en el conjunto del IN:

```
SELECT *
FROM datos d, letras l
WHERE d.campo = l.letra;
```

También hay que tener cuidado cuando se mete un SELECT dentro del IN, ya que esa consulta puede retornar muchas filas, y se estaría cayendo en el mismo error. Normalmente, una condición del tipo "WHERE campo IN (SELECT...)" se puede sustituir por una consulta con *join*.

- Cuando se hace una consulta multi-tabla con *joins*, el orden en que se ponen las tablas en el FROM influye en el plan de ejecución. Aquellas tablas que retornan más filas deben ir en las primeras posiciones, mientras que las tablas con pocas filas deben situarse al final de la lista de tablas.

- Si en la cláusula WHERE se utilizan campos indexados como argumentos de funciones, el índice quedará desactivado. Es decir, si tenemos un índice por un campos IMPORTE, y utilizamos una condición como WHERE ROUND(IMPORTE) > 0, entonces el índice quedará desactivado y no se utilizará para la consulta.

- Siempre que sea posible se deben evitar las funciones de conversión de tipos de datos e intentar hacer siempre comparaciones con campos del mismo tipo. Si hay que hacer algún tipo de conversión, intenta evitar el uso del *cast* y aplica siempre la función de conversión sobre la constante, y no sobre la columna.

- Una condición negada con el operador NOT desactiva los índices

- Una consulta cualificada con la cláusula DISTINCT debe ser ordenada por el servidor aunque no se incluya la cláusula ORDER BY.

- Para comprobar si existen registros para cierta condición, no se debe hacer un SELECT COUNT(*) FROM X WHERE xxx, sino que se hace un SELECT DISTINCT 1 FROM X WHERE xxx. De este modo evitamos al servidor que cuente los registros.

- Si vamos a realizar una operación de inserción, borrado o actualización masiva, es conveniente desactivar los índices, ya que por cada operación individual se actualizarán

los datos de cada uno de los índices. Una vez terminada la operación, volvemos a activar los índices para que se regeneren.

- La mejor optimización es rediseñar y normalizar la base de datos. Las bases de datos relacionales están diseñadas para funcionar lo más rápidamente posible para un buen diseño relacional, pero con diseños erróneos, se vuelven muy lentas. La mayoría de los problemas de rendimiento tienen un problema de fondo de mal diseño, y muchos de ellos no podrán ser optimizados si no se rediseña el esquema de base de datos.

Toda consulta SELECT se ejecuta dentro del servidor en varios pasos. Para la misma consulta, pueden existir distintas formas para conseguir el mismo resultados, por lo que el servidor es el responsable de decidir qué camino seguir para conseguir el mejor tiempo de respuesta.
La parte de la base de datos que se encarga de estas decisiones se llama Optimizador.
El camino seguido por el servidor para la ejecución de una consulta se denomina "Plan de ejecución"
En *Oracle8* existen dos optimizadores para la decisión del plan de ejecución:

Optimizador basado en reglas (RULE)

Se basa en ciertas reglas para realizar las consultas. Por ejemplo, si se filtra por un campo indexado, se utilizará el índice, si la consulta contiene un ORDER BY, la ordenación se hará al final, etc. No tiene en cuenta el estado actual de la base de datos, ni el número de usuarios conectados, ni la carga de datos de los objetos, etc. Es un sistema de optimización estático, no varía de un momento a otro.

Optimizador basado en costes (CHOOSE)

Se basa en las reglas básicas, pero teniendo en cuenta el estado actual de la base de datos: cantidad de memoria disponible, entradas/saludas, estado de la red, etc. Por ejemplo, si se hace una consulta utilizando un campo indexado, mirará primero el número de registros y si es suficientemente grande, entonces merecerá la pena acceder por el índice, si no, accederá directamente a la tabla.

Para averiguar el estado actual de la base de datos se basa en los datos del catálogo público, por lo que es recomendable que esté lo más actualizado posible (a través de la sentencia ANALYZE), ya que de no ser así, se pueden tomar decisiones a partir de datos desfasados (la tabla tenía 10 registros hace un mes pero ahora tiene 10.000).

Oracle8 recomienda que todas las consultas se hagan por costes, aunque hay ciertos casos en los que una consulta no se resuelve (o tarda mucho) por costes y por reglas es inmediata.

