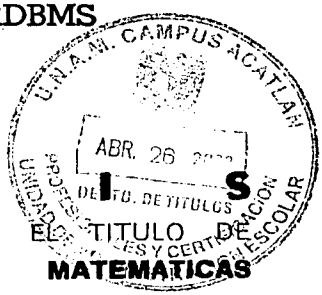
2

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES

"ACATLAN"

OPTIMIZACION DE APLICACIONES EN UN RDBMS



T E S I S

QUE PARA OBTENER EL TITULO DE

LICENCIADO EN MATEMATICAS APLICADAS Y COMPUTACION

P R E S E N T A

ANTONIO BASTIDA CRUZ

ASESOR: MTRA. JUDITH JARAMILLO LOPEZ

ACATLAN, EDO DE MEX. ABRIL DEL 2002

INDICE

Introducción.

El siguiente trabajo tiene por objeto mostrar algunas formas de explotar la información contenida en las bases de datos relacionales, utilizando las herramientas que la tecnología ha desarrollado, y que deben ser aprovechadas para nuestro beneficio de la mejor manera posible, según sean nuestros requerimientos e infraestructura disponible.

Las bases de datos jerárquicas han evolucionado hacia las bases de datos relacionales y recientemente, con tecnología orientada a objetos; causando el reciclaje de muchos profesionales y el cambio de conceptos hacia esquemas más avanzados. Ligado al cambio desde el modelo jerárquico al relacional, y a hora al modelo orientado a objetos, la computación tradicional se ha ido descentralizando a un paulatino proceso de traslado, desde sistemas grandes y medianos hacia redes de computadoras personales, esto ha dado origen a la arquitectura cliente-servidor, la cual estructura las redes en estaciones de trabajo que envían y reciben datos desde y hacia los múltiples servidores.

En la actualidad, este continuo crecimiento de la tecnología de la computación, ha creado diversas herramientas que nos facilitan las tareas en todos los campos; en el caso de los sistemas de información, el acceso a los datos se ha venido facilitando continuamente, al grado de que contamos con herramientas de desarrollo visuales (que generan automáticamente la interfaz gráfica sin escribir una sola línea de código) como Visual Basic, que junto a otro potente lenguaje integrado, para el acceso a las bases de datos como lo es SQL, resulta una buena combinación, que facilita enormemente el desarrollo de aplicaciones para los sistemas de información, lo mismo ocurre con otras herramientas de desarrollo como Delphi , Power Builder, etc. Estas herramientas de cuarta generación están orientadas a que es lo que se quiere y no cómo se debe conseguir, además integran el desarrollo rápido de aplicaciones (RAD) que no es otra cosa que mostrar varias ventanas con herramientas al mismo tiempo, en diferentes zonas de la pantalla para facilitar el desarrollo.

La información, es lo más importante de una organización, por eso son necesarias aplicaciones que realicen la explotación de ésta, apoyándonos en una herramienta visual de desarrollo y de un lenguaje específicamente diseñado para el manejo de las bases de datos relacionales como lo es SQL, es posible crear una aplicación para tal efecto, de manera muy fácil y rápida; ya que es más sencillo para un usuario común utilizar elementos gráficos para solicitar información (cajas de verificación, cajas de opción, botones, iconos, listas, imágenes, etc.), que escribir directamente la consulta en SQL. Actualmente todo se proyecta a utilizar ambientes gráficos, desde el mismo sistema operativo, por lo que el desarrollo de aplicaciones no debe quedarse atrás, además siempre es más atractiva una aplicación gráfica, que otra que no lo es y por lo tanto más fácil de entender y de utilizar para los usuarios finales de las empresas u oficinas de gobierno.

Así en un ambiente en el que la tecnología evoluciona constantemente, creando más y mejores herramientas, el objetivo será utilizarlas de la mejor manera para obtener el máximo rendimiento posible. Porque aunque la tecnología mejore las herramientas, éstas deben ser configuradas correctamente, y *adaptadas al caso particular* para el que van a servir; de lo contrario la herramienta "solo funcionará", es decir no será utilizada por completo, ni al máximo. Por lo tanto las aplicaciones desarrolladas también "sólo funcionarán" y en el caso particular de las aplicaciones desarrolladas sobre un sistema manejador de bases de datos relacional el objetivo es configurar la base de datos y recuperar la información que se requiere de la mejor manera posible, para que las aplicaciones no sólo funcionen; si no que lo hagan de tal manera, que la base de datos realice menos trabajo, ya que esto se traduzca en menor tiempo de respuesta, es decir: eficiencia.

Al final quizás, el objetivo es que funcionen los sistemas, pero si es posible realizar que los sistemas funcionen más rápido, vale la pena intentarlo, porque el tiempo que se gane, mucho o poco puede ser aprovechado para realizar otras cosas, igual o más importantes.

Así, el objetivo de este documento, es reducir la carga de trabajo del RDBMS para mejorar el tiempo de respuesta de las aplicaciones, optimizando los diferentes procesos para eliminar las tareas innecesarias, ejecutándolas en el orden y en el ambiente adecuado. De esta forma la aplicación mejorara su velocidad de respuesta. En el presente trabajo se muestran los capítulos distribuidos de la siguiente manera:

En el primer capítulo se definen los conceptos que son necesarios para comprender mejor el presente trabajo: bases de datos relacionales, SQL (Structured Query Language, Lenguaje Estructurado de Consulta), arquitectura cliente servidor y RDBMS (Relational Database Management System, Sistema Manejador de Bases de Datos Relacional). Al desarrollar aplicaciones sobre una base de datos relacional, el papel del RDBMS es trascendental, pues es el que controla accesos, recupera la información, etc., es decir, se encarga de administrar todas las tareas de la base de datos; es muy importante conocer estos conceptos para que al desarrollar aplicaciones, se desarrolle de la mejor manera, aprovechado todas las características con que cuentan actualmente estas herramientas. Se entiende por aplicación a un sistema de información, un Data Warehouse, un juego, un sistema de ventas por Internet, un proyecto de investigación, etc.; que utilice una base de datos relacional y sea soportada por un RDBMS.

Antes de entrar en materia, se explica en el segundo capítulo, la importancia de conocer que se pretende realizar, para saber que acciones se deben tomar y así concretar lo que se plantea. Se deben definir los alcances, límites y herramientas, para iniciar el trabajo con orden, realizar una secuencia lógica y saber cuando detener el proceso, si los objetivos han sido alcanzados. En este capítulo se mencionan, algunos puntos a seguir en el momento de iniciar un proceso de optimización a un sistema, desde cero o a un sistema que ya se encuentra implementado.

En el tercer capítulo se destaca la idea de optimizar desde los niveles que pertenecen al DBA (Administrador de la Base de Datos), como la distribución física de los archivos que contienen la base de datos y la forma de organizarlos. Esto tiene mucho que ver a la hora de ejecutar las sentencias SQL, ya que si no hay una buena distribución y organización de los elementos físicos que integran la base de datos, estas sentencias sufrirán las consecuencias al incrementarse el tiempo de respuesta.

Aunque esto depende del DBA, es importante que los desarrolladores de la aplicación tengan conocimiento, porque de esto depende en gran medida que el rendimiento de la aplicación sea óptimo. Si la base de datos esta mal configurada o diseñada, el sistema no tendrá el mejor desempeño, aunque las sentencias SQL y los recursos estén bien. Por lo tanto un diseño adecuado de tablas, índices y una configuración adecuada de los archivos internos del manejador brindará un gran beneficio al realizar la aplicación. Se hará referencia específicamente al RDBMS de Oracle para ilustrar estos conceptos.

En el último capítulo se mostrarán casos prácticos de cómo saber el camino que siguen las sentencias que recuperan la información y de qué criterios debemos establecer para cambiar la forma de obtener el mismo resultado. Una de las partes más importantes de una aplicación es el diseño; si cuando la aplicación se esta diseñando, se hace de la mejor manera, la aplicación en el desarrollo no sufrirá mayores problemas y su rendimiento será el adecuado. Si no se hace así, habrá retrasos en los tiempos de desarrollo y lo más importante problemas de rendimiento; porque se alterarán tablas, campos, referencias, índices, etc. y al tratar de arreglar el problema se crearán otros, por soluciones no planificadas. Por eso en este capítulo se muestran muchos ejemplos de cómo cambiar una sentencia por otra y los beneficios o problemas que se generan al elegir una u otra.

Capítulo I. Conceptos Generales

1.1 Bases de datos

Antes de definir el concepto de una base de datos es necesario entender un poco acerca del modelo de datos relacional, que es el modelo que adopta la base de datos dentro de un RDBMS. Por lo tanto, cuando se haga referencia a una base de datos en el presente documento se debe entender que es a una base de datos relacional. Es importante conocer que existe un fundamento matemático que respalda estos conceptos, y que esos mismos fundamentos, sirven para entender mejor la manera en que un RDBMS realiza su trabajo y por lo tanto de como mejorarlo.

Modelos de datos

Un modelo de datos es un sistema formal y abstracto que permite describir los datos de acuerdo con reglas y convenios predefinidos, es formal pues los objetos del sistema se manipulan siguiendo reglas perfectamente definidas y utilizando exclusivamente los operadores definidos en el sistema, independientemente de lo que estos objetos y operadores puedan significar. Se compone de los siguientes elementos:

- Estructuras de datos: es la colección de objetos abstractos formados por los datos.
- Operadores entre las estructuras: el conjunto de operadores con reglas bien definidas que permiten manipular a dichas estructuras.
- Definiciones de integridad: es una colección de conceptos y reglas que permiten expresar que valores de datos pueden aparecer válidamente en el modelo.

Jerárquico

Consiste en una colección de elementos (registros) que se conectan entre sí por medio de enlaces, cada registro es una colección de atributos (campos), que contienen un solo valor cada uno de ellos. Un enlace es una asociación o unión entre dos elementos exclusivamente; por tanto, este concepto es similar al de enlace para el modelo de red. Ejemplo: Consideremos una base de datos, que contiene la relación alumno-materia de un sistema escolar. Existen dos tipos de registros en este sistema, alumno y materia, el registro alumno consta de tres campos: NombreA, Control y Especialidad; el registro Materia esta compuesto de tres campos: Clave, NombreM y Créditos. En este tipo de modelos la organización se establece en forma de árbol, donde la raíz es un nodo ficticio. Así tenemos que, una base de datos jerárquica es una colección de árboles de este tipo, el contenido de un registro específico puede repetirse en varios sitios (en el mismo árbol o en varios árboles). La repetición de los registros tiene dos desventajas principales:

- Puede producirse una inconsistencia de datos.
- El desperdicio de espacio.

Red

También denominado modelo CODASYL. Fue el primero en aparecer comercialmente, a principios de los años 70. Una base de datos de red como su nombre lo indica, esta formada por una colección de registros, los cuales están conectados entre sí por medio de enlaces. El registro es similar a una entidad, como las empleadas en el modelo entidad-relación. Un registro es una colección de campos (atributos), cada uno de los cuales contiene almacenado un solo valor, el enlace es la asociación entre dos registros exclusivamente, así que podemos verla como una relación estrictamente binaria. Una estructura de datos de red, abarca más que la

estructura de árbol, porque un nodo hijo en la estructura de red puede tener más de un padre. En otras palabras, la restricción de que en un árbol jerárquico cada hijo puede tener un solo padre, se hace menos severa.

Relacional

El modelo de datos relacional fue introducido por Edgard F. Codd en 1970, desde entonces las bases de datos que utilizan este modelo se pueden encontrar en muchos lugares; muchas aplicaciones y usuarios de bases de datos tienen a su alcance software para el manejo de bases de datos relacionales de muchas marcas comerciales, los hay para computadoras personales, medianas y grandes. Pocas veces se reflexiona en cual es el motivo de su éxito y por qué se ha popularizado tanto, una de las causas principales es el modelo de datos matemático que lo respalda.

La ventaja del modelo relacional es que los datos se almacenan, al menos conceptualmente, de un modo en que los usuarios entienden con mayor facilidad, los datos se almacenan como tablas y las relaciones entre las filas y las tablas son visibles en los datos, este enfoque permite a los usuarios obtener información de la base de datos sin asistencia de sistemas profesionales de administración de información.

Las características más importantes de los modelos relacionales son:

- Las entradas en la tabla tienen un solo valor (son atómicos); no se admiten valores múltiples, por lo tanto la intersección de un renglón con una columna tiene un solo valor, nunca un conjunto de valores.
- Todas las entradas de cualquier columna son de un solo tipo. Por ejemplo, una columna puede contener nombres de clientes, y en otra puede tener fechas de nacimiento.
- Cada columna posee un nombre único.
- El orden de las comunas no es de importancia para la tabla, las columnas de una tabla se conocen como atributos.
- Cada atributo tiene un dominio, que es una descripción física y lógica de valores permitidos.
- No existen 2 filas en la tabla que sean idénticas.
- La información en las bases de datos son representados como datos explícitos, no existen apuntadores o ligas entre las tablas.
- El enfoque relacional es sustancialmente distinto de otros enfoques en términos de sus estructuras lógicas y del modo de las operaciones de entrada/salida.
- En el enfoque relacional, los datos se organizan en tablas llamadas relaciones, cada una de las cuales se implanta dentro de un archivo de datos.
- En terminología relacional una fila en una relación representa un registro o una entidad.
- Cada columna en una relación representa un campo o un atributo.
- Así, una relación se compone de una colección de registros cuyos propietarios están descritos por cierto número de atributos predeterminados implantados como campos.

Conceptos del modelo relacional

El modelo de datos relacional representa la base de datos como una colección de relaciones, en términos informales, cada relación semeja una tabla o, hasta cierto punto un archivo simple. Si visualizamos una relación como una tabla de valores, cada fila de la tabla representa una colección de valores relacionados entre sí; dichos valores se pueden interpretar como hechos que describen una entidad o vínculo entre entidades del mundo real. El nombre de la tabla y los nombres de las columnas ayudan a interpretar el significado de los valores que están en cada fila, en la terminología del modelo relacional una fila se denomina tupla, una cabecera de columna es un atributo y la tabla es una relación. El tipo de datos que describe a los tipos de valores que pueden aparecer en cada columna se llama dominio.

Los valores nulos representan atributos cuyos valores se desconocen o no existen para algunas tuplas. La definición de relación puede expresarse también como sigue: una relación *r*(R) es un subconjunto del producto cartesiano de los dominios que definen a R:

$$r(R) \subseteq (dom(A_1) \times dom(A_2) \times \ldots dom(A_n) \times)$$

El producto cartesiano especifica todas las combinaciones posibles de valores de los dominios implicados. Así pues si denotamos el número de valores o cardinalidad de un dominio D con | D | y suponemos que todos los dominios son finitos, el número total de tuplas del producto cartesiano es:

$$| dom(A_1) | * | dom(A_2) | * \ldots * | dom(A_n) |$$

Definiciones de integridad en el modelo relacional

Los conceptos de integridad para el modelo relacional son llave primaria, foránea, los valores nulos y dos reglas de integridad que se enuncian a continuación.

Llave primaria

Dentro de los atributos debe haber uno o varios que sirvan para distinguir cada entidad en la relación, en tal caso cada una de ellas se denomina llave candidata, es común designar a una de las llaves candidatas como llave primaria de la relación, ésta, es la llave candidata cuyos valores sirven para identificar las tuplas de la relación, es un atributo el cual definimos como atributo principal, es una forma única de identificar a una entidad. Cabe señalar que, cuando un esquema tiene varias llaves candidatas, la elección de una para fungir como llave primaria es arbitraria; sin embargo, siempre es mejor escoger una llave primaria con un solo atributo o un número reducido de atributos y que sus valores sean únicos. Esta llave primaría, funciona como un índice único, es decir, no se repiten sus registros y sus datos están ordenados. Al realizar sentencias sobre tablas que contengan una llave primaria, es importante utilizarla, para ello necesitamos ubicar el campo o campos que la forman; como primera restricción, en una sentencia de selección de registros, sin utilizar una función que cambie su valor original, de lo contrario, no se aprovechará la existencia de la llave para mejorar el rendimiento de una sentencia, y sólo servirá como candado de integridad para que no se repitan registros.

Cabe hacer un paréntesis para definir brevemente el concepto de índice, que es muy importante conocer desde ahora: un índice es un objeto de la base de datos lógica y físicamente independiente, que es utilizado para acceder los datos de una tabla a través de la columna que lo tiene asociado, esto es, que en la tabla hay una columna ligada a un índice, el índice tiene las direcciones para acceder a los registros de la tabla, y en el índice los datos se encuentran ordenados. Así los datos se localizan más rápido en el índice y luego se extraen los registros de la tabla con las direcciones obtenidas del índice. Una llave primaria, se encuentra asociada a un índice único, pero pueden existir índices con registros repetidos, para otras columnas.

Ejemplo:

En la relación Alumnos anterior la llave primaria es: `No.Cta. (Número de cuenta)`
En la relación Materias la llave es: `Clave-mat. (Clave de la Materia)`
En Evaluaciones es: `La unión de No.Cta y Clave-mat.`

Restricciones de integridad

La restricción de integridad de entidades, establece que ningún valor de la llave primaria puede ser nulo. Esto es porque el valor de la llave primaria sirve para identificar las tuplas individuales en una relación; el que la llave primaria tenga valores nulos implica que no se puedan identificar algunas tuplas.

La restricción de integridad referencial se especifica entre dos relaciones y sirve para mantener la consistencia entre tuplas de las dos relaciones. En términos informales, la restricción de integridad referencial establece que una tupla en una relación que haga referencia a otra relación deberá referirse a una tupla existente en esa relación. Para dar una definición más formal de integridad referencial primero debemos definir el concepto de llave externa (foránea).

Llave Externa (Foránea)

Las condiciones que debe satisfacer una llave externa CE (conjunto externo) especifican una restricción de integridad entre los dos esquemas de relaciones R_1 y R_2. Un conjunto de atributos CE en el esquema de relación R_1 es una llave externa si satisface las dos reglas siguientes:

- Los atributos de CE tienen el mismo dominio que los atributos de la llave primaria CP (conjunto primario) de otro esquema de relación R_2, se dice que los atributos de CE hacen referencia o se refieren a la relación R_2.
- Un valor de CE en una tupla t_1 de R_1 ocurre como valor de CP en alguna tupla t_2 de R_2 o bien es nulo. En el primer caso, tenemos t_1 [CE] = t_2 [CP], y decimos que la tupla t_1 hace referencia o se refiere a la tupla t_2.

Hace referencia a una llave primaria en otra relación. Una relación puede tener una o varias llaves foráneas.

Ejemplo: Los atributos No.Cta y Clave-mat son llaves externas en la relación Evaluaciones.

Una relación puede tener varias llaves, la que se elige para identificar a una relación se le llama llave primaria, en el modelo relacional, es el concepto de llave la única forma de encontrar una entidad. Un valor nulo denotado por '?', proporciona la posibilidad de manejar situaciones como las siguientes:

- Se crea una tupla y no se conocen los valores de los atributos.
- Se agrega un atributo a una relación ya existente.
- Se usan para no introducir valores numéricos al hacer cálculos.

El álgebra relacional

El álgebra relacional es una colección de operaciones que sirven para manipular relaciones. Estas operaciones sirven, por ejemplo para seleccionar tuplas de relaciones individuales y para combinar tuplas relacionadas a partir de varias relaciones con el fin de especificar una consulta (o solicitud de obtención) de la base de datos. El resultado de cada operación es una nueva relación.

Las operaciones del álgebra relacional suelen clasificarse en dos grupos, uno contiene las operaciones de la teoría matemática de conjuntos; es posible aplicarlas porque las relaciones se definen como conjuntos de tuplas. Entre las operaciones de conjuntos están la unión, la intersección, la diferencia y el producto cartesiano. El otro grupo consiste en operaciones

creadas específicamente para bases de datos relacionales, como: selección, proyección y reunión, entre otras. Para ejemplificar estas operaciones se utilizarán las siguientes relaciones:

Sean *R* (Figura 1.1) y *S* (Figura 1.2) relaciones con esquema en {A,B,C} y *Q* (Figura 1.3) en {D, E, F}, estos conjuntos serán utilizados para ejemplificar las diferentes operaciones que se explican más adelante:

A	B	C
A1	B1	C1
A1	B2	C1
A2	B1	C2

Figura 1.1. Relación **R**.

A	B	C
A1	B1	C1
A2	B2	C1
A2	B2	C2

Figura 1.2. Relación **S**.

D	E	F
D1	E1	F1
D2	E2	F1
D2	E2	F2

Figura 1.3. Relación **Q**.

Operación de selección

Sirve para seleccionar un subconjunto de tuplas de la relación **R** que cumplen con una condición (simple o compuesta) sobre los valores para uno o varios de los atributos. Se denota por:

$$\sigma_{(\text{<condición de selección>})}(\text{<Nombre de la relación>}).$$

El operador seleccionar es unitario; esto es, se plica a una sola relación. Por lo tanto no podemos usar SELECCIONAR para obtener tuplas de más de una relación. Por añadidura la operación de selección se aplica a cada tupla individualmente; por tanto, las condiciones de selección no pueden abarcar más de una tupla. El grado de la relación resultante de una operación SELECCIONAR es el mismo que el de la relación original R a la que se le aplicó la operación, porque tiene los mismos atributos que R. El número de tuplas de la operación resultante siempre es menor que el número de tuplas de la relación original R o igual a ella. La operación seleccionar es conmutativa:

$$\sigma_{\text{<cond1>}}(\sigma_{\text{<cond2>}}(R)) = \sigma_{\text{<cond2>}}(\sigma_{\text{<cond1>}}(R))$$

Ejemplo:

$\sigma_{B=B1}(R)$, el resultado se muestra en la figura 1.4.

A	B	C
A1	B1	C1
A2	B1	C2

Figura 1.4. Resultado de la operación de Selección sobre la relación R.

Operación de proyección

Es también una operación unitaria, el resultado es un subconjunto de dominios, permite obtener subrelaciones de otras más grandes seleccionando algunos atributos, y se eliminan las tuplas repetidas. Si visualizamos una relación como una tabla, la operación SELECCIONAR obtiene algunas filas de la tabla y desecha otras. La operación PROYECTAR, en cambio, selecciona ciertas columnas de la tabla y desecha las demás. Si sólo nos interesan ciertos atributos de una relación, "proyectaremos" la relación sobre esos atributos con la operación PROYECTAR, se denota con:

$$\pi_{\text{<lista de atributos>}}(\text{<nombre de la relación>});$$

Ejemplo:
$\pi_{<B,C>}(R)$, el resultado esta en la Figura 1.5.

B	C
B1	C1
B2	C1
B1	C2

Figura 1.5. Resultado de la operación Proyección sobre la relación R.

Operaciones con conjuntos

El siguiente grupo de operaciones del álgebra relacional son las operaciones matemáticas normales de conjuntos, se aplican al modelo relacional porque las relaciones se definen como conjunto de tuplas y pueden servir para procesar las tuplas de dos relaciones como conjuntos. Se utilizan varias operaciones de la teoría de conjuntos para combinar de varias maneras los elementos de dos conjuntos, entre ellas: UNION, INTERSECCIÓN y DIFERENCIA. Estas operaciones son binarias, es decir, se aplican a dos conjuntos. Al adaptar estas operaciones a las bases de datos relacionales, debemos asegurarnos de que se puede aplicar a dos relaciones para que el resultado también sea una relación válida. Por esto es necesario que las relaciones deban tener el mismo tipo de tuplas (compatibilidad de unión).

Se dice que dos relaciones $R(A_1,A_2,...,A_n)$ y $S(B_1,B_2,...,B_n)$ son compatibles con la unión si tienen el mismo grado de n y si $dom(A_i) = dom(B_i)$ para $1 \leq i \leq n$. Esto significa que las dos relaciones tienen el mismo número de atributos y que para cada par de atributos correspondientes tienen el mismo dominio. Podemos definir las tres operaciones para dos relaciones compatibles con la unión R y S como sigue:

Unión

$R \cup S$. El resultado de esta operación es una relación que incluye todas las tuplas que están en R o en S o en ambas, las tuplas repetidas se eliminan. En SQL existe una variante llamada UNION ALL que no elimina los registros repetidos, es útil cuando se sabe que los conjuntos contienen valores distintos, se evita el proceso de ordenación y eliminación de registros repetidos de la sentencia UNION normal, lo cual ahorra recursos (tiempo).

Intersección

$R \cap S$. El resultado de esta operación incluye las tuplas que están tanto en R como en S. Útil para saber si hay registros duplicados entre dos tablas.

Diferencia

$R - S$. El resultado de esta operación es la relación con las tuplas que están en R pero no en S. Para saber que datos están en una tabla y no en otra.

Ejemplos:

R ∪ S. Unión, figura 1.6.

A	B	C
A1	B1	C1
A1	B2	C1
A2	B1	C2
A2	B2	C1
A2	B2	C2

Figura 1.6. Resultado de la operación Unión sobre las relaciones R y S.

R ∩ S. Intersección, figura 1.7.

A	B	C
A1	B1	C1

Figura 1.7. Resultado de la operación Intersección sobre las relaciones R y S.

R – S. Diferencia, figura 1.8.

A	B	C
A1	B2	C1
A2	B1	C2

Figura 1.8 Resultado de la operación Diferencia sobre las relaciones R y S.

Cabe señalar que tanto la UNIÓN como la INTERSECCIÓN son operaciones conmutativas; es decir:

$R \cup S = S \cup R$ y $R \cap S = S \cap R$

Ambas operaciones pueden aplicarse a cualquier número de relaciones, y las dos son operaciones asociativas:

$R \cup (S \cup T) = (R \cup S) \cup T$ y $(R \cap S) \cap T = R \cap (S \cap T)$

La operación diferencia no es conmutativa:

$R - S \neq S - R$

Producto Cartesiano

Se denota por: X .Obtiene todas las tuplas que se construyen concatenando cada tupla de *R* con otra de *S*. En este caso los dominios de *R* y *S* no tienen que ser los mismos. En general, el resultado de $R(A_1,A_2,...,A_n)$ X $S(B_1,B_2,...,B_n)$ es una relación Q con n+m atributos $Q(A_1,A_2,...,A_n, B_1,B_2,...,B_m)$ en ese orden. La relación resultante Q tiene una tupla por cada combinación de tuplas: una de *R* y una de *S*. Por tanto si *R* tiene n_R tuplas y *S* tiene n_S tuplas, *R* X *S* tendrá $n_R * n_S$ tuplas. El producto cartesiano crea tuplas con los atributos combinados de dos relaciones, después podemos seleccionar sólo las tuplas relacionadas de las dos relaciones especificando una condición de selección apropiada. Con esta secuencia de PRODUCTO CARTESIANO seguido de SELECCIONAR se utiliza con mucha frecuencia para identificar y seleccionar tuplas relacionadas de dos relaciones, se creó una operación especial, llamada REUNION, con el fin de especificar esta secuencia con una sola operación.

Ejemplo: **R X Q**, Producto Cartesiano, figura 1.9.

A	B	C	D	E	F
A1	B1	C1	D1	E1	F1
A1	B1	C1	D2	E2	F1
A1	B1	C1	D2	E2	F2
A1	B2	C1	D1	E1	F1
A1	B2	C1	D2	E2	F1
A1	B2	C1	D2	E2	F2
A2	B1	C2	D1	E1	F1
A2	B1	C2	D2	E2	F1
A2	B1	C2	D2	E2	F2

Figura 1.9. Resultado de la operación Producto Cartesiano sobre las relaciones R y Q.

Operación Reunión

Sirve para combinar tuplas de dos relaciones en una sola tupla. Esta operación es muy importante en cualquier base de datos relacional que comprenda más de una relación (casi todas), porque permite procesar los vínculos entre las relaciones. La forma general de una operación REUNIÓN con dos relaciones es:

$R(A_1,A_2,...,A_n)$ y $S(B_1,B_2,...,B_n)$ **es** $R \bowtie_{\text{<condición de reunión>}} S$.

Ejemplo, sean las relaciones *R* (figura 1.10) y *S* (figura 1.11):

R={A,B}

A	B
A1	B1
A2	B2

Figura 1.10. Relación **R**.

S={A,D}

A	D
A1	D1
A1	D2
A2	D1

Figura 1.11. Relación **S**.

$R \bowtie_{<A=A1>} S$ = {A,B,D}, operación Reunión figura 1.12.

A	B	D
A1	B1	D1
A1	B1	D2

Figura 1.12. Resultado de la operación Reunión sobre las relaciones R y S.

El resultado de la **REUNION** es una relación *Q* con n+m atributos $Q(A_1,A_2,...,A_n, B_1,B_2,...,B_m)$, en ese orden. Tiene una tupla por cada combinación de tuplas una de R y una de S, siempre que la combinación satisfaga la condición de reunión. Esta es la principal diferencia entre el PRODUCTO CARTESIANO y la REUNION.

Conjunto completo de operaciones del álgebra relacional

Se ha demostrado que el conjunto de operaciones del álgebra relacional ($\sigma,\pi,\cup,-,X$) es un conjunto completo; es decir, cualquiera de las otras operaciones del álgebra relacional se puede expresar como una secuencia de operaciones de ese conjunto. Por ejemplo la operación intersección se puede emplear Unión y Diferencia como sigue:

$$R \cap S \equiv (R \cup S) - ((R - S) \cup (S - R))$$

Aunque en términos estrictos, la intersección no es indispensable, resulta poco cómodo especificar esta expresión compleja cada vez que deseemos especificar una intersección. Y así con otras operaciones se han incluido en el álgebra relacional por comodidad más que por necesidad, como la división.

División

La división es útil para un tipo especial de consultas que se presenta ocasionalmente en aplicaciones de bases de datos. Un ejemplo es "obtener los nombres de los empleados que trabajan en todos los proyectos en los que trabaja 'José Pérez'. Para expresar esta consulta con la operación división, precedemos como sigue. Primero, obtenemos la lista de números de los proyectos en los que trabaja 'José Pérez', colocando el resultado en la relación intermedia NUM_PEREZ:

PEREZ $\leftarrow \sigma_{\text{nombrep='José' y apellido = 'SILVA'}}$(EMPLEADO)
NUM_PEREZ $\leftarrow \pi_{\text{nump}}$ (TRABAJA_EN * $_{\text{NSSE=NSS}}$ PEREZ)

En seguida creamos una relación intermedia NSS_NUMSP que incluye una tupla (NUMP,NSSE) por cada vez que el empleado cuyo número de seguro social es NSSE trabaja en el proyecto cuyo número es NUMP:

NSS_NUMSP $\leftarrow \pi_{\text{nump, nsse}}$ (TRABAJA_EN)

Por último aplicamos la operación DIVISIÓN en las dos relaciones, obteniendo los números de seguro social de los empleados que queremos:

NSSS(NSS) ← NSS_NUMSP ÷ NUMP_PEREZ
RESULTADO $\leftarrow \pi_{\text{nombrep, apellido}}$ (NSSS * EMPLEADO)

Parecido a una subconsulta en una sentencia SQL, es decir una consulta dentro de otra.

Otras operaciones

Permutación

Esta operación se aplica a una sola relación, consiste en cambiar el orden de las columnas. Se denota por P $_{\text{<lista_columnas>}}$ *R* en donde se indican el orden en que estarán las columnas de la relación original. En este caso la primera columna es la i, la segunda la j y la tercera la k. Ejemplo: $P_{<B,C,A>}(R)$, operación Permutación, figura 1.13.

B	C	A
B1	C1	A1
B2	C1	A1
B1	C2	A2

Figura 1.13. Resultado de la operación Permutación sobre las relaciones R y S.

El Modelo Entidad-Relación (E-R)

Propuesto por Peter Chen a mediados de los años setenta como medio de representación conceptual de los problemas y para representar la visión de un sistema de forma global. Físicamente adopta la forma de un grafo escrito en papel al que se denomina diagrama Entidad-Relación. Sus elementos fundamentales son las entidades y las relaciones.

- Una entidad caracteriza a un tipo de objeto, real o abstracto, del problema a modelar. Toda entidad tiene existencia propia, es distinguible del resto de las entidades, tiene nombre y posee atributos definidos en un dominio determinado, una entidad es todo aquello de lo que se desea almacenar información. En el diagrama E-R las entidades se representan mediante rectángulos. Es importante decidir si una entidad es realmente necesaria a la hora de crear las tablas, esto es, si sus datos se pueden obtener de otras, para evitar redundancia; que como veremos en casos especiales es necesaria para mejorar el rendimiento.
- Una relación es una asociación o relación matemática entre varias entidades. Las relaciones también se nombran y representan en el diagrama E-R mediante flechas, rectángulos, rombos y cuadros redondeados, Fig. 1.14 Cada entidad interviene en una relación con una determinada cardinalidad. La cardinalidad (número de instancias o elementos de una entidad que pueden asociarse a un elemento de la otra entidad relacionada) se representa mediante una pareja de datos, en minúsculas, de la forma (cardinalidad mínima, cardinalidad máxima), asociada a cada uno de las entidades que intervienen en la relación. Son posibles las siguientes cardinalidades: (0,1), (1,1), (0,n), (1,n), (m,n). También se informa de las cardinalidades máximas con las que intervienen las entidades en la relación.

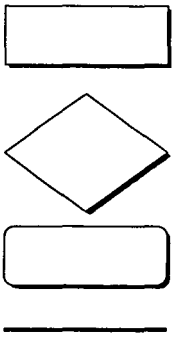


Fig. 1.14 Elementos que forman un diagrama Entidad-Relación.

El tipo de relación se define tomando el máximo de la cardinalidad que intervienen en la relación. Hay cuatro tipos posibles:

- Una a una (1:1). En este tipo de relación, una vez fijado un elemento de una entidad se conoce la otra. Ejemplo: nación y capital.
- Una a muchas (1:N). Ejemplo: cliente y pedidos.
- Muchas a una (N:1). Simetría respecto al tipo anterior según el punto de vista de una u otra entidad.
- Muchas a muchas (N:N). Ejemplo: personas y viviendas.

Toda entidad debe ser unívocamente identificada y distinguible mediante un conjunto de atributos (quizás un solo atributo) denominado identificador o llave principal o primaria. Puede haber varios identificadores posibles para una misma entidad, en cuyo caso se ha de escoger uno de ellos como identificador principal siendo el resto identificadores alternativos. Ejemplo: CURP y número de seguridad social de una persona. Hay unas normas de sentido común a seguir cuando se dibuja un diagrama E-R, la primera es emplear preferentemente líneas rectas en las relaciones y evitar en lo posible que estas líneas se crucen, suele usarse nombres para describir las entidades y verbos para las relaciones, esto es lógico ya que las entidades se ponen en común cuando se realiza alguna acción. Los verbos empleados no necesariamente tienen que ser siempre infinitivos.

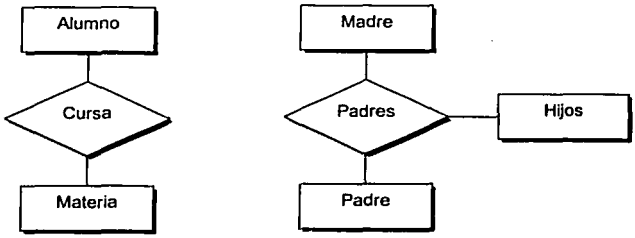
Ejemplo: Se desea almacenar información sobre personas y los coches que eventualmente posean, una misma persona puede poseer varios coches aunque puede haber personas que no posean ningún coche. Los coches se identifican mediante su número de matrícula y las personas mediante su documento nacional de identidad. Todo coche tiene un solo propietario, se ha de almacenar la fecha en que una determinada persona adquirió un determinado coche.

Un esquema único que agrupe a todos los atributos de la entidad coche (matrícula, marca, modelo, etc.), de la entidad persona (CURP, nombre, dirección, etc) y de la relación entre ambas entidades (fecha de compra), ocasiona problemas:

- Personas sin coche (valores nulos y gasto de espacio de almacenamiento).
- Multiplicidad de almacenamiento (redundancia) de los atributos de una persona si ésta es propietaria de más de un coche.
- Modificación del valor de un atributo de una persona en una sola de sus apariciones en la instancia de la base de datos (inconsistencia).

Para evitar estos problemas se separa el esquema único de la base de datos en tres separados para coche, persona y la relación entre ambos

Muchas veces es posible simplificar el diagrama E-R eliminando entidades innecesarias. Por ejemplo, si una entidad que interviene únicamente en una relación del tipo una a una (1:1) no tiene como atributo más que su código, este atributo puede incluirse en la entidad con la que está relacionada eliminar tanto la relación como la entidad. Y esto ahorra espacio y tiempo a la hora de recuperar la información, por lo tanto es más rápido, de esta manera, desde el diseño se puede ir afinando la aplicación final. Ejemplos de diagramas E-R (sin considerar los atributos, sólo las entidades) de las relaciones ALUMNO-MATERIA es de grado 2, ya que intervienen la entidad ALUMNO y la entidad MATERIA, la relación PADRES, es de grado 3, ya que involucra las entidades PADRE, MADRE e HIJO, Figura 1.15 (a y b).



a) Relación de grado dos. b) Relación de grado tres.

Fig. 1.15 Ejemplos de diagramas entidad - relación.

¿Qué es una Base de datos?

Una base de datos es un conjunto de datos relacionados entre sí, por datos entendemos hechos u objetos conocidos que pueden registrarse y que tienen un significado en el mundo real, por ejemplo los nombres, números telefónicos y direcciones de personas que conocemos, se trata de un conjunto de datos relacionados entre sí y que tienen un significado común; por tanto constituyen una base de datos. La definición anterior es muy general; por ejemplo podemos considerar el conjunto de palabras que forman esta página de texto como datos relacionados entre sí, de modo que son una base de datos. Pero la acepción común del término base de datos suele ser más restringida, una base de datos tiene las siguientes propiedades:

- Una base de datos representa algún aspecto del mundo real, en ocasiones llamado mini-mundo o universo de discurso. Las modificaciones del mini-mundo se reflejan en la base de datos. [AESB2000]
- Una base de datos es un conjunto de datos lógicamente coherente, con cierto significado inherente. Una colección aleatoria de datos no puede considerarse una base de datos. [AESB2000]
- Toda base de datos se diseña, construye y puebla con datos para un propósito específico. Está dirigida a un grupo de usuarios y tiene ciertas aplicaciones preconcebidas que interesan a dichos usuarios. [AESB2000]
- Para una empresa, una base de datos es la parte más importante para su negocio. Si funciona correctamente, la empresa gana dinero; si no funciona, la empresa pierde dinero.

En otras palabras una base de datos tiene una fuente de la cual se derivan los datos, interactúa con el mundo real y un público que esta activamente interesado en su contenido, su estado y su estructura, haciendo modificaciones sobre ella si es necesario, y lo más importante, es que los usuarios necesitan explotar la información de manera eficiente.

Clasificación de las bases de datos

Desde el punto de vista de la consulta, una base de datos puede ser:

- **Online**, si su soporte físico es la memoria de un servidor y es consultada a distancia realizando operaciones de altas, bajas y cambios, mediante comunicación remota desde una terminal; de esta forma, el usuario se conecta al servidor que contiene la información, realiza las operaciones que tiene que hacer y se desconecta. Sólo utiliza el servidor de la base de datos el tiempo que tarda en hacer la manipulación de datos, compartiendo el tiempo y servidor con otros múltiples usuarios que también pueden estar accediendo.
- **Autónoma**, si se encuentra en un soporte independiente, fácilmente manejable e intercambiable, y puede ser consultada en la computadora del propio usuario. Éste es el caso, por ejemplo, de las bases de datos que actualmente se están ofreciendo en soporte CD-ROM, como enciclopedias, diccionarios, juegos, etc.

Desde el punto de vista de la comercialización, una base de datos puede ser:

- **Abierta** si se ofrece comercial o gratuitamente al mercado o público en general que pueda estar interesado. Por ejemplo, una base de datos de legislación, estadísticas varias, productos comerciales, etc.
- **Cerrada** si la base de datos es desarrollada por una persona física o jurídica, ya sea privada o pública, para su uso interno. Por ejemplo, una base de datos de clientes de un gran almacén, de contribuyentes, de ventas en una empresa, etc.

Desde el punto de vista de la localización geográfica, una base de datos puede ser:

- **Centralizada**: todos los datos están físicamente almacenados en el mismo servidor y bajo un control unitario. Los datos pueden estar compartidos por múltiples aplicaciones y usuarios.
- **Distribuida**: los datos están almacenados en varios servidores geográficamente repartidos y conectados mediante una red, vistos como una sola base de datos lógica. La administración de la base de datos puede realizarse en varios lugares distintos y por personas distintas. Toda esta problemática debe ser transparente a los usuarios, los cuales no necesitan saber dónde están realmente almacenados los datos a los que acceden.

Diseño de bases de datos

Es sencillo diseñar una base de datos, pero a menudo hay que reconsiderar posteriormente la estructura de los datos, lo cual ocasiona retrasos y modificaciones, es más lenta la obtención de un diseño óptimo, pero el tiempo invertido se recupera al no tener que volver atrás para replantearse el diseño de los datos. Un buen diseño es la clave para iniciar correctamente el desarrollo de una aplicación, un diseño apresurado o simplemente bosquejado puede mostrarse inservible o muy mejorable cuando la aplicación ya está parcialmente codificada, o el administrador de la base de datos ya tiene organizados el mantenimiento y el control de acceso a los datos. Si un sistema se concibe desde el diseño de la base de datos, con la idea de optimizar, se debe analizar muy bien la construcción de tablas e índices, se tiene que tomar en cuenta el espacio que ocupan, sus ventajas y desventajas en el sistema, para valorar si es viable su construcción o no.

Esquema: diseño general de la base de datos a nivel lógico. Incluye el tipo de datos y las relaciones entre ellos, es de naturaleza fija y sólo se altera excepcionalmente. El esquema se define y se mantiene utilizando el lenguaje de definición de datos (DDL), desde este punto conviene ir pensando en como optimizar, conviene revisar el diseño, pensando en si será aprovechado o no; si hay puntos que no se aprovechan, vale la pena pensar si en realidad es necesario que se creen.

Instancia: contenido concreto de la base de datos en un momento dado. Varía con el tiempo, al añadir, eliminar o modificar datos; utilizando el lenguaje de modificación de datos (DML), es importante tomar en cuenta que no es lo mismo una base de datos recién creada o parcialmente llena, que una base de datos con un año en producción, por eso las pruebas de optimización se deben realizar en un ambiente lo más parecido posible al real.

El diseño de una base de datos se realiza a dos niveles. El primero es el nivel conceptual, en la cual se contempla una estructura abstracta y no implementable directamente en un RDBMS; el segundo es el nivel físico, en el cual la base de datos es ya implementable. Detalladamente, las fases del diseño de una base de datos son las siguientes:

- Descripción en lenguaje natural.
- Diagrama Entidad-Relación (E-R). Estos diagramas modelan el problema mediante entidades asociadas por relaciones. Adoptan la forma de grafos donde los datos se relacionan mediante flechas. El diagrama E-R no depende del modelo de datos.
- Elección del modelo de datos (usualmente el relacional)
- Conversión del diagrama E-R al modelo relacional (tablas)
- Normalización (eliminar diversos defectos de diseño).
- *Optimización*

Las tres primeras fases pertenecen al nivel conceptual del diseño de bases de datos mientras que las tres últimas se relacionan con el nivel físico.

Sistema Manejador de Bases de Datos Relaciónales (RDBMS)

Un sistema manejador de bases de datos en general es un conjunto de programas que permite a los usuarios crear y mantener una base de datos, por tanto un manejador de bases de datos es un sistema de software que automatiza la creación y manipulación de las tablas, suministrando facilidades para crear y eliminar tablas, y permitiendo asimismo el almacenamiento, recuperación y cambio de los datos mantenidos en las tablas. Los sistemas manejadores de bases de datos pueden ser soportados por los mainframes más grandes como por los PC más pequeños, estos sistemas incorporan toda la teoría de las bases de datos relaciónales, de ahí que sirvan como medio para sustentar la teoría relacional con la realidad de la información. Los RDBMS utilizan en su mayoría el lenguaje SQL para manejar la información de las bases de datos.

Para definir una base de datos hay que especificar los tipos de datos, las estructuras y las restricciones de los datos que se almacenarán en ella. Construir una base de datos es el proceso de guardar los datos mismos en algún medio de almacenamiento controlado por el RDBMS. En la manipulación de una base de datos intervienen funciones como consultar la base de datos para obtener datos específicos, actualizar la base de datos para reflejar cambios en el *mini-mundo* y generar informes a partir de los datos. No hace falta un software RDBMS de propósito general para implementar una base de datos computarizada, podríamos escribir un conjunto de programas para crear o mantener la base de datos, con lo cual se crearía un software RDBMS de propósito específico. En cualquier caso, ya sea que utilicemos un RDBMS de propósito general o no, casi siempre requeriremos un software de gran capacidad para manipular la base de datos. Al conjunto de formado por la base de datos y el software RDBMS lo llamaremos sistema de base de datos (figura 1.16).

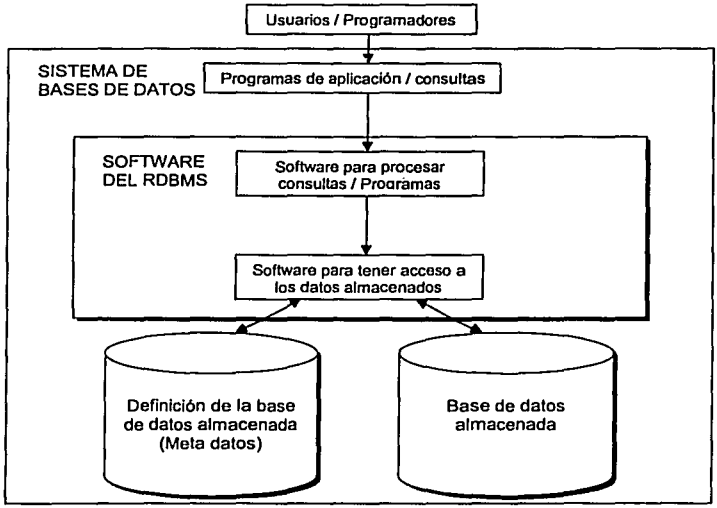


Figura 1.16. Entorno simplificado de un sistema de base de datos.