¿Y cómo hacer para que una consulta se ejecute por reglas o por costes? Pues hay dos modos de forzar a Oracle a que utilice un optimizador concreto.
La primera es modificando la sesión activa para que todas las consultas sean optimizadas de una manera:

```
ALTER SESSION SET OPTIMIZER_GOAL = [RULE|CHOOSE];
```

Con esto todas las consultas se ejecutarán utilizando el optimizador indicado.
La otra manera es forzando a Oracle a que utilice un optimizador en una consulta concreta. Esto se hace a través de los "hints" o sugerencias.

Sugerencias o *hints*

Un *hint* es un comentario dentro de una consulta SELECT que informa a Oracle del modo en que tiene que trazar el plan de ejecución.
Los *hint* deben ir junto detrás de la palabra SELECT:

```
SELECT /*+ HINT */ . . .
```

A continuación se muestra una lista de algunos de los *hints* posibles:

Hint	Descripción
`/*+ CHOOSE */`	Pone la consulta a costes.
`/*+ RULE */`	Pone la consulta a reglas.
`/*+ ALL_ROWS */`	Pone la consulta a costes y la optimiza para que devuelva todas las filas en el menor tiempo posible. Es la opción por defecto del optimizador basado en costes. Esto es apropiado para procesos en masa, en los que son necesarias todas las filas para empezar a trabajar con ellas.
`/*+ FIRST_ROWS */`	Pone la consulta a costes y la optimiza para conseguir que devuelva la primera fila en el menor tiempo posible. Esto es idóneo para procesos online, en los que podemos ir trabajando con las primeras filas mientras se recupera el resto de resultados. Este *hint* se desactivará si se utilizan funciones de grupo como SUM, AVG, etc.
`/*+ INDEX( tabla o índice ) */`	Fuerza la utilización del índice indicado para la tabla indicada. Se puede indicar el nombre de un índice (se utilizará ese índice), de varios índices (el optimizador elegirá uno entre todos ellos) o de una tabla (se utilizará cualquier índice de la tabla).
`/*+ ORDERED */`	Hace que las combinaciones de las tablas se hagan en el mismo orden en que aparecen en el join.

Para más información sobre los *hints* consultar el *Oracle 8 Tunning*.

Una misma consulta puede generar distintos planes de ejecución en las siguientes situaciones:

- Cambios en las estadísticas de las tablas (COMPUTE STATISTICS) si se utiliza un optimizador basado en costes.
- Uso de los *hints* en si se utiliza el optimizador basado en reglas.
- Añadir o eliminar índices de una tabla, si se utiliza el optimizador basado en reglas.

Calcular el coste de una consulta

Para calcular el coste de una consulta, el optimizador se basa en las estadísticas almacenadas en el catálogo de Oracle, a través de la instrucción:

```
ANALYZE [TABLE,INDEX] <object_name> [COMPUTE, ESTIMATE] STATISTICS;
```

Si no existen datos estadísticos para un objeto (por ejemplo, porque se acaba de crear), se utilizarán valores por defecto. Además, si los datos estadísticos está anticuados, se corre el riesgo de calcular costes basados en estadísticas incorrectas, pudiendo ejecutarse planes de ejecución que a priori pueden parecer mejores.
Por esto, si se utiliza el optimizador basado en costes, es muy importante analizar los objetos periódicamente (como parte del mantenimiento de la base de datos). Como las estadísticas van evolucionando en el tiempo (ya que los objetos crecen o decrecen), el plan de ejecución se va modificando para optimizarlo mejor a la situación actual de la base de datos. El optimizador basado en reglas hacía lo contrario: ejecutar siempre el mismo plan, independientemente del tamaño de los objetos involucrados en la consulta.
Dentro de la optimización por costes, existen dos modos de optimización, configurables desde el parámetro OPTIMIZER_MODE:

- FIRST_ROWS: utiliza sólo un número determinado de filas para calcular los planes de ejecución. Este método es más rápido pero puede dar resultados imprecisos.

- ALL_ROWS: utiliza todas las filas de la tabla a la hora de calcular los posibles planes de ejecución. Este método es más lento, pero asegura un plan de ejecución muy preciso. Si no se indica lo contrario, este es el método por defecto.