Un sistema manejador de bases de datos proporciona facilidades para:

- Crear una base de datos nueva y vacía
- Eliminar una base de datos existente
- Añadir tablas nuevas y vacías a una base de datos
- Guardar un diccionario de datos
- Borrar tablas existentes de una base de datos
- Acceder a datos de las tablas
- Compartir datos
- Añadir datos a las tablas
- Borrar datos de las tablas
- Modificar datos de las tablas
- Copiar datos de una tabla a otra
- Mover datos de una tabla a otra
- Crear, modificar y eliminar relaciones entre las tablas
- Crear, modificar y eliminar funciones, triggers* o procedimientos almacenados
- Crear, modificar y eliminar usuarios

Entre otras tantas tareas, incluyendo las propias de cada manejador, existen muchas utilidades para tales tareas, en su mayoría con una interfaz gráfica diseñada específicamente para esos fines, lo que facilita mucho su realización aún sin conocer la teoría del modelo relacional. Evitando tener que escribir en lenguaje SQL la instrucción que realice determinada tarea. Las bases de datos pueden compartir datos, los usuarios pueden acceder a través de una red, a bases de datos ubicadas en un mainframe, servidor, estación de trabajo (workstation), un RDBMS en este aspecto tiene ventajas importantes como:

- Permite múltiples usuarios
- Controla el acceso concurrente a datos compartidos
- Coordina la ejecución de los procesos compartidos
- Garantiza la seguridad de los datos

Es recomendable la utilización de una única base de datos, que se respalde con la regularidad necesaria, que varias bases de datos con la misma información para evitar cambios no deseados en una u otra base de datos. Cuando el volumen de información crece, la velocidad de acceso a la información se vuelve cada vez más importante, cualquier RDBMS que se comercialice ofrece un rápido acceso a los datos, ya que utiliza mecanismos bastante sofisticados de recuperación de la información, que relacionan al RDBMS con el sistema operativo y la máquina que contiene la información. Pero aún así a veces es necesario mejorar la velocidad de respuesta, es entonces cuando el desarrollador de la aplicación y el DBA o administrador de la base de datos deben trabajar conjuntamente para optimizar las formas de recuperación de datos y reducir la velocidad de respuesta. Esto implica configuración correcta del servidor de base de datos, de la propia base de datos y de las consultas (querys) o sentencias de SQL que se realizan.

En este caso se abordará la optimización por parte del desarrollador, que se enfoca en la correcta utilización del lenguaje SQL para mejorar el tiempo de respuesta, así como de otras ventajas que se puedan aprovechar del RDBMS como las funciones y los procedimientos almacenados. Y aunque no es tarea del desarrollador el diseño y configuración de la base de datos es muy importante conocer estos conceptos ya que influyen mucho en el rendimiento de la aplicación. En este trabajo se mencionan características específicas de un manejador en especial que es Oracle, uno de los más reconocidos a nivel mundial, aunque la mayoría de sus especificaciones se encuentran en otros manejadores del mercado.

1.2 SQL

SQL es el lenguaje estándar reconocido para los RDBMS comerciales, su nombre se deriva de *Structured Query Language* (lenguaje estructurado de consulta). Existen otros dos lenguajes menos extendidos pero que también se utilizan en las bases de datos relacionales, estos son: QUEL y QBE (Query By Example. Consulta por ejemplo), que se basan en el cálculo relacional. La principal diferencia entre el cálculo relacional y el álgebra relacional es que en el cálculo relacional se escribe una expresión declarativa para especificar una solicitud de obtención de datos, en tanto que en el álgebra relacional debemos escribir una secuencia de operaciones. SQL, que es específicamente un lenguaje de cuarta generación utilizado para trabajar con bases de datos relacionales y el más difundido en los RDBMS actuales.

Breve historia de SQL

La historia de SQL esta íntimamente ligada con el desarrollo de las bases de datos relacionales. El concepto de bases de datos relacional fue desarrollado originalmente por el Dr. Edgard F. Codd, un investigador de IBM en junio de 1970 publicó un artículo titulado "A Relational Model of Data for Large Shared Data Banks", que esquematizaba una teoría matemática de cómo los datos podían ser almacenados y manipulados utilizando una estructura tabular, de ahí el nombre de tablas. Las bases de datos y SQL tienen sus orígenes en este artículo que apareció en la revista científica de la Association for Computing Machinery (ACM): "*Communications of the ACM*". A continuación se listan los principales acontecimientos del origen de este lenguaje, figura 1.17.

Fecha	Acontecimiento
1970	Codd define el modelo de base de datos relacional.
1974	Comienza el proyecto System/R de IBM.
1974	Primer artículo que describe el lenguaje SEQUEL (SQL).
1978	Test de clientes del System/R.
1979	Oracle introduce el primer RDBMS comercial
1981	Relational Tecnology introduce Ingres.
1981	IBM anuncia SQL/DS.
1986	ANSI forma el comité de estándares SQL.
1997	ISO aprueba uno de los últimos estándares de SQL.

Figura 1.17 Breve historia del lenguaje SQL.

El artículo desencadena una racha de investigaciones en bases de datos relacionales, incluyendo un importante proyecto de investigación dentro de IBM, el objetivo del proyecto, llamado System/R, fue demostrar la operabilidad del concepto relacional y proporcionar alguna experiencia en la implementación efectiva de un DBMS relacional. El trabajo del System/R comenzó a mediados de los setenta en los laboratorios de Santa Teresa de IBM en San José, California, E.U.

En 1974 y 1975 la primera fase del proyecto System/R, produjo un mínimo prototipo de un DBMS relacional. Además del propio DBMS, el proyecto System/R incluía trabajos sobre lenguajes de consulta de bases de datos. Uno de estos lenguajes fue denominado SEQUEL, un acrónimo de Structured English Query Language. Luego en 1978 y 1979 fueron realizadas las primeras instalaciones del System/R y fue también renombrado a SQL, en 1979 el proyecto de investigación llegó a su final, e IBM concluyo que las bases de datos relacionales no solamente eran factibles sino que podrían ser la base de un producto comercial útil. Que posteriormente se comprobaría con creces, con muchos manejadores que conocemos hoy en día, como ORACLE, SYBASE, INFORMIX, SQL Server, PROGRESS, etc. IBM tiene ahora a su manejador DB2.

El lenguaje SQL

No es en sí mismo un sistema manejador de bases de datos, ni un producto autónomo, no se puede comprar SQL, porque es parte central de un sistema manejador de bases de datos, un lenguaje o una herramienta para comunicarse con el RDBMS. Es un lenguaje no procedural que permite al usuario organizar, gestionar y recuperar datos almacenados, en una base de datos; trabaja con un tipo específico de base de datos, llamada base de datos relacional, al que solo hay que decirle que haga y no nos preocupamos tanto por como lo haga, ventaja de los lenguajes de 4° generación Figura 1.18.

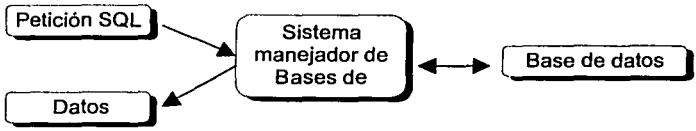


Fig. 1.18. Así funciona SQL con el sistema manejador de bases de datos.

La máquina de base de datos es el corazón del RDBMS, responsable de estructurar, almacenar y recuperar realmente los datos en el disco, junto con el sistema operativo. SQL es un lenguaje fácil de entender para nosotros, con el cual solicitamos a un manejador realice alguna operación sobre nuestros datos, SQL entonces traduce nuestra frase sencilla al lenguaje que entienden el DBMS, y el sistema operativo de la máquina que son los que hacen el trabajo. SQL también acepta peticiones SQL procedentes de otros componentes RDBMS.

Cuando se necesita recuperar datos, se utiliza SQL para efectuar la petición, el RDBMS procesa la petición SQL, recupera los datos solicitados y los devuelve. Este proceso de solicitar datos a la base de datos y de recibir los resultados se llama consulta o "query" a la base de datos. Pero no solo realizar consultas es todo lo que se puede hacer con este lenguaje, ya que es muy poderoso, SQL es mucho más que una herramienta de consulta, ya que se utiliza para controlar todas las funciones que un RDBMS proporciona a sus usuarios.

Durante los últimos años SQL se a convertido en el lenguaje estándar para trabajar con las bases de datos, la mayoría de los productos manejadores de bases de datos soportan SQL (si no, es que todos), y cada uno de ellos contiene el SQL estándar, más agregados que ellos anexan o pequeñas modificaciones que realizan; ejecutándose en sistemas informáticos que van desde computadoras personales a supercomputadoras. Los comandos de SQL se pueden clasificar así:

DML Data Management Language

Las siguientes instrucciones forman parte de este lenguaje de manipulación de datos, es decir sin afectar la estructura original de la tabla sobre la que se trabaja. Las instrucciones son las siguientes:

Para realizar consultas o recuperación de datos:
Seleccionar: SELECT
Altas: INSERT
Bajas: DELETE
Cambios: UPDATE

DDL Data Definition Language

Las siguientes instrucciones se encargan de crear una base de datos, es decir definir la estructura de la misma, tipos de los campos etc., también para crear otro tipo de objetos como vistas.

Crear tablas: CREATE
Modificarlas: ALTER
Borrarlas: DROP

DCL Data Control Language

Seguridad en los datos: Como en una BD debemos tener mucho control de la información que cada usuario tiene, se precisa de que el administrador de la BD genere para cada uno de ellos sus privilegios sobre las tablas. En SQL podemos otorgar privilegios a los usuarios con el comando GRANT.

Tenemos 3 tipos de GRANTS:

- Privilegios del Sistema
- Privilegios de Tablas
- Privilegios de Vistas

Para eliminarlos se cuenta con el comando REVOKE.

Otorgar privilegios: GRANT
Eliminar privilegios: REVOKE

Operadores relacionales de SQL

UNION. El operador de Unión acepta como entrada 2 tablas con las mismas columnas en el mismo orden, y produce como resultado todas las columnas y todos los renglones de ambas tablas. Si existe algún renglón con la misma información en ambas tablas, solo aparece una vez, aunque se pueden mostrar los renglones repetidos especificando la opción: UNION ALL en Oracle.

INTERSECCION. El operador Intersección selecciona de ambas tablas los renglones que tengan exactamente la misma información en todas las columnas.

DIFERENCIA. El operador de diferencia acepta como entrada dos tablas que tengan al menos una columna, en donde el resultado será los registros de la primera tabla que no estén en la segunda.

Por tanto SQL es un lenguaje completo de control e interacción con un sistema manejador de bases de datos. Pero no es un lenguaje informático completo tal como "C", "Pascal", no dispone de la sentencia "if" para examinar condiciones, ni la sentencia "GOTO" para bifurcaciones, ni de las sentencias "DO" o "FOR" para iteraciones, en vez de ello es un sublenguaje de bases de datos, consistente en unas 30 sentencias especializadas para tareas de gestión de base de datos. Estas sentencias se "incorporan" a otro lenguaje, como "C", Visual Basic, Delphi, Power Builder, entre otros, para extender sus posibilidades y permitirle utilizar el acceso a las bases de datos. Finalmente, no es un lenguaje particularmente estructurado, especialmente cuando se compara con lenguajes altamente estructurados como "C". En vez de ello, las sentencias SQL se asemejan a frases en inglés, completadas con palabras de relleno que no añaden nada al significado de la frase pero que hace que se lea más naturalmente.

Transacciones

Una transacción es una unidad lógica de trabajo que comprende una o más sentencias SQL ejecutadas por un solo usuario entre un "*commit*" (sentencia SQL) y otro. De acuerdo con el estándar ANSI/ISO, la transacción inicia con la primer sentencia SQL ejecutable, una transacción se finaliza cuando le es explícitamente indicada con la instrucción: *Commit*; o es anulada con la instrucción: *Rollback*.

Para entender mejor este concepto, se muestra el siguiente ejemplo: considere una base de datos bancaria, cuando un cliente transfiere dinero desde una cuenta de ahorros a una cuenta de cheques, la transacción podría consistir de tres operaciones separadas: disminuir la cuenta de ahorros, aumentar la cuenta de cheques, y registrar la transacción en el diario de transacciones. Se debe garantizar el desempeño correcto de las sentencias SQL para mantener las cuentas en un balance apropiado. Cuando algo impide que se lleve acabo alguna de las sentencias (por ejemplo, una falla del hardware), las otras sentencias de la transacción no se deben realizar, esto es llamado "rollback". Si no ocurre ningún error en cada una de las actualizaciones, los cambios son realizados. El siguiente ejemplo ilustra esta transacción, figura 1.19.

	Inicio de la transacción.
`UPDATE cuentas_ahorro SET balance = balance - 500 WHERE cuenta = 12345;`	Decremento a la cuenta de ahorros.
`UPDATE cuentas_ahorro SET balance = balance + 500 WHERE cuenta = 12346;`	Incremento a la cuenta de cheques.
`INSERT INTO jornada VALUES (jornada.seq_NEXTVAL, '1B',12345,12346,500);`	Guardar en el registro de transacciones.
`COMMIT WORK;`	Fin de la transacción.

Figura 1.19. Una transacción bancaria.

En este caso la transacción fue aceptada, pero si la transacción fuera rechazada con un *rollback*, los datos no sufrirían ningún cambio en ninguna de las tres sentencias. Para que hubiera cambios parciales, es necesario hacer un *commit*, después de cada sentencia.

Aceptando o cancelando transacciones

Los cambios hechos por las sentencias SQL que constituyen una transacción pueden ser aceptados o cancelados. Después que una transacción es aceptada o cancelada, la siguiente transacción comienza con la próxima sentencia SQL.

Aceptar la transacción hace permanente los cambios resultantes de todas las sentencias SQL de la transacción. Los cambios hechos por las sentencias SQL de una transacción son visibles a otro usuario solo después que la transacción es aceptada (COMMIT).

Cancelar una transacción retracta todos los cambios resultantes de las sentencias SQL en la transacción (ROLLBACK). Después que una transacción es cancelada, los datos afectados permanecen sin cambios como si las sentencias SQL de la transacción nunca hubieran sido ejecutadas.

1.3 Arquitectura Cliente Servidor

Los sistemas de cómputo centralizados, como resultado natural de la madurez de la tecnología de redes, han evolucionado a un esquema (propuesto por los proveedores) de descomposición funcional en dos plataformas de hardware, denominadas cliente y servidor. Estas interactúan de manera cooperativa según un modelo simple: en la máquina cliente se genera y envía una petición de servicio a la máquina servidora, en la cual se procesa la petición y se produce una respuesta que se remite al cliente en cuestión. Este modelo de interacción básico es válido sin importar la estrategia de conectividad (medio de transmisión, mecanismo de acceso, dispositivos para interconexión, etc.) considerada en los niveles inferiores del sistema.

Por conveniencia práctica, la plataforma cliente puede ser PC o estación de trabajo RISC, con una adecuada interfaz de usuario (por ejemplo, Windows, Open Look, Motif o Xwindow). De manera complementaria, la plataforma servidora puede concretarse mediante PCs, estaciones de trabajo RISC, minicomputadoras o inclusive mainframes, con la posibilidad de albergar una diversidad de servicios (archivos, impresión, base de datos, conocimiento, etc.).

Pues bien, este modelo simple de petición-respuesta se ha convertido en un paradigma y ha sido implantado por múltiples proveedores de diversos productos mediante arquitecturas específicas, inclusive en su desarrollo han empleado cierta tecnología de valor agregado (por ejemplo, servidor multitarea). De esta manera, cada fabricante integra y ofrece sus muy particulares sistemas cliente/servidor.

Es importante señalar que en el modelo básico cliente/servidor solamente se define al cliente como generador y emisor de peticiones, y al servidor como productor de las respuestas correspondientes. No se especifica si el mecanismo de comunicación entre ellos es por paso de mensaje o por invocación de procedimiento remoto (RPC: Remote Procedure Call). Tampoco se indica si la iniciativa de la interacción puede ser en ambos sentidos: que el servidor genere peticiones y el cliente produzca respuestas, o inclusive que haya interacciones entre los servidores. Es más, la petición en sí es una orden que generalmente carece de la flexibilidad necesaria para transportar cierto volumen de información para el servidor.

Como el modelo básico no corresponde a estos cuestionamientos, cada fabricante ha hecho las extensiones avanzadas pertinentes, lo que significa entonces que en el mercado existen diversas implantaciones que concretan varios modelos cliente/servidor con mayor o menor abstracción. Por otro lado, para que opere una arquitectura cliente/servidor se requiere de una red local o global como infraestructura física. Esta funcionalidad permite afirmar que un sistema cliente/servidor en realidad es un sistema distribuido particular. De hecho constituye el primer paso en la transición hacia los sistemas distribuidos.

Una buena idea consiste en tomar el modelo cliente/servidor como punto de partida y extenderlo paulatinamente hasta conformar sistemas distribuidos complejos. Este enfoque corresponde sobre todo a algunos proveedores de sistemas cliente/servidor de base de datos. En síntesis, los sistemas cliente/servidor ya constituyen una tecnología madura que puede emplearse para resolver los problemas de compartir recursos o de implantación de sistemas de información distribuidos. He aquí algunas razones para su uso:

- Existe un gran mercado actual y potencial
- Es fácil de entender y de poner en práctica
- La implantación del servidor es relativamente fácil, pues en realidad es una extrapolación a red de un sistema centralizado
- Existen herramientas de programación adecuadas (sockets, APIs de alto nivel, etcétera.)
- Constituye una fase intermedia para soluciones cooperativas complejas

Este concepto implica que la aplicación de la base de datos y la base de datos se encuentran separadas en dos partes: "front-end" o cliente y "back-end" o servidor. El cliente ejecuta la aplicación de la base de datos que accesa al servidor para obtener la información e interactúa con el usuario a través del teclado, la pantalla y otros dispositivos como el mouse. El servidor ejecuta el software del manejador (RDBMS) y maneja las funciones requeridas por concurrencia, acceso compartido, etc. a la Base de Datos. Dentro de este esquema se pueden dar muchas variantes, para Oracle por ejemplo, la arquitectura cliente servidor es flexible en cuanto a la ejecución de los procesos, es decir se puede dividir el trabajo de un proceso en más de una tarea para aumentar la velocidad de respuesta, este concepto se conoce como procesamiento distribuido. Este concepto se explica a continuación.

Procesamiento distribuido

Este tipo de procesamiento ocurre cuando se utiliza más de un procesador para dividir un proceso en tareas individuales. Ejemplos de procesos distribuidos:

A. El cliente y el servidor se encuentran en diferentes computadoras; las computadoras están conectadas mediante una RED, ver figura 1.20 (a).
B. Una sola computadora, y diferentes procesadores mantienen la ejecución de la aplicación (cliente), ver figura 1.20 (b).

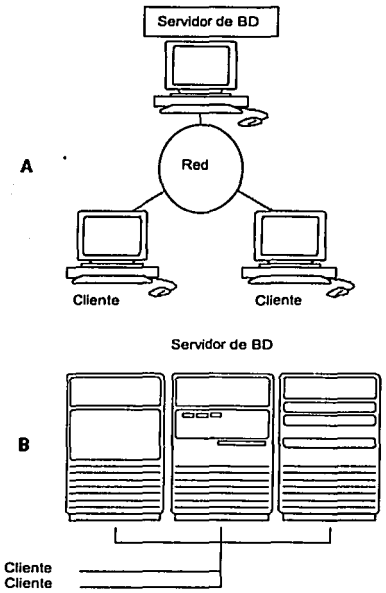


Fig. 1.20. Esquemas en la arquitectura cliente servidor.

Beneficios de la arquitectura Cliente/Servidor en un ambiente de proceso distribuido:

- Las aplicaciones cliente no son responsables del performans del proceso de los datos. Las aplicaciones cliente sólo se concentran en hacer formularios de requrimientos de información para los usuarios, solicitar información del servidor, el análisis y la presentaicón de estos datos usando las capacidades de visualización de la estación de trabajo del cliente o la terminal. Por ejemplo utilizando gráficas o reportes.
- Las aplicaciones del cliente pueden ser diseñadas sin depender de una localización fija de la base de datos. Si los datos son movidos o distribuidos a otros servidores de base de datos, la aplicación sigue funcionando sin cambios o con pequeñas modificaciones.
- Son explotadas las capacidades de multitatarea y memoria compartida que fácilitan los sistemas operativos, esto da como resultado un alto desempeño en tareas como concurrencia, integridad de los datos y performance de las aplicaciones del cliente.
- Las estaciones de trabajo o términales del cliente pueden ser optimizadas para la presentación de datos (por ejemplo, por agregar gráficos y soporte para el mouse) y el servidor puede ser optimizado para el proceso y almacenamiento de los datos (por ejemplo, agregando grandes cantidades de memoria y espacio en disco)
- Si es necesario, el RDBMS puede ser escalado. Si los sistemas crecen, se pueden agregar varios servidores para distribuir el proceso de la base de datos mediante la red (escalado horizontal). Otra alternativa es cambiar el RDBMS a una computadora más poderosa, como una microcomputadora o mainframe. Cuando el RDBMS corre en una máquina de estas características toma ventaja del gran performans del sistema (escalado vertical). En ambos casos, todos los datos y aplicaciones se mantienen sin camios o con pequenas modificaciones.
- En ambientes de red, los datos compartidos son almacenados en los servidores y no en todas las computadoras del sistema. Esto hace un fácil y mas eficiente manejo del acceso concurrente.
- En ambientes de red, sencillos, las estaciones de trabajo pueden ser usadas para accesar los datos remotos del servidor eficientemente.
- En ambientes de red, las aplicaciones del cliente hacen peticiones de información al servidor usando sentencias SQL, una vez recibida, la sentencia SQL es procesada por el servidor, y los resultados son devueltos a la aplicación cliente mediante la red, el tráfico de la red mantiene guardada la respuesta un tiempo mínimo portque únicamente las peticiones y los resultados viajan en la red.
- Una aplicación cliente/servidor pone en comunicación una estación de trabajo con un servidor de base de datos central. Con la disponibilidad generalizada de redes y estaciones de trabajo baratas, las aplicaciones cliente/servidor se han hecho cada vez más populares.
- Una base de datos puede servir como repositorio central de informaciones corporativas, por ejemplo, una base de datos relacional podrá mantener y estructurar todos los datos contables y de ventas para un negocio o todos los datos de investigación de una compañía de explotación de petróleo.
- Mucha gente en una corporación puede desear acceder a tal base de datos, el departamento de contabilidad, el departamento de ventas y el de distribución; podrían todos querer acceder a partes de los datos contables o de ventas.

Capítulo II Conceptos de Optimización

2.1 ¿Qué es la optimización?

La optimización es el proceso de hacer que la aplicación responda de manera más rápida a las peticiones del usuario y que esto implique menos trabajo para el desarrollador y el RDBMS. Por ejemplo, optimizar una sentencia SQL, significa transformar la sentencia original en sentencias alternativas que seleccionen el camino más eficiente para su ejecución, además de alterar la base de datos si es necesario (como al agregar algún índice), esto es un paso importante en el proceso de cualquier sentencia del lenguaje de manipulación de datos (SELECT, INSERT, UPDATE o DELETE). Oracle tiene diferentes caminos para realizar las sentencias, cambiando por ejemplo, cuales tablas o índices se deben accesar y en que orden. El procedimiento utilizado para ejecutar una sentencia puede ser modificado para hacer más rápida su ejecución. Una parte de Oracle llamada optimizador elige el camino que cree será el más eficiente.

El optimizador considera un número de factores que hacen que usualmente encuentre la mejor elección entre varias alternativas. Sin embargo, el diseñador de la aplicación conoce más acerca de un dato particular que lo que el optimizador pueda conocer. A pesar de los mejores esfuerzos del optimizador, en algunas situaciones el desarrollador puede escoger un camino más efectivo para ejecutar una sentencia SQL que el optimizador. El optimizador de Oracle puede tomar diferentes decisiones para una versión y otra. En futuras versiones de Oracle el optimizador puede hacer diferentes decisiones basadas en mejor y más información.

La optimización es una parte del ciclo de vida de toda aplicación de base de datos (en algunos casos debería serlo). La mayoría de los problemas de rendimiento no son problemas aislados, sino resultado del diseño del sistema, por tanto las tareas de optimización se deberían centrar en identificar y solucionar los defectos de la base de datos que dan lugar a un mal rendimiento. La optimización es el último de un proceso de cuatro pasos, que va precedido por los pasos de planificación, desarrollo y supervisión. Si se realiza la optimización sin tener ningún propósito en concreto, lo más probable es que no se resuelvan los defectos que fueron el origen del problema de rendimiento, esto significa que se deben analizar perfectamente las causas del bajo rendimiento y sus posibles soluciones, porque existen muchas maneras de resolverlo, quizás la más optima sea alguna combinación de alternativas, como ha sucedido en la práctica. Para citar un ejemplo, una correcta optimización del espacio en la base de datos, una sintaxis alternativa, el empleo de procedimientos almacenados y el uso de algunas tablas desnormalizadas; entre otras soluciones en conjunto, hacen que el rendimiento mejore sustancialmente.

Además es muy importante conocer cómo maneja las áreas de memoria, cada una de estas áreas debe ser lo suficientemente grande como para contener los datos que se solicitan con más frecuencia a la base de datos. Si están dimensionadas correctamente, las áreas de memoria pueden mejorar de manera sustancial el rendimiento de las consultas individuales y de la base de datos en su conjunto.

2.2 Métodos de Optimización

La metodología es la llave del éxito en la optimización del rendimiento. Diferentes estrategias ofrecen rendimiento decreciente, es importante usar estrategias que obtengan el máximo de ganancia, además, sistemas con diferentes propósitos, como sistemas de transacciones en línea y sistemas de apoyo en la toma de decisiones, pueden requerir diferentes

necesidades. ¿Cuándo es más efectiva la optimización? Los mejores resultados ocurren si la optimización se realiza durante la fase de diseño y hasta antes de implementar el sistema.

- Optimización proactiva mientras se diseña y desarrolla el sistema.
- Optimización reactiva para mejorar sistemas de producción.

Optimización proactiva mientras se diseña y desarrolla el sistema

Por mucho la forma mas efectiva de optimizar es trabajando proactivamente. Para iniciar se deben establecer los objetivos justificables de rendimiento y las expectativas realistas entre el diseñador de la aplicación y los usuarios finales. Durante el diseño y el desarrollo, los diseñadores de la aplicación pueden determinar la combinación de recursos del sistema y las características disponibles del RDBMS que mejor cumplan estas necesidades. Si el diseño del sistema esta bien, se minimizan costos y frustraciones. La figura 2.1 ilustra el costo relativo de la optimización durante la vida de una aplicación.

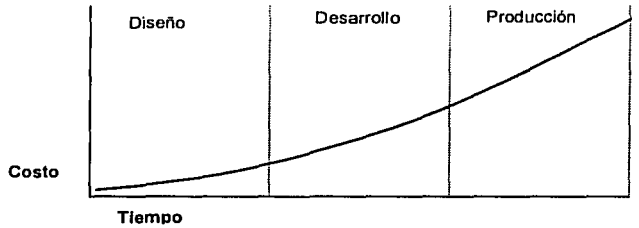


Figura 2.1 Costo de optimizar durante la vida de la aplicación.

Como complemento a la gráfica anterior, en la siguiente (figura 2.2) se muestra el beneficio de optimizar sobre el ciclo de vida del sistema, el beneficio es inversamente proporcional al costo.

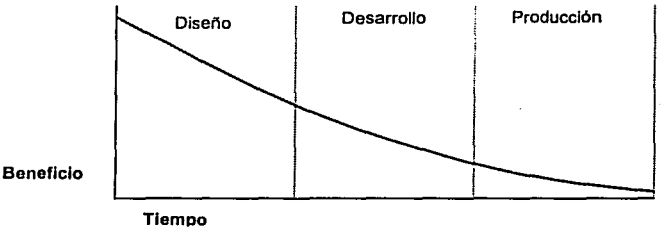


Fig. 2.2 Beneficio de optimizar durante la vida de la aplicación.

Como se puede ver, el tiempo más efectivo para optimizar es durante la fase de diseño, se obtiene así el máximo beneficio a un menor costo.

Optimización reactiva para mejorar sistemas en producción

Desafortunadamente mucha gente cree que el proceso de optimización empieza cuando los usuarios se quejan por el tiempo de respuesta, es decir que los procesos se tardan demasiado y es peor cuando se incrementa el volumen de la información y el número de usuarios que accedan el sistema. Si el tiempo normal de respuesta para un reporte es de menos de 5 minutos, y para un proceso de aproximadamente de media hora a una hora, aunque para cada sistema los procesos y la cantidad de información son distintas, por sentido común, no es de buen gusto esperar por mucho tiempo en la pantalla un resultado. Si ocurren problemas de rendimiento y no es posible rediseñar la aplicación, sólo se puede mejorar de manera parcial. Configurando adecuadamente el RDBMS, cambiando las formas de acceso a la información de las consultas, agregando más potencia al servidor, optimizando el tráfico en la red, etc., aun así la optimización es posible, pero como es lógico es más costoso que si desde el inicio se tomaran en cuenta todas las medidas necesarias para alcanzar el nivel más óptimo.

2.3 Prioridad en los pasos de la optimización

Una vez definido el sistema y establecidos los niveles de rendimiento es recomendable seguir un orden para alcanzar una optimización adecuada. Cada etapa significa una forma de ayudar a lograr la optimización total, se desarrolla de forma distinta y con sus propias características y métodos. Los siguientes pasos son una forma correcta de alcanzar un rendimiento óptimo en un sistema de información:

1. Reglas del negocio
2. Diseño de los datos
3. Diseño de la aplicación
4. Estructura lógica de la base de datos
5. SQL
6. Rutas de acceso
7. Memoria
8. I/O* y estructuras físicas
9. Recursos compartidos
10. Sistema operativo

Esta forma de optimizar, funciona aún mejor si se realiza de forma iterativa, es decir, si se revisan las etapas varias veces hasta asegurarse de que se ha alcanzado el nivel deseado. Siempre recorriendo todas las etapas y en orden, ya que las etapas iniciales son en las que más se puede intervenir, hasta las últimas donde no depende completamente de quién desarrolla la aplicación. Las decisiones tomadas en cada etapa influyen en las siguientes. Por ejemplo: si en el paso 5 se re escriben algunas sentencias SQL, estas sentencias SQL pueden tener efecto en las rutas de acceso (paso 6) y la memoria (paso 7). Las optimización de las operaciones de I/O en el disco (paso 8) dependen del tamaño de la memoria cache (paso 7).

Aunque en el presente trabajo se enfoca en la optimización del SQL y la base de datos, es importante mencionar que la optimización total depende de muchas cosas más, y de muchas personas especializadas en cada área para optimizar mejor en su campo de acción. Enseguida se describen brevemente los pasos del ciclo de la optimización:

Paso 1. Optimizar las reglas del negocio

Para el desempeño óptimo, se tienen que adaptar las reglas de negocio. Estos conciernen el diseño y análisis de alto nivel de un sistema entero. Los puntos de configuración se consideran a este nivel, para determinar si se utiliza un servidor multitareas para todo el sistema

o no, de esta manera, los diseñadores aseguran que los requerimientos de desempeño del sistema corresponden directamente a necesidades concretas de negocio.

Los problemas de desempeño encontrados por el DBA pueden realmente ser ocasionados por problemas en el diseño e implementación, o por reglas impropias de negocio. La gente usualmente, va demasiado profundo, cuando ellos escriben las funciones de negocio de una aplicación, documentan una implementación, más que simplemente la función que debe desempeñarse. Si los ejecutivos de negocio tienen cuidado en abstraer el requerimiento o función de negocio desde la implementación, entonces los diseñadores tienen un campo más amplio para escoger la implementación apropiada.

Considérese, por ejemplo, la función de negocio de imprimir un cheque, el requerimiento real está en pagar dinero a la gente; el requerimiento no es necesariamente para imprimir pedazos de papel (cheques). Considerando sería muy difícil imprimir unos millones de cheques por día, sería relativamente fácil de registrar muchos pagos directos de depósito sobre una cinta, que puede enviarse al banco para procesar.

Las reglas de negocio deberían ser uniformes con expectativas realistas para el número de usuarios concurrentes, el tiempo de respuesta de transacción, y el número de registros almacenados que el sistema puede apoyar. Por ejemplo, no tendría sentido correr una aplicación altamente interactiva sobre la red lenta de área amplia sobre una línea telefónica.

Paso 2. Optimizar el diseño de los datos

En la fase de diseño de datos, se debe determinar qué datos son necesarios por sus aplicaciones. Es necesario considerar qué relaciones y atributos son importantes, finalmente se necesita estructurar la información para encontrar mejores metas de desempeño.

El proceso de diseño de la base de datos generalmente experimenta una etapa de normalización, en el que los datos se analizan para asegurar que no haya ningún dato redundante, un dato debería ser único en la base de datos. Una vez que los datos se normalizan cuidadosamente, sin embargo, se puede necesitar desnormalizar por razones de desempeño, se podría, por ejemplo, decidir que la base de datos debería contener valores resumen o calculados. Más que forzando a una aplicación a recalcular el precio total de todas las líneas en una orden determinada cada vez que se accese, se podría decidir incluir el valor total de cada orden en la base de datos, se podría establecer llave primaria, índice y llaves foráneas para accesar esta información rápidamente.

Otra consideración en el diseño de los datos es la prevención de la contienda sobre los datos. Considere una base de datos de 1 terabyte de tamaño, sobre la que unos mil usuarios accesan únicamente el 0.5% de los datos, este "detalle" en los datos podría ocasionar problemas de desempeño.

Paso 3. Optimizar el diseño de la aplicación

Los diseñadores de la aplicación y ejecutivos de negocio necesitan traducir las metas del negocio en un diseño efectivo del sistema.

Un ejemplo de diseño inteligente de procesos, es estratégicamente captar los datos. Por ejemplo, en una aplicación detallista se puede seleccionar la taza de descuento para algunos productos una vez al principio de cada día e introducirlo dentro de la aplicación, de esta manera se evita solicitar la misma información una y otra vez durante el curso del día.

A este nivel también, se puede considerar la configuración de procesos individuales, por ejemplo, algunos usuarios de computadoras personales pueden accesar el sistema central usando agentes móviles, considerando los otros usuarios que pueden conectarse directamente, aunque que ellos corran sobre el mismo sistema, la arquitectura es diferente. Ellos pueden requerir también servidores diferentes de correo, versiones diferentes de la aplicación etc.

Paso 4. Optimizar el diseño de la base de datos

Después que la aplicación y el sistema se han diseñado, se puede planificar la estructura lógica de la base de datos, esto implica la optimización efectiva de los índices. En la etapa de diseño de los datos se determinan las llaves primarias y foráneas, en la etapa del diseño de las estructuras lógicas se pueden crear índices adicionales para apoyar la aplicación.

Los problemas de desempeño debido a la contienda frecuentemente involucran inserciones en el mismo bloque de datos (es decir, en la misma tabla), se debe tener cuidado especial en diseñar el uso y ubicación de los índices, generadores de sucesión (secuencias), y clusters (grupos).

Paso 5. Optimizar la estructura lógica de la base de datos

Los diseñadores de sistemas y los desarrolladores de aplicaciones, deben comprender el mecanismo de procesamiento de consultas del RDBMS, para escribir sentencias SQL efectivas. En el capítulo 4, se muestran más a detalle la forma de escribir sentencias que logran los resultados más rápidos.

Antes de afinar el RDBMS, se debe estar seguro que la aplicación tome todas las ventajas del lenguaje SQL y las características del RDBMS diseñadas para hacer más rápido los procesos de la aplicación. Se deben utilizar aspectos y técnicas como las siguientes, con base en las necesidades de su aplicación:

- procesamiento en paralelo
- el optimizador del RDBMS (En este caso Oracle)
- PL/SQL

Paso 6. Optimizar las rutas de acceso

Asegurarse del acceso eficiente a los datos. Considerar el uso de clusters, hash clusters, índice B-tree ("normales") e índices bitmap (que son objetos de la base de datos asociados a una tabla, para mejorar su rendimiento, se explicarán más a detalle en el siguiente capítulo). Asegurando el acceso eficiente, puede significar reconsiderar su diseño después que se tiene construida la base de datos, puede requerirse hacer más normalización (ajuste) o crear índices alternativos en este punto. Al probar la aplicación si no se puede encontrar todavía con el tiempo de respuesta requerido, entonces se debe buscar otras opciones para mejorar el desempeño.

Paso 7. Optimizar la memoria

La distribución apropiada de recursos de memoria, a las estructuras de memoria de Oracle (RDBMS), puede tener un gran impacto sobre el desempeño. Oracle comparte la memoria y la destina dinámicamente a las estructuras siguientes, que son todas las partes de la memoria compartida, aunque que se coloque explícitamente la cantidad total de memoria disponible en la memoria compartida, el sistema dinámicamente coloca el tamaño de cada estructura:

- el cache de diccionario de datos
- el cache de biblioteca
- las áreas de contexto (si se corre un servidor multitarea)

Se puede colocar explícitamente distribución de memoria para las estructuras siguientes:

- buffer de cache
- buffer de registro
- los caches de secuencia

La distribución apropiada de los recursos de memoria puede mejorar el desempeño del cache, reduce el análisis de las sentencias SQL y el intercambio de información entre el disco y la memoria.

El proceso de áreas locales incluye:

- las áreas de contexto (para sistemas que no corren servidor multitarea)
- áreas de ordenación
- áreas hash

Se debe cuidar no destinar al SGA (Sistem Global Area, que es el segmento de memoria del RDBMS de Oracle que contiene datos e información de control de la base de datos) un porcentaje grande de la memoria física de la máquina ya que ocasiona paginación y más intercambio de información entre el disco y la memoria, y por lo tanto más tiempo de espera para terminar las operaciones.

Paso 8. Optimizar la estructura lógica de la base de datos

Las operaciones de I/O tienden a reducir el desempeño de muchas aplicaciones de software. El servidor de Oracle, sin embargo, esta diseñado para que su desempeño no necesite ser excesivamente limitado por las operaciones de I/O. Afinar el I/O y las estructuras físicas involucra estos procedimientos:

- distribuir datos para que el I/O se distribuya, y así evitar la contienda de los discos.
- almacenar datos en bloques de datos correctamente dimensionados para un mejor acceso: valores apropiados para PCTFREE y PCTUSED, que se definen ampliamente en el capítulo siguiente y que a grandes rasgos son parámetros de almacenamiento propios de las tablas o índices, que indican el porcentaje de espacio utilizado y libre, respectivamente.
- crear los extents (extensiones) grandes y suficientes para los datos, para evitar la extensión dinámica de tablas.
- evaluar el uso de dispositivos de almacenamiento especiales

Paso 9. Optimizar los recursos compartidos

El procesamiento concurrente por múltiples usuarios de la aplicación, puede crear que los procesos se hagan más lentos por que los recursos del RDBMS atenderán las peticiones de los usuarios conforme sus recursos lo permitan y conforme se vallan liberando, esto sucede si se solicitan los mismos recursos para más de un usuario al mismo tiempo, se atenderá al primero que realice su petición y a continuación al segundo y así sucesivamente. La contienda puede ocasionar que los procesos esperen hasta que los recursos estén disponibles. Se debe cuidar reducir los tipos siguientes de contienda:

- contienda de datos
- contienda la memoria compartida
- contienda de bloqueo

Paso 10. Optimizar el sistema Operativo

Analizar la plataforma específica y la documentación del RDBMS para investigar maneras de afinar el sistema operativo en cuestión. Por ejemplo, sobre sistemas basados en UNIX se podría querer optimizar lo siguiente:

- el tamaño del buffer del cache de UNIX
- los manejadores lógicos de volumen
- la memoria y tamaño para cada proceso

2.4 Como aplicar el método de optimización

Es necesario saber como aplicar el método de optimización para obtener mejores resultados, los siguientes puntos son importantes para su realización:

Tener objetivos claros de optimización

Nunca se debe comenzar la optimización sin tener establecidos objetivos claros, no se puede triunfar si no hay definición de "éxito".

"Simplemente hacer las cosas tan rápido como pueda", suena como un objetivo, pero será muy difícil determinar si esto se ha logrado. Será aun más difícil saber si los resultados satisfacen los requerimientos subyacentes al negocio, una declaración más útil de objetivos es el siguiente: "Nosotros necesitamos tener 20 operadores realizando 20 ordenes por hora, y las listas de empaque en menos de 30 minutos."; es decir, ubicar un proceso y tener en concreto su resultado para optimizarlo.

Se deben guardar las metas que se consideren como medidas "afinadoras"; y considerar el beneficio del desempeño al alcanzarlas. Se debe considerar que las metas pueden entrar en conflicto, por ejemplo, para lograr un mejor desempeño en una sentencia SQL especifica, es posible que se sacrifique el desempeño de otra sentencia SQL que corre concurrentemente en la misma base de datos y que si se afectan las estructuras físicas y lógicas de esta, probablemente se afectará el rendimiento de otras, por lo tanto debemos decidir que sentencias tienen mayor prioridad y enfocarnos en ellas.

Crear pruebas de verificación

Crear series de pruebas de casos reproducibles, por ejemplo, si se identifica una sentencia SQL que ocasiona un problema de desempeño, se intenta corregir y entonces se deben correr ambas sentencias: la original y la versión "corregida" para poder ver la diferencia en el desempeño. En muchos casos, un esfuerzo al afinar puede triunfar simplemente por identificar una sentencia SQL que ocasionaba el problema de desempeño.

Si la sentencia que se va a probar dura más de una hora, dos, tres o más, no es práctico realizar pruebas con la sentencia original, se recomienda que se agregue una condición restrictiva, digamos para un departamento en lugar de 500, si este fuera el caso, con el objetivo de esperar por menos tiempo en las pruebas de afinamiento, hasta alcanzar la sentencia optima, entonces ya vale la pena probar con la sentencia completa, de esta manera nos ahorramos mucho tiempo, quizás algunas pruebas podrían consumir más tiempo del que tomaba la

sentencia original y no tenemos que llegar a ese extremo si utilizamos una condición restrictiva con la que nos podemos dar cuenta si esto va a suceder.

Probar Hipótesis

Con un mínimo de pruebas repetibles establecidas, se obtienen resultados que se resumen y analizan, con los que se pueden probar diversas hipótesis y ver el efecto, esto es importante, cada sistema tiene condiciones distintas en las que una sentencia puede ser óptima para ciertas condiciones y no serlo para otras, se deben probar varias alternativas antes de decir que se a encontrado la correcta.

Se debe tener en cuenta que al probar una sentencia SQL, el manejador coloca en la memoria los datos que intervienen en esta, por lo tanto la segunda instrucción que tenga acceso a la misma información va a tener la ventaja de que los datos ya están en la memoria, por lo tanto es mejor tomar en cuenta los tiempos de respuesta de las sentencias después de probar al menos una vez cada sentencia, para que ambas sentencias estén en las mismas condiciones, de datos en memoria y que ambas sentencias ya han sido analizadas por el manejador.

Guardar resultados

Es recomendable guardar el registro de los efectos de cada cambio, así como automatizar las pruebas. De esta manera se alcanzarán resultados de manera más rápida. También se tendrá un registro de las sentencias probadas y sus resultados, esto es muy útil para afinar otras sentencias parecidas y no tener que iniciar desde cero.

Sería de gran utilidad crear un registro de sentencias problemáticas y sus alternativas para su utilización en distintos procesos o sistemas.

Errores comunes

Un error común hecho por los afinadores novatos está en seguir las nociones preconcebidas sobre qué puede ocasionar el problema y dar la solución establecida, otro error común está en tratar diversos enfoques al azar.

Para alcanzar mejores resultados se debe construir un equipo de gente para resolver problemas de desempeño. Mientras un afinador de desempeño puede afinar las sentencias SQL sin conocer la aplicación en forma detallada, el equipo debería incluir alguien que comprenda la aplicación y quien puede validar las soluciones que el afinador de SQL pueda idear.

Evitar Riesgos

Cuidar hacer cosas a la ligera. A veces se crean hipótesis, y se quieren implementar globalmente a lo largo del sistema y entonces esperar para ver los resultados. Se puede arruinar un buen sistema de esta manera!.

Evitar Preconcepciones

Evitar costumbres preestablecidas cuando se trata de afinar. Es importante que los mismos usuarios sean quienes informen del problema aunque no sepan porque existe. Se deben tomar en cuenta los síntomas, más no lo que el usuario crea que es.

Parar cuando los objetivos han sido alcanzados

Una de las grandes ventajas de tener objetivos para optimizar es que es posible definir su éxito, al pasar el punto establecido, no es eficiente continuar afinando el sistema.

Demostrar los alcances de los objetivos

El desarrollador de la aplicación puede estar seguro que los objetivos de desempeño se han alcanzado, pero sin embargo debe demostrar esto a dos grupos importantes de gente:

- los usuarios afectados por el problema
- los responsables del éxito de la aplicación

Siguiendo estos pasos, seguramente se alcanzará un rendimiento óptimo para todas las aplicaciones que se implementen en un RDBMS porque se toman en cuenta los factores más importantes que se influyen en el desarrollo de la aplicación desde su concepción hasta su puesta en producción. En este caso se hace referencia a la forma en que Oracle optimiza las sentencias SQL que integran una aplicación, y para entenderlo es necesario saber como es que se ejecutan las sentencias SQL y que elementos intervienen en este proceso, como la memoria, que es uno de los más importantes.