Plan de ejecución

Aunque en la mayoría de los casos no hace falta saber cómo ejecuta Oracle las consultas, existe una sentencia especial que nos permite ver esta información.
El plan de ejecución nos proporciona muchos datos que pueden ser útiles para averiguar qué está ocurriendo al ejecutar una consulta, pero principalmente, de lo que nos informa es del tipo de optimizador utilizado, y el orden y modo de unir las distintas tablas si la instrucción utiliza algún *join*.

Para obtener un plan de ejecución, debemos rellenar una tabla especial (llamada PLAN_TABLE) con un registro para cada paso en el plan de ejecución.
En Oracle8, la tabla PLAN_TABLE debe tener la siguiente estructura, aunque cambia en cada versión. Puedes encontrar la definición de la tabla en el *script* UTLXPLAN.SQL, dentro del directorio ORACLE_HOME/rdbms/admin

```
CREATE TABLE PLAN_TABLE(
       STATEMENT_ID        VARCHAR2(30),
       TIMESTAMP           DATE,
       REMARKS             VARCHAR2(80),
       OPERATION           VARCHAR2(30),
       OPTIONS             VARCHAR2(30),
       OBJECT_NODE         VARCHAR2(128),
       OBJECT_OWNER        VARCHAR2(30),
       OBJECT_NAME         VARCHAR2(30),
       OBJECT_INSTANCE     NUMERIC,
       OBJECT_TYPE         VARCHAR2(30),
       OPTIMIZER           VARCHAR2(255),
       SEARCH_COLUMNS      NUMERIC,
       ID                  NUMERIC,
       PARENT_ID           NUMERIC,
       POSITION            NUMERIC,
       COST                NUMERIC,
       CARDINALITY         NUMERIC,
       BYTES               NUMERIC,
       OTHER_TAG           VARCHAR2(255),
       OTHER               LONG);
```

El significado de estos campos es el siguiente:

Campo	Descripción
STATEMENT_ID	El identificador de sentencia utilizado para el plan de ejecución. Este identificador será un nombre que daremos nosotros mismo al plan.
TIMESTAMP	Fecha y hora del momento en el que se generó el plan de ejecución.
REMARKS	Una columna "comentario" que se puede utilizar para añadir comentarios a los registros de PLAN_TABLE, utilizando la instrucción UPDATE como con cualquier otra tabla.
OPERATION	La operación SQL realizada en la etapa.
OPTIONS	El modo utilizado para la operación, como puede ser UNIQUE SCAN, SORT JOIN, etc.
OBJECT_NODE	EL database link usado para acceder al objeto.
OBJECT_OWNER	El propietario del objeto referenciado en la operación.
OBJECT_NAME	El nombre del objeto referenciado en la operación.
OBJECT_INSTANCE	La posición ordinal del objeto en el SQL analizado.
OBJECT_TYPE	Un atributo del objeto, como UNIQUE para los índices.
OPTIMIZER	El modo en que se ha utilizado el optimizador: CHOOSE ó RULE.
SEARCH_COLUMNS	No se utiliza.
ID	Un número único asignado a cada etapa en el plan.
PARENT_ID	El ID de la etapa que es el "padre" de la etapa actual en la jerarquía del plan de ejecución. Una etapa de ejecución se puede resolver en sub-etapas, formando así una jerarquía.
POSITION	El orden de proceso para etapas con el mismo Parent_ID.
COST	Coste relativo de la etapa.
CARDINALITY	Cardinalidad de un índice o el número de filas esperado que devuelva la operación.
BYTES	El tamaño (en bytes) de cada fila devuelta.
OTHER_TAG	Si el valor es SERIAL_FROM_REMOTE, el SQL en la columna Other se ejecutará en el nodo remoto. Otros valores describen el uso de la operación en Parallel Query Option
OTHER	Para consultas distribuidas, Other contiene el texto del SQL que es ejecutado en el nodo remoto.

Una vez que la tabla está creada en el usuario donde vamos a ejecutar la consulta, de debe ejecutar la siguiente sentencia:

```
EXPLAIN PLAN
SET STATEMENT_ID = 'identificador de sentencia'
FOR <consulta SELECT a evaluar>;
```

El identificador de sentencia tiene que ser una cadena descriptiva para nuestra sentencia. Se utilizará más tarde para recuperar el plan entre todos los almacenados dentro de la consulta SELECT.

Esta sentencia lo que hará es almacenar en la tabla PLAN_TABLE un registro por cada paso en el plan de ejecución. El campos STATEMENT_ID de los pasos de nuestro plan de ejecución estará al valor indicado en 'identificador de sentencia'.