2.5 Optimizador del RDBMS de Oracle

Los manejadores de bases de datos tienen una forma preestablecida de ejecutar las sentencias SQL, por lo generar las realizan de una forma correcta, pero no siempre esa forma correcta es la mejor. Por ejemplo, Oracle optimiza las sentencias SQL mediante un *optimizador*, que sigue sus reglas preestablecidas y ejecuta la sentencia con el plan de ejecución que considera correcto, el optimizador hace lo siguiente:

- Evaluación de expresiones y condiciones. El optimizador primero evalúa expresiones y condiciones que contienen constantes completamente, hasta donde sea posible.
- Transformación de la sentencia. Para una sentencia compleja, por ejemplo, una que contenga subconsultas, el optimizador transforma la sentencia original en una sentencia join equivalente.
- Vistas combinadas. Para una sentencia SQL que acceda a una vista (las vistas no contienen o almacenan datos físicamente, es decir no ocupan espacio, mantienen temporalmente los datos sólo mientras se utilizan.), el optimizador muchas veces comparte la sentencia con la de la vista y optimiza el resultado.
- Selección de la técnica de Optimización. El optimizador selecciona entre las técnicas de base-regla o base-costo
- Selección de la ruta de acceso. Por cada tabla accesada por el sistema, el optimizador selecciona una o más rutas de acceso para obtener los datos de la tabla.
- Selección del orden de los joins. Para una sentencia ligada con más de una tabla, el optimizador selecciona cual par de tablas serán ligadas primero, y entonces cual es ligada con el resultado.
- Selección de las operaciones de los joins. Para cualquier sentencia join, el optimizador selecciona una operación para ejecutar el join.

El optimizador de Oracle realiza lo siguiente:

Transformar expresiones, en operaciones menos complejas o más fáciles de ejecutar por el manejador:

```
sal > 24000/12                          --> sal >2000
nombre like 'JUAN'                      -->  nombre = 'JUAN'
nombre IN ('JUAN','PEPE')               --> nombre = 'JUAN' OR nombre = 'PEPE'
salario > ANY (100000,150000)           --> salario > 100000 or salario > 150000
ANY y SOME                              --> EXISTS
ALL                                     --> NOT EXISTS
salario BETWEEN 100000 and 150000       -->  salario >= 100000 and salario <= 150000
```

También realiza transformaciones de consultas:

```
OR                                          -->  UNION ALL
(Se relizará si gracias a ello puede utilizar índices en las 2 consultas resultantes)

SELECT * FROM emp
WHERE job = 'CLERK' OR deptno = 10;

SELECT * FROM emp
WHERE deptno = 10 AND job <> 'CLERK'
 Union all
SELECT * FROM emp
WHERE job = 'CLERK' AND deptno <> 10;
```

Sentencias complejas en Joins:

```
SELECT * FROM accounts
WHERE custno IN (SELECT custno FROM customers);

SELECT account.* FROM accounts, customers
WHERE accounts.custno = customers.custno;
```

Sentencias que acceden a vistas:

Para optimizar sentencias que acceden a vistas, el optimizador elige entre 2 alternativas:

- Transforma la sentencia de forma que mezcla el SELECT de la vista con la consulta.
- Ejecuta la vista y se accede a las filas como si fuera una tabla.

Estas reglas pueden cambiarse, indicando en la sentencia SQL la forma en que deseamos que se realice, esto se analizará en el capítulo 4. Todas las sentencias son analizadas y una vez resueltas, es decir, con un plan de ejecución establecido se colocan en memoria para no volverse a analizar y se gana el tiempo que necesita el manejador para realizar el plan de ejecución de la sentencia. Esta área es llamada área compartida.

El área de memoria compartida de SQL

El área global del sistema (SGA) es un segmento de memoria asignado a Oracle que contiene datos e información de control para una determinada instancia de Oracle, una instancia contiene una base de datos figura 2.3, que es accesada por las aplicaciones que generan procesos independientes.

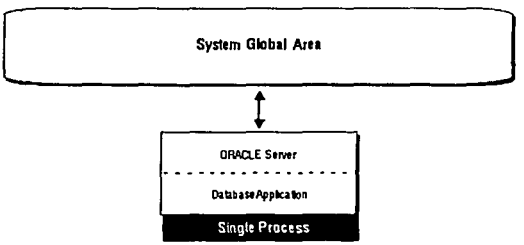


Figura 2.3. SGA. System Global Area (Área Global del Sistema)

El SGA está compuesta por la memoria caché de biblioteca, la memoria caché del diccionario (El diccionario de datos es un conjunto de tablas de sólo lectura que proveen información acerca de la base de datos asociada) y cierta información de sección del usuario y servidor (Figura 2.4.)

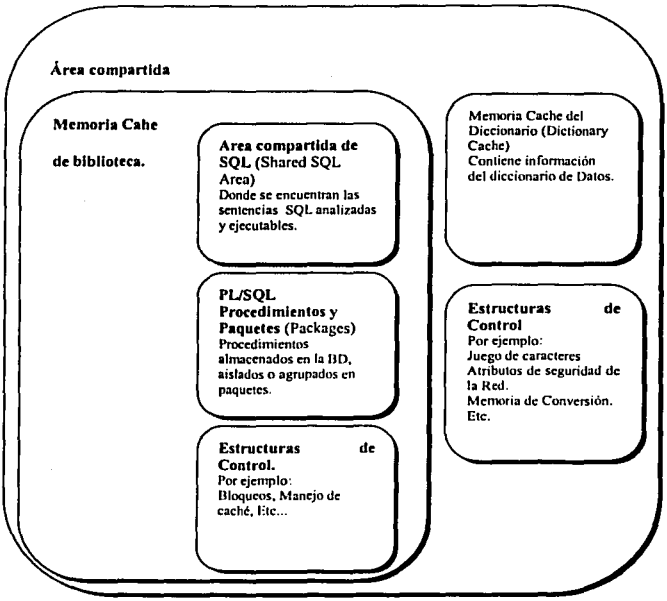


Figura 2.4 Partes principales del SGA.

La memoria caché de biblioteca. Esta memoria caché contiene sentencias SQL analizadas y ejecutables, para cada una de las sentencias SQL que Oracle procesa, existe siempre una parte compartida y otra privada. La porción compartida del SGA de cada sentencia SQL es la cantidad de memoria (perteneciente al área compartida) que contienen los siguientes componentes:

- El árbol de análisis. Una representación de los resultados obtenidos al analizar cualquier sentencia SQL.
- Plan de ejecución. Se trata de un mapa que construye Oracle y que contiene la planificación con la que se ejecutará una sentencia, se rescribe después de que se haya optimizado cada sentencia SQL.

La parte privada tiene dos componentes:

- La porción persistente que ocupa espacio en el SGA durante la vida de cada cursor asociado con una sentencia SQL.

- La porción en tiempo de ejecución que se adquiere cuando se ejecuta una sentencia SQL, y se libera cuando se completa la sentencia.

Para hacer un uso eficaz del espacio asignado al SGA, se deben cerrar los cursores cuando se haya terminado con ellos para liberar la memoria asignada a esta porción en tiempo de ejecución.

La memoria caché del diccionario, contiene información del diccionario de datos relacionada con los segmentos de la base de datos (índices, tablas, etc.), disponibilidad del espacio de archivo (para adquisición de espacio mediante creación y extensión de objetos) y privilegios de objeto.

En el SGA, se encuentra ubicada el área denominada compartida. Este segmento de memoria contiene sentencias SQL analizadas y ejecutables. Las sentencias que se mandan a ejecutar posteriormente se comparan con las sentencias contenidas en el área compartida, si la nueva sentencia concuerda con una que se encuentre ya en el área, el manejador ejecuta la sentencia compilada en lugar de procesar la sentencia que acaba de recibir. Para habilitar esta condición de emparejamiento se deben cumplir las siguientes reglas:

I. Debe haber una concordancia caracter a caracter (distinguiendo entre mayúsculas y minúsculas) entre la sentencia que se esta examinando y aquella que se encuentra ya en el área compartida. Antes de que se lleve a cabo esta comparación, Oracle aplica un algoritmo interno utilizando la nueva sentencia. Después comprueba los resultados frente a los valores de las sentencias que ya están en el área, si el nuevo valor coincide con uno que ya esté allí, se lleva acabo la comparación de cadena definida anteriormente. Las siguientes sentencias sirven para ejemplificar esta regla:

```
1. Select pin from person where last_name = 'LAU';
2. Select PIN from person where last_name = 'LAU';
```

Las sentencias 1 y 2 no son coincidentes porque "pin" se encuentra escrito en minúsculas en la primera y en mayúsculas en la segunda.

```
1. Select pin from person where last_name = 'LAU';
2. Select pin from person where last_name = 'LAU';
```

Las sentencias 1 y 2 son coincidentes por que los caracteres contenidos en ambas instrucciones son exactamente iguales.

```
1. Select pin from person where last_name = 'LAU';
2. Select pin from person where last_name = 'LAU';
```

Las sentencias 1 y 2 no son coincidentes porque la primera sentencia está dividida en dos líneas mientras que la segunda no.

II. Los objetos a los que se hace referencia en la nueva sentencia son exactamente iguales que los objetos de una sentencia que han pasado la comparación mencionada en la regla I. Si se modifica un objeto al que se hace referencia en la sentencia SQL del área compartida, a la sentencia se le pone un indicador de no válida, la siguiente vez que se le pase a Oracle una sentencia que sea igual a la sentencia no válida, se sustituirá la antigua instrucción por la nueva, ya que se ha modificado el objeto anterior. Para ejemplificar esta regla supongamos que los usuarios de un sistema tienen acceso a ciertas tablas, y la vía de acceso es mediante un sinónimo privado o público. Suceden los siguientes casos:

- Si las instrucciones hacen referencia a los mismos objetos pero con sinónimos privados distintos no son coincidentes, son distintos objetos.

- Si las instrucciones hacen referencia a un mismo objeto, con un sinónimo público si es coincidente, es el mismo objeto.
- Si un usuario hace referencia a una tabla mediante un sinónimo público y otro lo accesa y es el dueño del objeto son objetos distintos.
- Si se hace referencia a variables (parámetros), deberán tener el mismo nombre en las dos sentencias, tanto en la nueva como en la anterior. En los siguientes ejemplos las dos sentencias siguientes son idénticas, aunque la variable tome valores distintos:

```
Select user_nm from usuarios where user_id = :xuser;
Select user_nm from usuarios where user_id = :xuser;
```

Las siguientes sentencias no son idénticas aunque el valor de la variable, sea el mismo en tiempo de ejecución.

```
Select user_nm from usuarios where user_id = :xuser;
Select user_nm from usuarios where user_id = :xuser1;
```

Se puede saber el contenido de esta área revisando la tabla interna de Oracle llamada v_$sqlarea y otra que contiene la descripción de las sentencias v_$sqltext, Fig. 2.5. Se puede utilizar una sentencia como la siguiente:

```
Select b.address, b.sql_text, sorts, users_executing
From sys.v_$sqlarea a, sys.v_$sql_text b
Where a.hash_value = b.hash_value
Order by b.address, piece, sorts, users_executing;
```

```
ADDRESS  SQL_TEXT                                                    SORTS      USERS_EXEC
-------- ------------------------------------------------------------ ---------- ----------
02BF55E4 SELECT A.BPKG_CD FROM BPKG_X_SENT A, BEV_PKG B, SUBTRD_CH D,SUBC          0          0
02BF55E4 H_X_SENT E WHERE A.BPKG_CD=B.BPKG_CD AND B.BPKG_CD= '8001051497'          0          0
02BF55E4  AND A.SENT_CD =14 AND A.SENT_CD=E.SENT_CD AND D.SUBCH_CD=E.SUBC         0          0
02BF55E4 H_CD AND E.SUBCH_CD= 204 AND E.BPKG_CD=A.BPKG_CD AND A.BPP_STATU         0          0
02BF55E4 S_CD =1 AND  to_date('30-06-2000','dd-mm-yyyy') >= TO_DATE(A.ACT         0          0
02BF55E4 V_DT) AND (to_date('30-06-2000','dd-mm-yyyy') < TO_DATE(A.INAC_D         0          0
02BF55E4 T) or TO_DATE(A.INAC_DT) is null) AND to_date('30-06-2000','dd-m         0          0
02BF55E4 m-yyyy') >= TO_DATE(E.ACTV_DT) AND (to_date('30-06-2000','dd-mm-         0          0
02BF55E4 yyyy') < TO_DATE(E.INAC_DT) or TO_DATE(E.INAC_DT) is null)               0          0
0342ADD8 select * from v$sqlarea                                                  1          0
03430080 Select *  from desarrollo.area  order by area_nm                         1          0
```

Fig. 2.5. Contenido de las tablas: v_$sqlarea y v_$sqltext, que muestran las sentencias que se encuentran el en área compartida, pueden ocupar varios registros pero tienen la misma dirección (address).

Esta sentencia permite obtener el texto de todas las sentencias SQL y PL/SQL que se encuentran en el área compartida. PL/SQL combina el poder de manipulación de los datos de SQL con el poder de procesamiento de los datos de un lenguaje procedural, que cuenta con sentencias de control de flujo, declaración de constantes, variables, definición de procedimientos y funciones y la posibilidad de capturar errores. La columna *sql_text* muestra las sentencias que han sido ejecutadas por el manejador y que se encuentran ya "compiladas", es decir, ya tienen un plan de ejecución o rutas de acceso definidas para obtener o modificar los datos que se requieren.

Optimizar el área compartida

El área compartida contiene sentencias SQL y PL/SQL (conocidas como memoria cache de biblioteca), la memoria caché de diccionario de datos e información sobre sesiones de la base de datos. El tamaño del área compartida lo dicta el valor asignado a la entrada SHARED_POOL_SIZE del archivo de parámetros de inicialización. Puede que el DBA tenga que revisar el tamaño del área compartida, si recibe errores ORA-4031 (incapaz de asignar bytes de memoria compartida) cuando Oracle intenta encontrar más espacio en la memoria compartida.

Las sentencias se colocan en el área compartida y se eliminan de acuerdo a un algoritmo LRU (Last Recentrly Used). Aunque con el tiempo, las sentencias analizadas desaparezcan del área compartida, el texto SQL permanece en las tablas de diccionario contenidas en memoria durante periodos mayores de tiempo (si el espacio lo permite). Se deberá cerrar y volver a iniciar la base de datos para activar un nuevo tamaño del área compartida.

La forma más común de supervisar la actividad de la memoria caché de biblioteca es consultando la tabla del diccionario de datos denominada V$LIBRARYCAHE (Fig. 2.6) con un código similar al siguiente:

```
Select namespace, gets, gethits, gethitratio, pins, pinhits, pinhitratio
From v$librarycache;
```

NAMESPACE	GETS	GETHITS	GETHITRATI	PINS	PINHITS	PINHITRATI
SQL AREA	7518	5383	.716014898	17417	13104	.752368376
TABLE/PROCEDURE	8248	8061	.977327837	8783	8585	.97745645
BODY	15	14	.933333333	15	14	.933333333
TRIGGER	0	0	1	0	0	1
INDEX	21	0	0	21	0	0
CLUSTER	27	12	.444444444	15	5	.333333333
OBJECT	0	0	1	0	0	1
PIPE	0	0	1	0	0	1

Fig. 2.6. V$LIBRARYCAHE: memoria caché de biblioteca.

Una memoria caché saludable debería mostrar una tasa de aciertos en el SQL_AREA muy próxima al 100%, en el ejemplo de la Fig. 2.4, no se encuentra bien ya que tiene una tasa de aciertos del 72%, este dato se localiza en el primer registro de la Fig. 2.4, en la columna *"gethitratio"*, del "namespace" SQL AREA. La tasa de aciertos mide cómo está gestionando la caché del buffer de datos las solicitudes de datos, se calcula de esta manera:

Tasa de aciertos = (Lecturas Lógicas – Lecturas Físicas) / Lecturas Lógicas

Con la opción CACHE se pueden manipular las acciones del algoritmo LRU en la caché del buffer de datos. Esta opción carga automáticamente una tabla completa en el área global del sistema (SGA) la primera vez que se accede a ella y la marca como la más recientemente utilizada. Los datos de esta tabla todavía dependerán de los algoritmos LRU que gestionan las memorias caché del SGA, pero permanecerán en el SGA por más tiempo que si se hubiera tratado de la manera habitual. La opción CACHE se puede especificar en el nivel de tabla mediante las órdenes CREATE TABLE y ALTER TABLE, y también puede especificarse mediante indicadores ("sugerencias") en las consultas. Esta opción es mucho más útil para las tablas a las que se accede frecuentemente pero que no suelen cambiar con frecuencia. Con ella se pueden ejecutar consultas que vuelvan a cargar las tablas más utilizadas en las memorias caché SGA cada vez que se reinicialice la base de datos.

En todas las áreas de memoria global del sistema (los buffers de bloques de datos, la caché del diccionario y el área SQL compartida), lo más importante es compartir los datos entre usuarios. Los DBA pensarán que modificar el tamaño del área compartida es, de nuevo un juego de prueba y error. El tamaño asignado por defecto es 3.5 MB, y el valor hay que darlo en bytes. La memoria caché de biblioteca almacena todas las instrucciones SQL y PL/SQL que se hayan ejecutado, poder acceder con rapidez a esta memoria caché es la mejor garantía de que se ha realizado una correcta optimización del área compartida. Las instrucciones se cargarán y fluirán desde está área utilizando el algoritmo LRU.

Para acceder a las vistas V$ del diccionario, el DBA debe conceder privilegios a los usuarios de la base de datos para poder contemplar dichas vistas. Las vistas empiezan realmente con v_$, y se deberá utilizar este nombre (el real) en lugar del sinónimo público cuando se concedan los privilegios.

Oracle hace sitio en el área compartida para almacenar nuevas sentencias cuando procesa una sentencia que no concuerda con ninguna de las que ya se encuentran allí. La entrada CURSOR_SPACE_FOR_TIME del archivo de parámetros de inicialización es la encargada de definir el momento en que saldrá del área compartida una sentencia SQL y se vuelven a escribir en el disco. Cuando se asigna el valor FALSE a este parámetro (el valor implícito), Oracle elimina el espacio que contiene una sentencia en el área compartida, incluso si los cursores de la aplicación que utilizan esa sentencia están aún abiertos. Si la cantidad de memoria disponible es suficiente como para que el área compartida pueda tener un tamaño que le permita almacenar todos los cursores de la aplicación, se podrá pensar en asignar el valor TRUE a este parámetro. Cuando esto ocurra, cada vez que se vaya a ejecutar una sentencia SQL se ahorrará una pequeña cantidad de tiempo, Oracle no tiene que explorar el área para ver si la sentencia ya esta allí.

2.6 Proceso interno en la ejecución de una consulta

Antes de optimizar el SQL de las aplicaciones, es conveniente comprender el esquema de procesamiento del SQL de Oracle, ahora analizaremos brevemente como es que se procesa una sentencia SQL cada que se hace la petición a Oracle de ejecutar una sentencia.

- Ejecución de sentencias SQL
- Procesamiento de sentencias DML
- Procesamiento de sentencias DDL

Ejecución de sentencias SQL

La figura 2.7 muestra el escenario comúnmente usado en el proceso de ejecución de una sentencia SQL, en algunos casos, Oracle podría excluir algunos pasos o realizarlos en diferente orden. Por ejemplo la etapa DEFINE debe ocurrir justo antes del FETCH (recuperación de los registros, en este caso del primero), dependiendo de como se haya escrito el código. No es necesario conocer un nivel de detalle tan pequeño, sin embargo, esta información es útil cuando se desarrollan aplicaciones en Oracle.

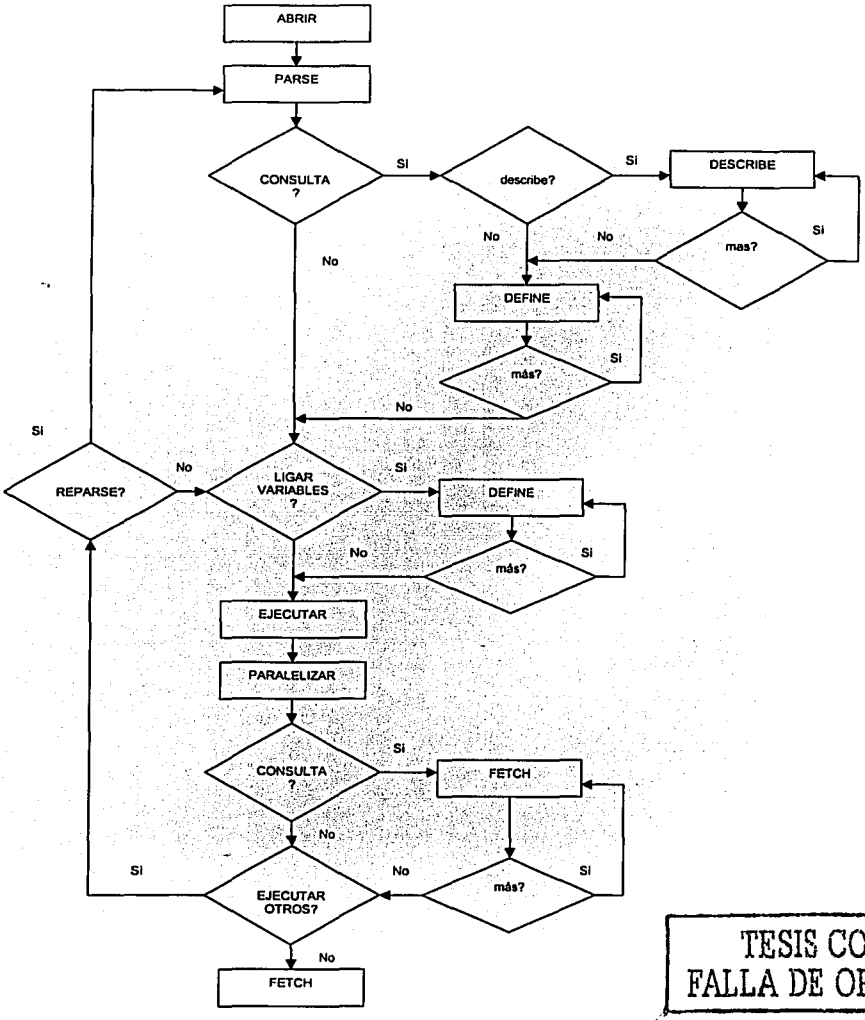


Figura 2.7. Las etapas en el procesamiento de sentencias SQL.

Fases en el procesamiento de una sentencia SQL

Etapa 1: Crear un Cursor (Abrir)

La interfaz del programa crea un cursor. El cursor es creado independientemente del tipo de sentencia SQL; es creado esperando cualquier tipo de sentencia SQL, en muchas aplicaciones, la creación del cursor es automática. Sin embargo en programas precompilados, la creación del cursor puede ocurrir implícitamente o explícitamente al declarar un cursor.

Etapa 2: Análisis de la sentencia ("Parse")

Durante el análisis, la sentencia SQL es pasada del proceso de usuario a Oracle y una representación del análisis de la sentencia SQL es cargada en la memoria compartida (SQL área). Muchos errores pueden detectarse durante esta fase del procesamiento de la sentencia. Analizar es el proceso de:

- Traducir la sentencia SQL, verificar si es una sentencia valida
- Realizar las revisiones al diccionario de datos para verificar las definiciones de las tablas y columnas
- Bloquear las definiciones de los objetos requeridos para que no cambien durante el análisis de la sentencia.
- Verificar los permisos de acceso a los objetos del esquema que se indican
- Determinar el plan de ejecución óptimo de la sentencia
- Cargar la sentencia en el área de memoria compartida SQL
- Para sentencias distribuidas, direccionar todo o parte de la sentencia a los nodos remotos que contienen los datos referidos.

Una sentencia SQL es analizada solo si no existe una sentencia SQL idéntica en la memoria cache, en este caso, la sentencia es colocada en la memoria compartida y se analiza. La fase de análisis incluye procesar los requerimientos que necesita para realizar solo una vez. Oracle traduce la sentencia SQL solo una vez, volviendo a ejecutar el análisis durante las llamadas subsecuentes a la sentencia si algún valor cambia. Además valida la sentencia, solo identifica errores que pueden ser encontrados antes de ejecutar la sentencia. Esto deja errores que no pueden capturarse durante el análisis, por ejemplo, errores en la conversión de los datos o en los datos (por ejemplo tratar de introducir datos duplicados a una llave primaria) y quedan pendientes todos los errores que solo pueden ser encontrados y reportados durante la fase de ejecución.

Procesamiento de consultas

Las consultas son diferentes a otro tipo de sentencias SQL porque estas recuperan datos como resultado si esta obtiene éxito, mientras otras sentencias simplemente regresan éxito o fracaso, una consulta puede recuperar un registro o miles de registros. Los resultados de una consulta siempre se muestran en un formato tabular, y los registros del resultado son recuperados, uno a la vez o en grupos. Las consultas no incluyen explícitamente una sentencia SELECT pero si hay consultas implícitas en otras sentencias. Por ejemplo, cada una de las siguientes sentencias requiere una consulta como parte de su ejecución:

```
INSERT INTO table SELECT...
UPDATE table SET x = y WHERE...
DELETE FROM table WHERE...
CREATE table AS SELECT...
```

En particular las consultas:

- Requieren consistencia de lectura de datos
- Pueden usar segmentos temporales para procesos intermedios
- Requieren de las fases de descripción, definición y recuperación del procesamiento de sentencias SQL

Etapa 3: Describir resultados (Describe)

La fase de descripción es necesaria solamente si las características del resultado de la consulta no es conocido, por ejemplo, cuando la consulta es introducida interactivamente por el usuario. En este caso, la fase de descripción es usada para determinar las características (tipos de datos, tamaños y nombres) del resultado de la consulta.

Etapa 4: Definir la salida (Define)

En la fase de definición para las consultas, nosotros especificamos la localización, tamaño y el tipo de dato de las variables definidas para recibir cada valor recuperado. Oracle realiza la conversión de tipos de datos si es necesario.

Etapa 5: Obtener el valor de las variables (Ligar variables)

En este punto, Oracle conoce el significado de la sentencia SQL pero aún no tiene la suficiente información para ejecutar la sentencia. Oracle necesita de los valores para todas las variables listadas en la sentencia, en el ejemplo, Oracle necesita el valor de DEPT_NUMBER, este proceso es llamado ligar variables. El programa debe especificar la localización (dirección de memoria) del valor de las variables, los usuarios finales de las aplicaciones pueden tomarse desprevenidos para especificar las variables porque Oracle puede solicitarlos si no se han asignado. Porque nosotros especificamos la localización (liga por referencia), necesitamos no volver a cambiar el valor de la variable antes de la ejecución. Podemos cambiar este valor y Oracle obtendrá el valor en cada ejecución, usando la dirección de memoria. Es conveniente indicar el tipo de dato correcto para las variables, de lo contrario Oracle realizara la conversión de los datos si es necesario, con esto evitaremos que se trabaje de más.

Etapa 6: Ejecutar la sentencia (Ejecutar)

Hasta este paso, Oracle tiene toda la información necesaria y los recursos, para que la sentencia pueda ser ejecutada. Si la sentencia es una consulta o una sentencia INSERT, no se necesita bloquear registros porque ningún dato será cambiado. Sin embargo si la sentencia es UPDATE o DELETE, todos los registros que afecta la sentencia son bloqueados, así ningún usuario de la case de datos puede utilizarlos, hasta el próximo COMMIT o ROLLBACK para la transacción. Esto asegura la integridad de los datos.

Etapa 7: Sentencias en paralelo

Cuando se usa la opción de consulta en paralelo, Oracle puede realizar consultas en paralelo y operaciones del tipo DML. Las sentencias ejecutadas en paralelo causan que múltiples servidores realicen y dividan el trabajo de la consulta para que esta se complete más rápido. La creación de índices y tablas mediante una subconsulta puede realizarse en paralelo.

Etapa 8: Recuperar los registros de la consulta (Fetch)

En la fase de recuperación, los registros son seleccionados y ordenados (si es requerido por la consulta), y sucesivamente hasta que se obtiene el último registro.

Procesamiento de sentencias DML

Una consulta (SELECT) requiere etapas adicionales, como la etapa de descripción. Si utilizamos un lenguaje como el Pro*C y queremos incrementar el salario de todos los empleados en un departamento. Se puede escribir la siguiente sentencia SQL en el programa:

```
EXEC SQL UPDATE emp SET sal = 1.10 * sal
WHERE deptno = :dept_number;
```

DEPT_NUMBER es una variable de programa que contiene un valor para el número de departamento. Cuando la sentencia SQL es ejecutada, el valor de DEPT_NUMBER es usado, y es proporcionado por el programa de aplicación.

Procesar sentencias del tipo DDL

La ejecución de sentencias DDL es diferente de la ejecución de sentencias DML y consultas, porque la realización exitosa de una sentencia DDL requiere acceso de escritura al diccionario de datos. Para estas sentencias, la fase de análisis incluye bloquear el diccionario de datos, para mantener su integridad. Los conceptos anteriores muestran de forma breve lo que el RDBMS realiza al recibir una sentencia SQL, para su ejecución, esto es importante para conocer el trabajo y las acciones que se desprenden de la ejecución de una sentencia.

Capítulo III. Optimización de la base de datos

3.1 Estructura de la base de datos

Una base de datos Oracle tiene una estructura física y una lógica. Porque la estructura física y lógica está separada, el almacenamiento físico de los datos puede ser administrado sin afectar el acceso lógico a las estructuras lógicas de almacenamiento. La estructura física de la base de datos es determinada por los sistemas de archivo que constituyen la base de datos. Cada base de datos de Oracle esta echa de tres tipos de archivos: uno o mas archivos de datos, dos o más archivos de registro y uno o mas archivos de control. Los archivos de una base de datos Oracle proveen del almacenamiento físico actual, para la información de la base de datos.

Estructuras lógicas de la base de datos

En Oracle las estructuras lógicas de la base de datos se determinan por:

- Uno o más "tablespaces" (Es una área lógica de almacenamiento)
- Objetos del esquema de la base de datos. Un esquema es una colección de objetos. Los objetos del esquema son estructuras lógicas que hacen referencia directa a los datos de la base de datos. Incluyen algunas estructuras como tablas, vistas, secuencias, procedimientos almacenados, sinónimos, índices, clusters (grupos) y ligas a la base de datos.

Las estructuras lógicas de la base de datos incluyen: tablespaces (espacio de tablas), schema objects (objetos de esquema), data blocks (bloques de datos), extents (extensiones), y segments (segmentos). Los objetos del esquema y las relaciones se consideran para el diseño relacional de la base de datos.

Tablespaces

Una base de datos es dividida en unidades lógicas de almacenamiento llamadas tablespaces (espacio de tablas). Un tablespace es utilizado para agrupar estructuras lógicas relacionadas. Por ejemplo, Los tablespaces comúnmente agrupan todos los objetos de una aplicación para simplificar las operaciones administrativas, también en varios tablespaces se pueden agrupar elementos relacionados para separar mejor las unidades lógicas de almacenamiento. La relación entre las bases de datos, tablespaces y archivos de datos se ilustra en la figura 3.1.

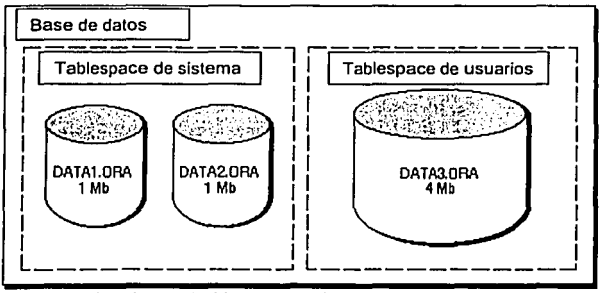


Figura 3.1. Databases, Tablespaces, y Datafiles.

Esta figura ilustra lo siguiente:

- Cada base de datos es dividida lógicamente en uno o más tablespaces.
- Uno o más datafiles (archivos de datos) son creados explícitamente por cada tablespace para almacenar físicamente los datos de todas las estructuras lógicas de un tablespace.
- El tamaño combinado de los datafiles de un tablespace es el total de la capacidad de almacenamiento de el tablespace (el tablespace SYSTEM tiene 2 Mb de capacidad de almacenamiento mientras que USERS tiene 4 Mb).
- La combinación de la capacidad de almacenamiento de los tablespaces de la base de datos es el total de la capacidad de almacenamiento de la base de datos (6 Mb).

Tablespaces en línea y fuera de línea

Un tablespace puede estar en línea (accesible) o fuera de línea (no accesible). Un tablespace esta normalmente en línea, así los usuarios pueden accesar la información que se encuentra dentro de este. Sin embargo aveces un tablespace es puesto en fuera de línea lo que hace que una parte de la base de datos no este disponible mientras se permite acceso normal en el resto de la base de datos. Esto hace muchas tareas administrativas fácil de realizar.

Esquemas

Un esquema es una colección de objetos. Un objeto de esquema es una estructura lógica que hace referencia directamente a los datos de la base de datos. Los objetos de esquema incluyen por ejemplo tablas, vistas, secuencias, procedimientos almacenados, sinónimos, índices, clusters (grupos) y ligas de la base de datos. (No hay relación entre un tablespace y un esquema; objetos en el mismo esquema pueden estar en diferentes tablespaces, y un tablespace puede contener objetos de diferentes esquemas)

Tablas

Una tabla es la unidad básica del almacenamiento de los datos en una base de datos Oracle. Las tablas de la base de datos tienen todos los datos accesibles por el usuario. La tabla es ordenada en renglones y columnas. Todas las tablas son definidas con un nombre y el establecimiento de sus columnas. A cada columna se le asigna un nombre, tipo de dato (por ejemplo: CHAR, DATE, o NUMBER) y tamaño (el cual puede ser predeterminado por el tipo de dato, como con DATE) o escalar su precisión (para el tipo de dato NUMERIC solamente). Una vez que la tabla es creada, pueden ser insertados renglones válidos de datos. Los renglones de la tabla pueden ser requeridos, borrados o actualizados.
Para hacer cumplir las reglas del negocio definidas en los datos de la tabla, reglas de integridad y triggers se puede definir también en la tabla.

Vistas

Una vista es una presentación de los datos diseñada para algún uso de una o más tablas. Una vista puede ser también considerada como un "Stored Query" (consulta almacenada). Las vistas no contienen o almacenan datos físicamente, es decir no ocupan espacio.

Secuencias

Una secuencia genera una lista serial de números para columnas numéricas de tablas de la base de datos. Una secuencia simplifica la programación de aplicaciones porque automáticamente genera valores numéricos únicos para los renglones de una sola tabla o múltiples tablas.

Unidades de Programación

Son los procedimientos almacenados, funciones y paquetes. Un procedimiento o una función es un conjunto de sentencias SQL y PL/SQL (Extensiones de lenguaje procedural para SQL) agrupadas en una unidad ejecutable que realizan una tarea específica. Los procedimientos y funciones permiten combinar la facilidad y flexibilidad de SQL con la funcionalidad procedural de un lenguaje estructurado de programación. Los procedimientos y funciones pueden ser definidos y almacenados en la base de datos para su uso continuo. Los procedimientos y funciones son idénticos, excepto porque las funciones siempre regresan un valor único al ser llamadas, mientras que los procedimientos no regresan valores al ser llamados.

Los paquetes proveen un método de encapsulación y almacenamiento de procedimientos, funciones y otros paquetes relacionados. Mientras los paquetes proveen beneficios de organización para el administrador de la base de datos y desarrolladores de aplicaciones, también ofrecen incremento en la funcionalidad y rendimiento de la base de datos.

Sinónimos

Un sinónimo es un alias de una tabla, una vista, secuencia o unidad de programación. Un sinónimo es una referencia directa de un objeto no es un objeto real. Un sinónimo es usado para:

- Enmascarar el real del nombre y el dueño del objeto.
- Proveer acceso público a un objeto
- Proveer una localización trasparente para tablas, vistas o unidades de programa de bases de datos remotas.
- Simplificar las sentencias SQL para los usuarios de la base de datos

Un sinónimo puede ser público o privado

Índices, Clusters (Grupos), y Hash Clusters (Grupos Hash)

Son estructuras opcionales asociadas a tablas, las cuales pueden ser creadas para incrementar el rendimiento de la recuperación de los datos. Los índices son creados para este fin, proveen un acceso rápido a la localización de los datos en la tabla. Cuando se procesa una petición, Oracle puede usar algunos o todos los índices disponibles para localizar la información requerida más eficientemente. Los índices son utilizados cuando las aplicaciones necesitan con frecuencia querys de una tabla para un rango de renglones (por ejemplo todos los empleados con un salario mayor a 5000 pesos) o un renglón específico (por ejemplo el empleado con el código 1000). Los índices pueden ser creados para una o más columnas de la tabla. Una vez creado, un índice es automáticamente mantenido y usado por Oracle. Los cambios hechos a los datos de la tabla (por ejemplo: agregar nuevos renglones, actualizar o borrar) son automáticamente incorporados en todos los índices involucrados con completa transparencia para los usuarios.

Los índices son lógica y físicamente independientes de los datos. Estos pueden ser borrados y creados en cualquier momento sin afectar en las tablas u otros índices. Si un índice es borrado, todas las aplicaciones continúan funcionando; sin embargo, el acceso a los datos anteriormente indexados puede ser más lento. Los grupos (clusters) son un método opcional para almacenar datos. Son grupos de una o más tablas almacenadas juntas físicamente porque estas comparten columnas comunes y son comúnmente usadas juntas. Por eso los renglones relacionados son almacenados juntos físicamente. El tiempo de acceso a los datos del disco en mejorado.

Las columnas relacionadas de las tablas en un grupo son llamadas llave del grupo. La llave del grupo es indexado así estos renglones del grupo pueden ser recuperados con una

mínimo cantidad de trabajo de entrada-salida (I/O). Porque los datos en la llave de grupo (un non-hash-cluster) son almacenados únicamente una vez para múltiples tablas, los grupos pueden almacenar un grupo de tablas más eficientemente que si las tablas son almacenadas individualmente (no agrupadas). La figura 3.2 ilustra como los datos agrupados y no agrupados son almacenados físicamente.

Los grupos pueden mejorar el rendimiento de la recuperación de los datos, dependiendo de la distribución de los datos y que operaciones SQL son más comúnmente ejecutadas en los datos agrupados. En particular, las tablas agrupadas que son requeridas en joins son beneficiadas del uso de grupos porque los renglones comunes que hacen el join de las tablas son recuperados con la misma operación de I/O. Como los índices, los grupos (clusters) no afectan al diseño de la aplicación. Si es o no la tabla parte de un grupo es transparente para el usuario y las aplicaciones. Los datos almacenados en un grupo son accesados vía SQL con el mismo camino que si los datos estuvieran almacenados en una tabla no agrupada.

Los hash clusters (grupos hash) también agrupan tablas de una manera similar a la normal pero además agrupa los índices (la llave del grupo mejora más la función hash). Es decir hay un índice único para las tablas agrupadas y estas se guardan junto al índice en la misma localidad física. Los hash clusters son una mejor elección que usar una tabla indexada o un grupo indexado, cuando la tabla es más requerida para consultas de igualdad (por ejemplo, recuperar todos los renglones para el departamento 10). Para estas consultas, el valor de la llave de grupo esta ordenada. El resultado del valor de la hash key pone directamente en el área sobre el disco donde se almacenan los renglones especificados.

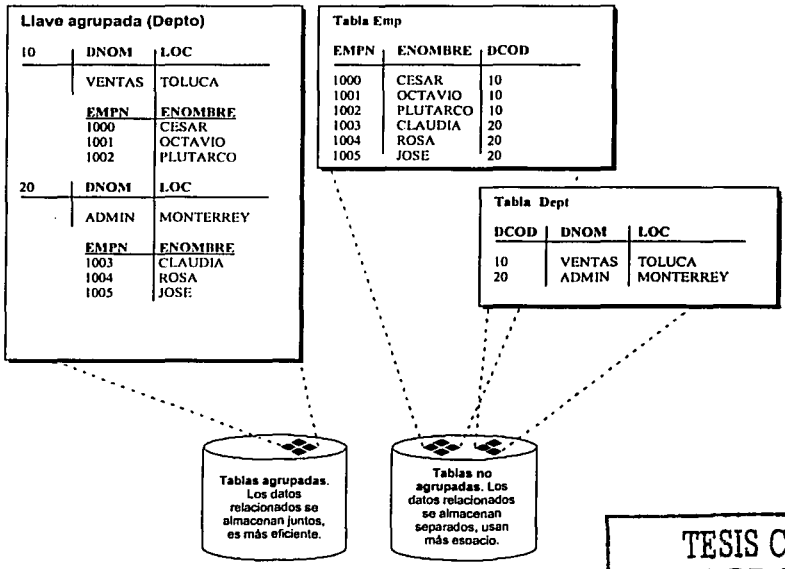


Figura. 3.2. Tablas agrupadas y no agrupadas.

Ligas a la base de datos (Database Links)

Es un objeto que describe el "path" o ruta de una base de datos a otra. Las ligas a las bases de datos son usadas implícitamente cuando se hace una referencia a un nombre de objeto global en una base de datos distribuida.

Bloques de datos, extensiones y segmentos

Oracle permite un fino control del espacio utilizado en el disco a través de las estructuras lógicas de almacenamiento, incluyendo los bloques de datos, extensiones y segmentos.

Bloques de datos

El más fino nivel de fragmentación, en las bases de datos Oracle es almacenado en bloques. Un bloque de datos corresponde a un número específico de bytes de el espacio físico en disco de una base de datos. El tamaño de un bloque de datos es especificado por cada base de datos Oracle cuando la base de datos es creada. Una base de datos usa y asigna espacio libre de la base de datos en bloques de datos Oracle.

Extensiones

El siguiente nivel del espacio lógico de las bases de datos es llamado extensión. Una extensión es un número específico de bloques de datos contiguos; encontrados en una sola distribución, usada para almacenar un tipo específico de información.

Segmentos

El nivel del almacenamiento lógico de la base de datos superior a las extensiones es llamado segmento. Un segmento es un conjunto de extensiones distribuidas para una estructura lógica segura. Por ejemplo, los diferentes tipos de segmentos incluyen la siguiente:

Segmento de datos

Cada tabla no agrupada tiene un segmento de datos. Todos los datos de la tabla son almacenados en las extensiones de este segmento de datos. Cada grupo tiene un segmento de datos. Los datos de todas las tablas de un grupo son almacenados en el segmento de datos del grupo.

Segmento índice

Cada índice tiene un segmento de índice que almacena todos sus datos.

Segmento de rollback (deshacer)

Uno o más segmentos de rollback (regresar al estado original los datos hasta antes de alguna transacción) son creados por el administrador de la base de datos para almacenar información temporal (*undo*) de la base de datos. Esta información es usada para:

- Generar lecturas consistentes de la información de la base de datos.
- Durante la recuperación de la base de datos de un respaldo.
- Realizar rollback de transacciones no terminadas por los usuarios.

Segmento temporal

Son creados por Oracle cuando una sentencia SQL necesita una área temporal de trabajo para completar su ejecución. Cuando la sentencia termina su ejecución, las extensiones del segmento temporal son devueltas al sistema para usos futuros.

Oracle asigna dinámicamente espacio cuando las extensiones existentes de un segmento llegan a llenarse. Así, cuando las extensiones existentes de un segmento están llenas, Oracle asigna otra extensión para el segmento que la necesite. Porque las extensiones son asignadas como se necesitan, las extensiones de un segmento pueden o no pueden estar continuas sobre el disco.

Estructuras físicas de la base de datos

Los siguientes conceptos explican las estructuras físicas de una base de datos en Oracle, incluyendo archivos de datos, archivos de registro de eventos y archivos de control.

Archivos de datos

Todas las bases de datos de Oracle tienen uno o más archivos de datos físicos. Estos archivos contienen todos los datos de la base de datos. Los datos lógicos de las estructuras de la base de datos como las tablas e índices son físicamente almacenados en los archivos de datos asignados para la base de datos. Características:

- Un archivo de datos puede ser asociado con únicamente una base de datos.
- Los archivos de bases de datos pueden tener características de seguridad que permiten automáticamente extenderse cuando se termina el espacio de la base de datos.
- Uno o más archivos de datos forman la unidad lógica de almacenamiento de la base de datos llamada *tablespace.*

El uso de los archivos de datos

Los datos en un archivo de datos, son leídos y requeridos durante una operación normal de la base de datos y almacenados en la memoria cache de Oracle. Por ejemplo, asume que los usuarios quieren accesar a algunos datos en una tabla de una base de datos. Si la información requerida no esta ya en la memoria cache de la base de datos, esta es leída de los archivos de datos apropiados y almacenada en la memoria. Los datos nuevos o modificados no se escriben necesariamente de inmediato. Para reducir la cantidad de accesos al disco e incrementar el desempeño, los datos son combinados en memoria y escritos en los archivos de datos apropiados al mismo tiempo, esto es determinado por el DBWR proceso de fondo de Oracle.

Archivos de registro (redo log files)

Todas las bases de Oracle tienen un conjunto de uno a más archivos de registro. El conjunto de archivos de registro de transacciones de una base de datos es conocido como el registro de transacciones de la base de datos. La función primaria del registro de transacciones es registrar todos los cambios hechos a los datos. Si ocurre una falla se impide a los datos modificados ser escritos permanentemente en los archivos de datos, los cambios pueden ser obtenidos de los archivos de registro o bitácora de transacciones y así el trabajo no se pierde nunca. Los archivos de registro de transacciones son críticos en la protección de la base de datos sobre posibles fallas. Para protegerse contra una falla se involucra el registro de transacciones en sí mismo, Oracle permite un registro de transacciones multiplexado con una o más copias en diferentes discos.

Uso de los archivos de registro de transacciones

La información de estos archivos es usada únicamente para recuperar la base de datos de una falla del sistema que impide que los datos de la base de datos sean escritos en los archivos de datos. Por ejemplo, si un paro del suministro eléctrico inesperado termina abruptamente con la operación de la base de datos, los datos en memoria no pueden ser escritos en los archivos de datos y los datos se pierden. Sin embargo, cualquier dato perdido puede ser recuperado cuando la base de datos es abierta, después de poderse restaurar. Aplicando la información contenida en los más recientes archivos de transacciones en los archivos de datos de la base de datos, Oracle restaura la base de datos al momento en que el fallo del suministro eléctrico ocurrió.