Para mostrar el plan de ejecución se debe hacer un SELECT filtrando aquellos registros de nuestro plan de ejecución.

Una sentencia típica que nos muestra el plan de ejecución formateado podría ser:

```
SELECT id, parent_id,
       LPAD(' ', 2*(level-1)) ||operation|| ' ' ||options|| ' ' ||object_name||
       ' ' || DECODE(id, 0, 'Cost = '||position ) "Plan de consulta"
FROM PLAN_TABLE
START WITH id = 0 and statement_id = 'identificador de sentencia'
CONNECT BY prior id = parent_id and statement_id = 'identificador de sentencia';
```

Con esta instrucción, obtendremos un plan parecido a este:

```
QUERY PLAN
------------------------------------------------------------------------------
  SORT                              ORDER BY
    NESTED LOOPS
      FILTER
        NESTED LOOPS                      OUTER
          TABLE ACCESS                      FULL                    HEADER
          TABLE ACCESS                      BY ROWID                DETAIL
            INDEX                             RANGE SCAN            DETAIL_PK
          TABLE ACCESS                      FULL                    HEADER
          TABLE ACCESS                      BY ROWID                DETAIL
            INDEX                             RANGE SCAN            DETAIL_PK
        NESTED LOOPS                      OUTER
          TABLE ACCESS                      FULL                    HEADER
          TABLE ACCESS                      BY ROWID                DETAIL
            INDEX                             RANGE SCAN            DETAIL_PK
          TABLE ACCESS                      FULL                    HEADER
          TABLE ACCESS                      BY ROWID                DETAIL
            INDEX                             RANGE SCAN            DETAIL_PK
      TABLE ACCESS                    BY ROWID                      DETAIL
        INDEX                           RANGE SCAN                  DETAIL_PK
```

Interpretando el plan de ejecución

Una de las tareas que más confusión crea es la interpretación y lectura de un plan de ejecución. Generar este plan es relativamente fácil, y basta con seguir ciertos pasos para obtenerlo. Sin embargo, leer e interpretar correctamente un plan de ejecución es una tarea compleja, en la que la experiencia es lo más importante.
De todas formas, todo se puede aprender, y se pueden dar algunas indicaciones para ir introduciéndonos en este mundo.

La lectura del plan de ejecución se hace de arriba a abajo, y de izquierda a derecha. Como unas instrucciones están anidadas dentro de otras, esto significa que una instrucción se resuelve en instrucciones más pequeñas, que son las que están dentro de la más externa.
A continuación se representa un plan de ejecución de ejemplo, en el que para resolver una operación A, es necesario resolver las instrucciones más sencillas "a1", "a2" y "a3", y si para resolver "a2" es necesario ejecutar "a21" y "a22":

```
A
      a1
      a2
            a22
            a23
      a3
```

En plan se organiza en tres columnas, de mayor a menor generalidad. Estas columnas que indican:

- La operación a realizar
- Las opciones (o modo) que se aplican a la operación
- El objeto sobre el que se realiza la operación

Por ejemplo, la siguiente puede ser una línea de un plan de ejecución:

TABLE ACCESS BY ROWID DETAIL

Se muestran estas tres columnas, que significan:

Operación: acceso a tabla (TABLE ACCESS)
Modo: se accede a la tabla utilizando el ROWID (BY ROWID)
Objeto: Se accede a la tabla "DETAIL" (DETAIL)

Las operaciones y modos más típicos que podemos encontrarnos son:

Operación y modo	Descripción
FILTER	Se descartan aquellas filas de la tabla que no cumplen con una condición impuesta en la cláusula WHERE.
INDEX UNIQUE	Se accede a la información localizada a través de una clave primaria o un índice único. Es uno de los modos más rápidos de acceder a una fila (además del acceso por ROWID).
INDEX RANGE SCAN	Se accede a la información localizada a través de un índice con repeticiones. Es típico que aparezca esta operación cuando utilizamos los operadores BETWEEN, <, >, <=, >= sobre columnas indexadas con repeticiones.
MERGE JOIN	Se combinan dos tablas ordenadas para dar como resultado una tabla ordenada y sin repeticiones.
HASH JOIN	Se combinan dos tablas utilizando un algoritmo de *hash*.
NESTED LOOP JOIN	Se combinan dos tablas utilizando bucles anidados. Por cada fila de la tabla padre, se recorren todas las filas de la tabla hija.
SORT ORDER BY	Se ordenan los datos de una tabla según lo indicado en la cláusula ORDER BY.
SORT GROUP BY	Se ordenan los datos de una tabla según lo indicado en la cláusula GROUP BY.
SORT JOIN	Se ordenan los datos de una tabla. El modo (JOIN) indica que la ordenación se hace para preparar los datos para un *join* de tipo MERGE JOIN.
SORT UNIQUE	Se ordenan los datos de una tabla. El modo (UNIQUE) indica que la ordenación se hace para eliminar duplicados, bien por un DISTINTC o el uso del operador UNION.
TABLE ACCESS FULL	Se hace una lectura secuencial (desde el primer registro hasta el último) de los datos de la tabla. Este es el modo más lento de acceder a los datos de una tabla.
TABLE ACCESS BY ROWID	Se hace una lectura de un registro concreto accediendo por su ROWID o por un índice único. Es el modo más rápido de acceder a un registro.

Trazas de ejecución

Las trazas de ejecución son una posibilidad que incluye Oracle para mostrar todas las sentencias y su plan de ejecución de programas que acceden a bases de datos Oracle. Es decir, no es necesario disponer del código fuente, ni de la sentencia SQL para saber qué y cómo se ha ejecutado.
Además de las instrucciones y su plan de ejecución, el archivo de traza nos proporciona información sobre las sentencias erróneas, los tiempos de ejecución, el optimizador utilizado, etc.

Básicamente, activar la traza de ejecución consiste en ejecutar un procedimiento que tiene predefinido Oracle dentro del paquete DBMS_SYSTEM.
El procedimiento tiene la siguiente cabecera:

```
SYS.DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION( sid, serial#, TRUE/FALSE );
```

El parámetro sid (session identifier) indica un número único para cada sesión establecida con Oracle.
El parámetro serial# indica un número de serie para la sesión indicada.
Ambos parámetros se podrán consultar en la tabla del catálogo SYS.V_$SESSION con la siguiente consulta:

```
SELECT SID, SERIAL#, MACHINE, TERMINAL, PROGRAM
FROM SYS.V_$SESSION
WHERE USERNAME = 'usuario_conectado';
```

Esta consulta nos mostrará los valores que debemos indicar en SID y SERIAL# para la sesión a la cual queremos hacer la traza.
El tercer parámetro indica si queremos activar/desactivar la traza de ejecución.

Suponiendo que la anterior consulta ha retornado un SID de 9 y un SERIAL# de 30, activaremos la traza utilizando la siguiente llamada desde SQL*Plus:

```
BEGIN
      SYS.DBMS_SYSTEM.set_sql_trace_in_session(9, 30 TRUE);
END;
```

Una vez que hemos ejecutado el procedimiento set_sql_trace_in_session, toda sentencia ejecutada sobre la sesión indicada quedará registrada en un archivo ORAxxxxx.TRC, normalmente bajo el directorio ORACLE_HOME/trace80, aunque esto depende de la opción USER_DUMP_DEST del archivo INIT.ORA.

Este fichero de traza contiene información detallada, pero ilegible, de los pasos seguidos por el plan de ejecución.
Para conseguir una salida legible de esta traza se debe ejecutar la utilidad TKPROF, que podremos encontrar en el directorio BIN de Oracle. Una llamada típica podría ser la siguiente:

```
TKPROF <fichero_traza> <fichero_salida> [explain=usuario/password] [sys=no]
```

Con el parámetro "explain=usuario/password" indicamos que nos muestre el plan de ejecución de todas las instrucciones, conectándose para ello al usuario/password indicados.
Con el parámetro "sys=no" indicamos que no nos muestre las instrucciones realizadas por el usuario SYS.

Existe otro modo más sencillo de obtener la traza de una sola instrucción, desde SQL*Plus con el auto-trace.
Otro modo de activar el auto-trace es a través de la instrucción

```
Set auto trace ON/OFF
```

Una vez activada la auto-traza, toda sentencia ejecutada en SQL*Plus vendrá acompañada de su plan y estadísticas de ejecución.
Además, con la instrucción

```
set timing ON/OFF
```

Se activará el reloj interno de Oracle con el que se podrá cronometrar el tiempo de ejecución de cada consulta.