Archivos de control

Todas las bases de datos tienen un archivo de control. Un archivo de control contiene entradas que especifican la estructura física de la base de datos. Por ejemplo, contiene los tipos siguientes de información:

- Nombre de la base de datos.
- Nombres y ubicaciones de los archivos de datos y los archivos de registro de transacciones de la base de datos.
- Fecha y hora de la creación de la base de datos.

Al igual que los archivos de registro de transacciones, Oracle permite que los archivos de control sean multiplexados.

Uso de los archivos de control

Todas las veces que una instancia de una base de datos Oracle es iniciada, estos archivos de control son usados para identificar la base de datos y los archivos de registro de transacciones son abiertos para que la operación de la base de datos proceda. Si la estructura física de la base de datos es alterada (por ejemplo, un nuevo archivo de datos o de registro de transacciones es creado), el archivo de control de la base de datos es automáticamente modificado por Oracle para reflejar el cambio. Es necesario para restaurar una base de datos guardada.

El diccionario de datos

Todas las bases de datos Oracle tienen un diccionario de datos. Un diccionario de datos Oracle es un conjunto de tablas y vistas que son usadas como referencia de sólo lectura acerca de la base de datos. Por ejemplo, un diccionario de datos almacena información acerca de la estructura lógica y física de la base de datos. En adicción a esta importante información, el diccionario de datos también almacena información como:

- Los usuarios válidos de una base de datos Oracle.
- Información acerca de las reglas de integridad definidas para las tablas de la base de datos.
- Cuanto espacio es destinado para un objeto de esquema y cuanto de este es utilizado.

Un diccionario de datos es creado cuando la base de datos es creada. Para reflejar con precisión el estado de la base de datos en cualquier momento, el diccionario de datos es automáticamente actualizado por Oracle en repuesta a acciones específicas (por ejemplo, cuando la estructura física de la base de datos es alterada). El diccionario de datos es crítico en la operación de la base de datos que confía en el diccionario de datos para registrar, verificar y conducir los procesos. Por ejemplo, durante la operación de la base de datos, Oracle lee el diccionario de datos para verificar que los objetos de esquema existan y los usuarios tengan acceso apropiado a ellos.

3.2 Configuración y diseño de la base de datos

La configuración lógica de la base de datos influye enormemente en su rendimiento y en la facilidad de su administración. La primera persona que formalizó la distribución efectiva de los objetos lógicos de una base de datos fue Cary Millsap, de Oracle, quién denominó a la arquitectura resultante *arquitectura flexible óptima* (Optimal Flexible Architecture, OFA). La planificación de la distribución utilizando esta arquitectura facilita la administración de la base de datos y permite al DBA un mayor número de opciones a la hora de planificar y optimizar la distribución física. La disposición OFA estándar de espacios de tablas se crea automáticamente al utilizar el software de instalación de Oracle.

El objetivo del diseño de la base de datos es configurarla de forma que los objetos estén separados por el tipo de objeto y por su tipo de actividad. Esta configuración reduce enormemente la cantidad de trabajo administrativo que debe realizarse en la base de datos a la vez que disminuye las necesidades de supervisión. De esta forma los problemas de un área no afectarán al resto de la base de datos. Además la correcta distribución de los objetos de la base de datos permite una mayor flexibilidad a la hora de realizar el modelo físico de la base de datos, es decir, el modelo físico se facilita si el modelo lógico esta bien hecho.

Para distribuir los objetos lo primero que se debe hacer es establecer un sistema de clasificación. Los objetos lógicos de la base de datos deben clasificarse en función de la forma en la que vayan a utilizarse y de la influencia que tenga su estructura física en la base de datos. Para ello hay que separar las tablas de sus índices y las tablas con mucha actividad de aquellas con poca actividad. Aunque el volumen de actividad de los objetos sólo puede determinarse durante el uso en el entorno de producción, normalmente puede ser posible aislar un grupo de tablas muy utilizadas.

Dimensionamiento de los objetos de la base de datos

Uso del espacio de la base de datos

Para entender como se debería asignar el espacio dentro de la base de datos, es necesario saber cómo se utiliza el espacio en ella. Al crear una base de datos, ésta se divide en numerosas secciones lógicas denominadas tablespaces (espacios de tablas). El tablespace SYSTEM es el primero que se crea, luego se crean otros en los que se guardan los diferentes tipos de datos. Al crear un tablespace también se crean archivos de datos para guardar los datos correspondientes. Estos archivos asignan inmediatamente el espacio que se ha especificado durante su creación. Por tanto existe una relación uno a muchos entre las bases de datos y los tablespaces y entre los tablespaces y los archivos de datos. Una base de datos puede tener numerosos usuarios, cada uno de los cuales posee un esquema. Cada esquema de usuario es una colección de objetos lógicos de la base de datos, como tablas e índices. Estos objetos hacen referencia a estructuras físicas de datos que se almacenan en tablespaces. Los objetos del esquema de un usuario pueden guardarse en múltiples tablespaces y un solo tablespace puede contener objetos de múltiples esquemas.

Cuando se crea un objeto de la base de datos (por ejemplo, una tabla o un índice), se asigna a un tablespace mediante las opciones predeterminadas del usuario o mediante instrucciones específicas. En este tablespace se crea un segmento (que puede ser del siguiente tipo: TABLE, INDEX, ROLLBACK, TEMPORAY, PARTITION y CLUSTER), que es el que contiene los datos asociados con dicho objeto. El espacio que se asigna al segmento no se libera hasta que el segmento se elimina, se contrae o se trunca (con la cláusula TRUNCATE). A partir de la versión 7.3 de Oracle se puede asignar parte del espacio de las tablas, índices y clusters (agrupaciones). Cada segmento consta de una serie de secciones denominadas extensiones, que son conjuntos de bloques de Oracle contiguos. Cuando las extensiones

existentes ya no pueden contener nuevos datos, el segmento obtiene una nueva extensión. Este proceso de ampliación continúa hasta que ya no hay más espacio disponible en los archivos de datos del tablespace o hasta que se alcance el número máximo interno de extensiones por segmento. Si un segmento se compone de numerosas extensiones, no hay ninguna garantía de que vayan a ser contiguas.

Implicaciones de la cláusula storage

La correcta definición del tamaño de los archivos que contienen los objetos de la base de datos implica un buen funcionamiento de estos cuando son requeridos. Si no es correcto el dimensionamiento de los objetos se puede afectar el rendimiento, porque un mismo objeto lógico guardado en muchos archivos físicos requiere de más uso de recursos para analizarlo, que uno que esta guardado en un solo archivo físico de tamaño adecuado.

La cantidad de espacio que utiliza un segmento esta determinada por sus parámetros de almacenamiento. Estos parámetros los determina la base de datos en el momento de crear el segmento; si no se especifica ningún parámetro de almacenamiento (storage) en la orden CREATE TABLE, CREATE INDEX, CREATE CLUSTER o CREATE ROLLBACK SEGMENT, utilizará los parámetros de almacenamiento predeterminados del espacio de tablas en el cual se va a guardar. Los parámetros de almacenamiento especifican el tamaño inicial de la extensión (initial), el tamaño de la siguiente extensión (next), el valor pcincrease (un factor por el cual cada extensión sucesiva aumentará de forma geométrica), el valor número máximo de extensiones (maxextens) y el número mínimo de extensiones. Una vez creado el segmento, no se pueden modificar los valores initial y minextens. Los valores predeterminados de los parámetros de almacenamiento de cada tablespace se encuentran en las vistas DBA_TABLESPACES y USER_TABLESPACES.

Al crear un segmento, éste adquirirá al menos una extensión, (se pueden especificar otros valores por medio de minextens). Esta extensión se utilizará para almacenar los datos hasta que ya no haya más espacio libre disponible (se puede utilizar la cláusula pctfree para reservar, en el interior de cada bloque de cada extensión, un porcentaje de espacio, que se mantendrá disponible para las actualizaciones de las filas existentes). Cuando se añadan datos adicionales al segmento, esté se ampliará obteniendo una segunda extensión del tamaño que se especifique en el parámetro next. No hay ninguna garantía de que la segunda extensión se encuentre contigua físicamente a la primera. El parámetro pctincrease está diseñado para minimizar el número de extensiones de las tablas en crecimiento. Podría ser peligroso que el valor de este parámetro fuera diferente a cero, ya que haría que el tamaño de cada extensión sucesiva aumentará geométricamente por el factor pctincrease especificado. Por ejemplo, imaginemos un segmento que tuviera un tamaño de extensión inicial (initial) de 20 bloques de Oracle y un valor pctincrease de 50, en la tabla 3.3 vemos el tamaño de las primeras 10 extensiones de ese segmento.

Número de extensión	Tamaño (en bloques de Oracle)	Total	Comentarios sobre el tamaño de extensión
1	20	20	INITIAL
2	20	40	NEXT
3	30	70	NEXT * 1.5
4	45	115	NEXT * 1.5 * 1.5
5	70	185	NEXT * 1.5 * 1.5 * 1.5
6	105	290	NEXT * 1.5 * 1.5 * 1.5 * 1.5
7	155	445	ETC.
8	230	675	
9	345	1020	
10	520	1540	

Tabla 3.3 Efecto de utilizar un valor de pctincrease distinto a cero.

En tan solo 10 extensiones, el tamaño se ha incrementado un 7,700%. Esto, además de indicar que la planificación del espacio que ha elaborado el desarrollador es inadecuada, también representa un problema administrativo para el DBA. La tabla se encuentra bastante fragmentada, los más probable es que las extensiones no sean contiguas y la próxima vez que esta tabla se extienda (aunque solo sea para una fila de datos) necesitará alrededor de 750 bloques (que con un tamaño de bloque de 2 K, equivale a 1.5 MB). Sería preferible tener una sola extensión del tamaño adecuado, con un valor reducido para next e igual a cero para pctincrease. Esto evitaría el tener que realizar la defragmentación de los segmentos.

Nuca cambie el valor de pctincrease sin cambiar también el de next. El tamaño de cada extensión sucesiva se calcula utilizando de nuevo los parámetros de almacenamiento de la tabla; no se tiene en cuenta el tamaño de la última extensión que se ha añadido. Si por ejemplo, ahora tuviera la intención de cambiar el valor de pctincrease a 0 para el segmento de la tabla 3.1, la extensión número 11 tendría un tamaño de 20 bloques, no de 520 bloques.

Dimensionamiento adecuado

Seleccionar la adecuada asignación del espacio para los objetos de la base de datos es extremadamente importante. Los desarrolladores deberían determinar cuáles son los requisitos de espacio antes de crear los primeros objetos de la base de datos. Posteriormente, esos requisitos de espacio se podrán ajustar en función de las estadísticas de uso reales. Enseguida se mostrarán los cálculos de los requisitos de espacio para las tablas, los índices y las agrupaciones y del valor de adecuado para *pctincrease*. En las formulas que recomienda Oracle, hay que llevar una serie de cálculos muy detallados para dimensionar las tablas y los índices. No es necesario que estos cálculos sean muy precisos, ya que al terminar el cálculo se aconseja añadir entre un 10% y un 20% de espacio más a los requisitos de espacio estimados.

Dimensionamiento de las tablas no agrupadas

Además de mostrar los requisitos del espacio inicial de una tabla, también se debería proporcionar datos sobre el porcentaje estimado del crecimiento anual en el número de registros por cada tabla. Si es posible también se debería definir un número máximo de registros. Una vez que se conocen las definiciones de columna de la tabla y el volumen de datos, es el momento de determinar los requisitos de almacenamiento. En este momento no es posible hacer más que conjeturas basadas en la propia experiencia, ya que el volumen de datos y la longitud reales de las filas no se pueden conocer hasta haber creado la tabla. En este momento es importante disponer de datos de ejemplo para que el resultado de estos cálculos sea lo más exacto posible.

En primer lugar calcule la cantidad de espacio que va a utilizar la cabecera del bloque; Oracle empleará este espacio para manejar los datos contenidos en el bloque. El tamaño de la cabecera del bloque es de 90 bytes, aproximadamente. Si se utiliza un bloque de 2K, quedarían 1,958 bytes libres, mientras que si se utiliza un bloque de 4K, habría 4,006 bytes libres.

Ejemplo de espacio útil para un bloque de 2k:

2048 – 90 = 1958

A continuación, multiplique el espacio libre por el factor que viene determinado por el parámetro *pctfree* de la tabla, para determinar la cantidad de despacio que se va a reservar para la actualización de las filas. Si utiliza un valor de *pctfree* de 10, multiplique el espacio libre disponible por 0.10, como se muestra enseguida:

1958*(pctfree/100) = 1958*.01= 191 (redondeado)

Del espacio libre disponible, 196 son para las ampliaciones de filas. El espacio libre disponible es el espacio libre del bloque menos el espacio reservado por *pctfree*:

1958 – 196 = 1762 bytes disponibles

De los 2048 bytes del bloque, 1762 se utilizan para almacenar las filas, figura 3.4.

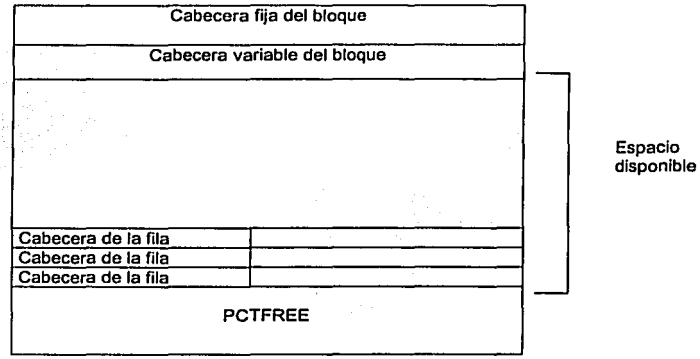


Fig. 3.4 Asignación del espacio dentro de los bloques.

En el siguiente paso habrá que calcular el espacio que se va a calcular por fila. Para esto hay que estimar primero la longitud media de las filas, que es la suma de la longitud media de cada valor de una fila. Si no se dispone de ningún dato, se debe realizar una estimación de la longitud real de los valores de una columna. No utilice como longitud real la longitud total de una columna, a menos que los datos vallan a llenar siempre por completo dicha columna. Por ejemplo vamos a suponer que tenemos una tabla de tres columnas de tipo VARCHAR2(10). La longitud media de las filas no puede exceder de 30; su longitud real depende de los datos que se vayan a almacenar en ellas. Si se dispone de algún dato de ejemplo, podría usar la función VSIZE para determinar el espacio real que han utilizado los datos. Vamos a suponer una vez más que la tabla posee tres columnas. Para determinar la longitud media de sus filas, se ejecuta la siguiente consulta:

```
Select  AVG(NVL(VSIZE(Columna1),0)) +
        AVG(NVL(VSIZE(Columna2),0)) +
        AVG(NVL(VSIZE(Columna3),0)) Longitud_media_x_ registro
From    Nombretabla;
```

En este ejemplo se ha calculado la longitud media de cada columna y luego se han sumado esas medias para determinar la longitud de la fila. Vamos a suponer que en el ejemplo de la tabla de tres columnas la longitud media de las filas es de 24 bytes. A esta cantidad total se agrega 1 byte por cada columna que haya en la tabla, lo que da un total de 27 bytes por fila. Si en la tabla hay datos que contienen más de 250 caracteres, se añade un byte adicional por cada una de esas columnas. Por último se añaden 3 bytes para la cabecera de la fila. De esta manera el tamaño de la fila se calcula así:

Espacio utilizado por registro = longitud_media_x_ registro
+ 3
+ Número de columnas
+ Número de columnas largas

En la tabla de ejemplo el espacio por registro es el siguiente:

Espacio utilizado por la fila = 24
+ 3
+ 3
+ 0
= 30 bytes por registro.

Dado que disponemos de 1762 bytes y 30 bytes por registro, se pueden incluir 58 filas en cada bloque:

Filas por bloque = TRUNC(1762 bytes libres / 30 bytes por registro)
= 58 registros por bloque

Al poderse incluir 58 registros por bloque, en cuanto se pueda estimar el número de registros que se espera que haya, se podrá estimar el número de bloques que se necesitan. Este cálculo es una aproximación, a medida que se manipulen los registros de la tabla, los requisitos de espacio aumentan. Cuantos más elementos se borren o se actualicen, más cantidad de espacio se requerirá.

Determinación del valor de *pctfree* adecuado

Hay que determinar el valor *pctfree* adecuado para cada tabla. Este valor representa el porcentaje de cada bloque de datos que se reserva como espacio libre. Este espacio se utiliza cuando una de las filas que ya está guardada en ese bloque de datos aumenta de longitud, ya sea porque se han actualizado los campos que anteriormente tenían valores NULL o porque se han actualizado valores existentes, introduciendo valores más largos.

No existe un único valor para pctfree que sea adecuado para todas las tablas de todas las bases de datos. Pero dado que pctfree está íntimamente ligado a la forma en la que se realizan actualizaciones en una aplicación, el proceso para determinar la idoneidad de su valor es muy directo. El parámetro pctfree controla el número de registros que se almacenan un bloque de una tabla. Par ver si pctfree está configurado correctamente, primero se debe determinar el número de registros que hay en un bloque. Se puede utilizar la sentencia ANALYZE, para determinar el número de registros por bloque que hay en una tabla existente. En la siguiente sentencia se ha utilizado la cláusula *compute statistics* de la orden ANALYSE para generar las estadísticas desde las vistas del diccionario de datos:

```
Analize table nom_tabla compute statistics;
```

Una vez analizada la tabla, se consulta su información e la vista USER_TABLES y registre sus valores *num_rows* y *blocks*. Si dividimos el valor num_rows por el de blocks, obtendremos el número de filas que hay guardadas por bloque, como se muestra en la siguiente consulta:

```
Select num_rows,          /* Número de registros */
       blocks,            /* Número de bloques utilizados */
       num_rows/blocks    /* Número de registros por bloque */
from user_tables
where table_name = 'Nombre_tabla';
```

Una vez conocido el número de filas por bloque, habrá que actualizar (UPDATE) los registros de la tabla de una forma que se emule su utilización en el entorno de producción. Una vez completadas las actualizaciones, se comprueba el número de filas por bloque, analizando la tabla y volviendo a ejecutar la consulta anterior. Si el valor de pctfree no era lo suficientemente alto, puede que algunas filas se hayan trasladado a nuevos bloques de datos para ajustarse a su nueva longitud. Por el contrario, si su valor no ha cambiado su valor es adecuado. No obstante, este valor podría ser demasiado alto, por lo que se estaría desperdiciando espacio. La sentencia ANALYSE que se ha mencionado antes, genera también valores para la columna *avg_space* de la vista USER_TABLES. En esta columna se muestra el número medio de bytes libres en cada bloque de datos. Si este número es muy alto después de la prueba de actualización (UPDATE), se podría disminuir el valor de pctfree.

Determinación del valor de *pctused* adecuado

El parámetro *pctused* es el que determina cuándo se vuelve a añadir un bloque usado a la lista de bloques disponibles para insertar filas. Por ejemplo, vamos a suponer que tenemos una tabla que tiene un valor de pctfree de 20 y un valor de pctused de 50. Cuando se inserten registros en dicha tabla, Oracle mantendrá libre el 20% del espacio de cada bloque (para emplearlo en actualizaciones futuras de los registros insertados). Si en este momento comenzará a borrar (DELETE) registros del bloque, Oracle no volvería a utilizar el espacio liberado de los bloques. No se insertarán (INSERT) nuevos registros en el bloque hasta que el espacio utilizado de dicho bloque sea menor que el porcentaje definido por pctused; es decir, el 50%.

El parámetro pctused tiene un valor predeterminado de 40. Si en una aplicación se realizan borrados con frecuencia y utilizan el valor predeterminado de pctused, puede que en la tabla haya muchos bloques que solo se utilicen en un 40%. Para obtener mejores resultados, se debe asignar a pctused un valor igual a:

pctused = 95 – pctfree

Si el valor de pctfree es de 20%, se configura el parámetro pctused con un 75%. De esta manera se utilizarán al menos el 75% de cada bloque, mientras que se guardará un 20% para las actualizaciones (UPDATE) y las ampliaciones de fila.

Dimensionamiento de los índices

El proceso de dimensionamiento de los índices es muy parecido al de las tablas. Sin embargo existen algunas diferencias en el dimensionamiento de estos objetos, ya que las tablas y los índices tienen estructuras diferentes. Al igual que con las tablas los cálculos que se obtienen son aproximaciones. Cuando se conoce el volumen de los datos y las definiciones de columna de un índice, ya se puede determinar los requisitos de espacio. Es importante disponer de datos de ejemplo para que los resultados de los cálculos sean lo más exactos posibles. En primer lugar, se estima la cantidad de espacio utilizado por la cabecera de bloque; éste es el espacio que utiliza Oracle para administrar los datos contenidos en el bloque. El tamaño de la cabecera de bloque es de 161 bytes, aproximadamente. Si se utilizan 2k para el bloque, quedarían 1887 bytes libres, mientras que si se utilizan 4K, quedarían 3935 bytes libres.

2048 – 161 = 1887 bytes disponibles

A continuación se multiplica el espacio libre por el factor determinado por el parámetro pctfree de la tabla, para determinar la cantidad de espacio que se va a mantener libre para actualizar los registros. Si se utilizó un valor pctfree de 10, habría que multiplicar el espacio libre disponible por .10, como se muestra a continuación:

1887 – 189 = 1698

De los 2048 bytes del bloque, 1698 se reservan para almacenar los elementos del índice. El siguiente paso consiste en calcular el espacio utilizado por fila. Para hacerlo hay que estimar primero la longitud media del registro de las columnas del índice. En el caso de las tablas, se calcula la longitud media del registro de todas las columnas; en los índices sólo hay que preocuparse de las columnas indexadas. La longitud media del registro es la suma de las longitudes medias de las longitudes indexadas. Y se calcula de la misma manera que con las tablas, sólo que la cabecera del índice utiliza 8 bytes. Supongamos que la longitud media del registro indexado es de 16 bytes, se tienen dos columnas de tipo VARCHAR2(10) y si en el índice contiene datos con una longitud de más de 127 caracteres, se añade 1 byte. El espacio utilizado por registro se denota por:

Espacio utilizado por registro = Longitud_media_x_fila
+ Número de columnas
+ Número de columnas largas
+ 8 bytes de cabecera

En el índice de ejemplo el espacio utilizado por registro es:

Espacio utilizado por registro = 16
+ 2
+ 0
+ 8
= 26 bytes por elemento del índice

Si el índice es único, el espacio por registro será:

26 + 1 = 27 bytes por fila

Dado que hay 1698 bytes disponibles y 27 bytes por registro, se pueden incluir 62 elementos de índice en un bloque:

Elementos por bloque = 1698 bytes libre / 27 por elemento)
= 62 elementos por bloque

Ya que se pueden incluir 62 elementos de índice por bloque, se puede estimar el número de bloques necesarios, siempre y cuando se pueda estimar también el número de registros que se espera que haya. A medida que se vayan manipulando los registros de la tabla, los requisitos de espacio irán aumentando. Cuanto mayor sea el número de elementos borrados o actualizados, mayor cantidad de espacio se necesitará.

Casi nunca se vuelve a utilizar el espacio borrado de los índices, por lo que dichos índices pueden crecer aunque la tabla no lo haga. Por ejemplo, si se borran 100 registros y luego se insertan 100 más, puede que la tabla utilice el espacio que han dejado libre los registro borrados para los nuevos registros que se han insertado, por lo que el espacio utilizado sería constante. No obstante, lo más probable es que el índice de la tabla no sea capaz de volver a utilizar el espacio liberado por los registros liberados, por lo que el espacio utilizado aumentará.

Diseño lógico de la base de datos

La configuración lógica de la base de datos influye enormemente en su rendimiento y en la facilidad de su administración. La primera persona que formalizó la distribución efectiva de los objetos lógicos de una base de datos fue Cary Millsap, de Oracle, quién denominó a la arquitectura resultante arquitectura flexible óptima (Optimal Flexible Architecture, OFA). La disposición OFA estándar de tablespaces se crea automáticamente al software de instalación de Oracle.

Arquitectura flexible óptima (OFA: Optimal Flexible Architecture)

Esta arquitectura tiene su base en la organización de los objetos lógicos de la base de datos en tablespaces (espacio de tablas). El tablespace principal es llamado: SYSTEM.

Tablespace System

Aunque no es recomendable, es posible almacenar todos los objetos de la base de datos en un solo tablespace, esto equivaldría a almacenar todos los archivos de una computadora en el directorio raíz. El tablespace SYSTEM, que es el equivalente en Oracle a un directorio Raíz es el lugar donde se guardan las tablas del diccionario de datos (propiedad del usuario SYS). Aquí se encuentran también los segmentos Rollback (deshacer) SYSTEM. No hay ninguna razón para almacenar en el tablespace SYSTEM cualquier otra cosa que no sea las tablas del diccionario de datos y el segmento de rollback SYSTEM. Para impedir que los usuarios creen objetos en el espacio de tablas SYSTEM, hay que revocar toda cuota de espacio sobre SYSTEM, de la siguiente manera:

ALTER USER *nombre_usuario* QUOTA 0 ON *system*;

Cuando se crea un usuario se puede especificar un espacio de tablas predeterminado:

CREATE USER *nombre_usuario* IDENTIFIED BY *password*
DEFAULT TABLESPACE *algun_tablespace*;

Si el usuario ya ha sido creado, puede emplearse la orden siguiente para asignar un nuevo tablespace predeterminado.

ALTER USER *nombre_usuario* DEFAULT TABLESPACE *algun_tablespace*;

Tablespace Data

Los segmentos de datos, son las áreas físicas en las que se almacenan los datos asociados con las tablas y agrupaciones. La base de datos acceda a estos segmentos con mucha frecuencia, por lo que experimentan un gran número de transacciones de manipulación de datos. El principal objetivo de una base de datos de producción es administrar las solicitudes de acceso a los segmentos de datos. Normalmente un tablespace DATA contiene todas las tablas asociadas a una aplicación. El alto volumen de tráfico que registran estas tablas las convierten en candidatas ideales para aislarlas en su propio tablespace. Si se aíslan las tablas de la aplicación en un tablespace DATA, se pueden separar los archivos de datos de ese tablespace del resto de los archivos de datos de la base de datos. Esta separación de los archivos de datos en distintas unidades de disco puede mejorar el rendimiento, gracias a que la contienda por los recursos E/S es menor y simplifica la administración de los archivos.

Tablespace Indexes

Los índices asociados a tablas están sujetos a los números problemas E/S y crecimiento/fragmentación que hacen aconsejable sacar los segmentos de datos del tablespace SYSTEM. No conviene almacenar los segmentos de índices en el mismo tablespace que las tablas a las que estén asociados, ya que registran un gran volumen de operaciones de E/S concurrentes durante la manipulación de datos y las consultas. Los segmentos de índice también son objeto de fragmentación debida a un dimensionamiento inadecuado o a un crecimiento de la tabla imprevisto. El aislamiento de los índices de la aplicación en un tablespace independiente reduce considerablemente las labores administrativas relacionadas con la defragmentación de los tablespaces DATA o INDEXES.

Para separar los índices existentes de sus tablas, puede utilizarse la opción REBUILD de la orden ALTER INDEX. Si se crea un índice en el mismo tablespace que la tabla que indexa, puede sacarse del tablespace con una sentencia SQL. En el siguiente ejemplo el índice idx_user se desplaza al tablespace INDEXES y se le asignan nuevos valores de almacenamiento con la cláusula STORAGE.

```
ALTER INDEX idx_user
TABLESPACE indexes
STORAGE (INITIAL 2M NEXT 2M PCTINCREASE 0)
```

Tablespace Tools

A pesar de lo que se menciona anteriormente de no almacenar segmentos de datos en el tablespace SYSTEM, hay muchas herramientas que hacen precisamente eso. No lo hacen porque necesiten que sus objetos se almacenen en el tablespace SYSTEM, si no porque los almacenan en la cuenta SYSTEM de la base de datos, que normalmente tiene asignado el tablespace SYSTEM como área predeterminada para almacenar objetos. Para evitarlo basta con cambiar el tablespace predeterminado de la cuenta SYSTEM por el tablespace. Muchas herramientas de Oracle y de otros fabricantes crean tablas propiedad de SYSTEM. Si estas herramientas ya han creado las tablas en la base de datos, sus objetos pueden cambiarse de lugar exportando la base de datos, eliminando las tablas antiguas de las herramientas, revocando la cuota de la cuenta del tablespace SYSTEM, limitando la cuota del usuario SYSTEM al tablespace TOOLS e importando las tablas.

Tablespace RBS

Los segmentos de rollback mantienen la concurrencia de los datos. Para crear otros espacios de tablas distintos de SYSTEM, en primer lugar hay que crear un segundo segmento de Rollback en el tablespace SYSTEM. Para aislar los segmentos de Rollback (que realizan operaciones E/S para las transacciones de la base de datos) del diccionario de datos, hay que crear un tablespace de segmento de Rollback que sólo contenga segmentos de Rollback. Este modo de separarlos también simplifica su administración. Una vez que se ha creado el tablespace RBS y que se ha activado un segmento de Rollback dentro del mismo se puede anular el segundo segmento de Rollback del tablespace SYSTEM. Tal vez prefiera mantener inactivo este segundo segmento de rollback en SYSTEM y no deshacerse de él por si ocurriera algún problema con el tablespace RBS. Los segmentos de Rollback se amplían dinámicamente hasta adquirir el tamaño de la transacción más grande y se contraen hasta adquirir un tamaño óptimo especificado. Las operaciones de E/S que afectan a los segmentos de Rollback suelen ser concurrentes con las operaciones de E/S que afectan a los espacios de tablas DATA e INDEXES. Su separación ayuda a evitar la contienda de E/S y los hace más fáciles de administrar.

Tablespace TEMP

Los segmentos temporales son objetos que se crean dinámicamente en la base de datos y que almacenan datos durante las operaciones de ordenaciones de gran tamaño (como select, distinct, union y create index). Dada su naturaleza dinámica, no conviene almacenar los segmentos temporales junto con ningún otro tipo de segmentos. Cuando un tablespace TEMP se encuentra inactivo no almacena ningún segmento. El hecho de separarlos segmentos de SYSTEM elimina un potencial problema del área del diccionario de datos y crea un tablespace fácil de administrar. Para especificar un tablespace temporal distinto de SYSTEM, se puede emplear la orden CREATE USER, tal como se muestra enseguida:

```
CREATE USER nombre_usuario IDENTIFIED BY password
DEFAULT TABLESPACE algun_tablespace
TEMPORARY TABLESPACE temp;
```

Si la cuenta ya ha sido creada puede utilizarse la orden:

```
ALTER USER nombre_usuario TEMPORARY TABLESPACE temp;
```

Tablespace USERS

Aunque los usuarios no suelen tener derechos de creación de objetos en las bases de datos de producción, si pueden tenerlos en las bases de datos de desarrollo. Los objetos de usuario suelen ser de naturaleza transitoria y no estar bien dimensionados. Por esto, conviene separar estos objetos del resto de la base de datos. De esta manera se contribuye a minimizar la influencia de los experimentos de los usuarios en el funcionamiento de la base de datos. Para realizar esta operación hay que revocar las cuotas de los usuarios sobre otros espacios de tablas y cambiar su tablespace predeterminado por el tablespace USERS. Para especificar un tablespace predeterminado alternativo pude utilizarse la orden CREATE USER:

```
CREATE USER nombre_usuario IDENTIFIED BY password
DEFAULT TABLESPACE users
TEMPORARY TABLESPACE temp;
```

Si la cuenta ya ha sido creada puede utilizarse la orden:

```
ALTER USER nombre_usuario DEFAULT TABLESPACE users;
```

Otros tablespaces

Además de los tablespaces anteriores, es posible agregar otros, para hacer aun más personalizada la configuración de los tablespaces de la base de datos. Esto es aislar los objetos que tienen distintos requisitos de actualización.

Segmentos de datos de poco uso. Tablespace DATA_2

Al revisar la lista de tablas de datos, es probable que puedan clasificarse fácilmente en dos o más grupos en función de sus características: algunas contendrán datos muy dinámicos y otras muy estáticos. Estas últimas pueden contener una lista de países, por ejemplo. Las tablas de datos estáticos suelen experimentar menos operaciones de E/S que las tablas de datos activas. Cuando se consultan, el acceso a una tabla de datos estáticos suele ser concurrente con un acceso a una tabla de datos dinámicos.

Esta E/S concurrente puede distribuirse entre varios archivos y por tanto en múltiples discos para mejorar el rendimiento, colocando todas las tablas de datos estáticos en un tablespace dedicado.

Segmentos de índice de poco uso. Tablespace INDEXES_2

Los índices de tablas estáticas de poco uso son de la misma naturaleza pasiva. Para simplificar las labores administrativas que requiere el tablespace INDEXES, conviene colocar los índices de las tablas estáticas en un tablespace INDEXES_2 independiente. Esto mejora también el rendimiento de las opciones de optimización, ya que las operaciones de E/S concurrentes entre índices pueden ahora dividirse en varias unidades de disco. Si los índices de poco uso ya se han creado en el tablespace INDEXES, entonces hay que eliminarlos y volver a crearlos en INDEXES_2. Esto se debe realizar al mismo tiempo que se hace el traslado de las tablas de poco uso a DATA_2. En caso de que el índice haya sido creado mediante una definición de restricción UNIQUE (único) o PRIMARY KEY (llave primaria), habrá que modificar dicha restricción. En el siguiente ejemplo se crea restricción unívoca (UNIQUE) sobre una columna llamada '*Descripción*' de una tabla estática denominada '*TIPO_EMPLEADO*'. El índice

único que creará la base de datos para esta restricción se almacenará en el tablespace INDEXES_2.

```
ALTER TABLE tipo_empleado
  ADD CONSTRAINT pk_desc UNIQUE(descripcion)
USING INDEX TABLESPACE INDEXES_2;
```

En el caso de que el índice ya exista, se puede trasladar de su tablespace actual a uno nuevo mediante la cláusula REBUILD de la orden ALTER INDEX.

Segmentos de índice de herramientas. Tablespace TOOLS_2

Si se observa demasiada actividad en el espacio de tablas TOOLS, se pueden trasladar los índices de las tablas de herramientas a otro tablespace. Esto resulta espacialmente útil en aquellos entornos en los que el tablespace TOOLS se trata como un tablespace DATA; es decir, si sus tablas son objeto de muchas de las operaciones de E/S de la base de datos. Para separar los índices de las tablas se sigue el mismo procedimiento que en el tablespace INDEXES.

Segmentos de anulación especializados. Tablespace RBS_2

Para que los segmentos de rollback del tablespace RBS permitan el uso de la aplicación en el entorno de producción, han de ser del tamaño y del número adecuados. Casi siempre habrá una transacción de un tamaño que la configuración del segmento de rollbak no admita. Cuando se ejecute, esta transacción ocupará uno de los segmentos de rollback de producción y lo ampliará en gran medida, utilizando tanto espacio libre como pueda, antes de que la transacción se complete o falle. Esto no tiene porqué suceder. Los segmentos de rollback de producción deberían ser utilizados por los usuarios del entorno de producción. Los requisitos especiales de transacción los maneja un segmento de rollback independiente. Para especificar este segmento de rollback, el usuario debe emplear esta instrucción:

```
SET TRANSACTION USE ROLLBACK SEGMENT nombre_segmento;
```

Antes de ejecutar la transacción. Aunque esto sólo resuelve una parte del problema, ya que el segmento de anulación seleccionado sigue ocupando espacio en el tablespace del segmento de rollback de producción. Conviene crear un tablespace de rollback distinto, cuya única función sea dar servicio a este tipo de transacciones, que por lo general son transacciones de una gran carga de datos. Cuando se completa la transacción, el segmento de rollback puede desactivarse o eliminarse. Una vez más el hecho de separar los objetos lógicos según sus requisitos funcionales sirve para simplificar enormemente su administración y mejorar el rendimiento.

Segmentos temporales específicos de usuario. Tablespace TEMP_USUARIO

El tablespace principal final es, como RBS_2, un tablespace especial, diseñado para resolver las necesidades específicas de los usuarios de la aplicación. Determinados usuarios pueden requerir segmentos temporales mucho mayores que el resto de los usuarios de la aplicación. En tal caso es conveniente separar estos segmentos temporales del tablespace TEMP estándar. Es conveniente nombrar este tablespace temporal de acuerdo con el nombre del usuario o grupo de usuarios. Para especificar un espacio de tablas temporal para un usuario, se puede utilizar la orden CREATE USER:

```
CREATE USER nombre_usuario IDENTIFIED BY password
DEFAULT TABLESPACE nombre_tablespace
TEMPORARY TABLESPACE temp._usuario;
```

Diseño lógico de sentido común

El diseño lógico de la base de datos resultante debe cumplir los siguientes criterios:

- Los tipos de segmentos que se utilicen de la misma forma deben almacenarse juntos.
- El sistema debe diseñarse para una utilización estándar.
- Deben existir áreas independientes para las excepciones.
- Debe minimizarse la contienda entre los tablespaces.
- El diccionario de datos ha de estar aislado.

Para que se cumplan estos criterios, el DBA debe conocer la aplicación que se esté implementando: que herramientas utilizará, cuáles serán las tablas más activas, cuando tendrán las cargas de datos, que usuarios van a necesitar mayores recursos y cómo se comportarán las transacciones estándar. Para adquirir estos conocimientos el DBA debe estar muy integrado en el proceso de desarrollo. Si se cumplen estos criterios, el resultado es un sistema cuyos tipos de segmentos no interfieren entre si. Esto simplifica mucho la administración de la base de datos, el aislamiento y resolución de problemas de rendimiento. Si la base de datos esta diseñada de esta forma, cuando se produzca una fragmentación de segmentos o de espacio libre la solución será mucho más sencilla.

La combinación de un buen diseño lógico de la base de datos en un diseño físico eficiente produce unos sistemas que requieren muy poca optimización después de la primera comprobación antes de pasar a producción. Las labores de planificación previa tienen su recompensa en la flexibilidad y el rendimiento de la base de datos. El costo de implementar este diseño desde el principio es mínimo; puede incorporarse en todos los scripts de la base de datos de forma automática y constituye ya una parte de los scripts de creación que genera automáticamente el proceso de instalación de Oracle.

Diseño físico de la base de datos

Con frecuencia no se planifica el diseño físico de la base de datos y sólo se tiene en cuenta cuando se comienzan a experimentar problemas de rendimiento. Al igual que es necesario planear el diseño lógico, también hay que diseñar e implementar la disposición física de los archivos de la base de datos. Si no se planifica esta disposición antes de crear la base de datos, se producirá un ciclo repetitivo de problemas relacionados con el diseño y las tareas de optimización del rendimiento.

Disposición de los archivos de la base de datos.

Es importante establecer objetivos claros para el diseño de la distribución de archivos y la comprensión de la naturaleza de la base de datos (por ejemplo, las bases de datos orientadas a transacciones, frente a las bases de datos de lectura intensiva) permite determinar cuál es el diseño adecuado para distribuir los archivos entre cualquier número de dispositivos. Este proceso se llevará a cabo por medio de los siguientes pasos:

- Identificación de la contienda de E/S entre los archivos de datos.
- Identificación de los cuellos de botella de E/S entre todos los archivos de la base de datos.
- Identificación de las operaciones de E/S concurrentes entre procesos.
- Definición de los objetivos de rendimiento y seguridad de la base de datos.
- Definición del hardware del sistema y de la arquitectura de duplicación en espejo.
- Identificación de los discos que se pueden identificar de la base de datos.

En la mayoría de los casos, antes de crear la base de datos sólo se llevan acabo las tareas relacionadas con la contienda de archivos de datos, duplicación en espejo del hardware y

adquisición de discos, por lo que se esta introduciendo la posibilidad de contienda en el diseño del sistema. Al llevar a cabo todos los pasos que se mencionan anteriormente, el producto final ha de ser un diseño físico de bases de datos adaptado a las necesidades particulares de cada sistema. Cada espacio lógico de la base de datos necesita de un archivo físico para existir y la correcta distribución de estos en el servidor de base de datos hace una gran diferencia en el buen rendimiento del sistema. Enseguida se muestra una solución de diseño físico ideal con 22 discos, y se mostrarán más diseños con menos discos pero con la misma idea de mejorar el rendimiento de la base de datos. Se da por hecho que los discos que se utilizan están dedicados exclusivamente a la base de datos, además de que tienen el mismo tamaño y las mismas características de rendimiento.

Solución ideal de 22 discos

No es probable que se disponga de la configuración del la tabla 3.5, pero es importante comenzar con objetivos que apunten a esto. Con esta configuración se consigue eliminar por completo la contienda entre archivos de datos, proporcionándole a cada uno de ellos un disco diferente. También se elimina la contienda entre los procesos de segundo plano LGRW y ARCH, proporcionando también a cada registro de rollback independiente, además se asigna un disco al software de la aplicación que va a acceder a la base de datos. Es bastante improbable que se dé esta configuración debido a la cantidad tan importante de recursos que requiere; en los discos de los archivos de control se utiliza un disco entero (normalmente de más de 1 GB) para mantener un solo archivo, al que se accede poco y que rara vez excede los 200 K. Para conseguir una configuración más realista, hay que revisar de manera iterativa la disposición de discos hasta llegar al número de discos disponibles.

Disco	Contenido
1	Software de Oracle
2	Tablespace SYSTEM
3	Tablespace RBS
4	Tablespace DATA
5	Tablespace INDEXES
6	Tablespace TEMP
7	Tablespace TOOLS
8	Registro de rollback 1
9	Registro de rollback 2
10	Registro de rollback 3
11	Archivo de control 1
12	Archivo de control 2
13	Archivo de control 3
14	Software de aplicación
15	Tablespace RBS_2
16	Tablespace DATA_2
17	Tablespace INDEXES_2
18	Tablespace TEMP_*USUARIO*
19	Tablespace TOOLS_2
20	Tablespace USERS
21	Disco de destino del registro de rehacer archivado
22	Disco de destino del archivo de volcado de exportación

Tabla 3.5 Solución de 22 discos

Solución de 17 discos.

Cada iteración sucesiva del diseño de los discos implicará la colocación del contenido de varios discos en uno sólo. En la primera iteración de la tabla 3.6 se trasladan los tres archivos de

control a los tres discos de los registros de rollback. Los archivos de control provocarán una contienda de interferencia con los registros de rollback, pero únicamente en los puntos de conmutación de registro y durante la recuperación de la base de datos. Durante el funcionamiento de la base de datos la interferencia será mínima. Dando por hecho que se esta trabajando en una base de datos de producción, los contenidos del tablespace TOOLS_1 se combinarán con el tablespace TOOLS (normalmente sólo pueden estar separados en entornos de desarrollo intenso). En los entornos de producción, los usuarios no disponen de privilegios sobre los recursos, por lo que en estas configuraciones no se tendrá en cuenta el tablespace USERS.

Disco	Contenido
1	Software de Oracle
2	Tablespace SYSTEM
3	Tablespace RBS
4	Tablespace DATA
5	Tablespace INDEXES
6	Tablespace TEMP
7	Tablespace TOOLS
8	Registro de rollback 1, Archivo de control 1
9	Registro de rollback 2, Archivo de control 2
10	Registro de rollback 3, Archivo de control 3
11	Software de aplicación
12	Tablespace RBS_2
13	Tablespace DATA_2
14	Tablespace INDEXES_2
15	Tablespace TEMP_*USUARIO*
16	Disco de destino del registro de rehacer archivado
17	Disco de destino del archivo de volcado de exportación

Tabla 3.6 Solución de 17 discos

Solución de 15 discos

La segunda iteración de combinación de archivos que se muestra en la tabla 3.7, es la que da comienzo al proceso de colocación de múltiples espacios de tablas en el mismo disco. En este caso se han colocado se han colocado juntos los tablespaces RBS y RBS_2, porque no se suelen utilizar de manera concurrente; como se ha definido previamente, el tablespace RBS_2 contiene segmentos de rollback especiales que se utilizan durante las cargas de gran tamaño. Dado que no debería tener lugar ninguna carga de datos durante el uso en producción (para lo cual se utiliza el tablespace RBS), no debería producirse ninguna contienda entre RBS y RBS_2, por lo que se pueden colocar juntos.

Los tablespaces TEMP y TEMP_*USUARIO* también se pueden colocar en el mismo disco. El tablespace TEMP_*USUARIO* está dedicado a un usuario en concreto, que necesita muchos más segmentos temporales que el resto de los usuarios. El peso del tablespace TEMP, puede variar considerablemente; no obstante debería ser posible guardarlo en el mismo dispositivo en el que se encuentra el tablespace TEMP_*USUARIO*, sin que esto afecte demasiado a sus operaciones de E/S.

Disco	Contenido
1	Software de Oracle
2	Tablespace SYSTEM
3	Tablespace RBS, Tablespace RBS_2
4	Tablespace DATA
5	Tablespace INDEXES
6	Tablespace TEMP, Tablespace TEMP_*USUARIO*
7	Tablespace TOOLS
8	Registro de rehacer en línea 1, Archivo de control 1
9	Registro de rehacer en línea 2, Archivo de control 2
10	Registro de rehacer en línea 3, Archivo de control 3
11	Software de aplicación
12	Tablespace DATA_2
13	Tablespace INDEXES_2
14	Disco de destino del registro de rehacer archivado
15	Disco de destino del archivo de volcado de exportación

Tabla 3.7 Solución de 15 discos

Solución de 12 discos

Antes de seguir combinando tablespaces en un mismo disco, deberíamos colocar en el mismo disco los registros de rehacer en línea (tabla 3.8). En las bases de datos que utilicen copias de seguridad ARCHIVELOG, esto provocará una contienda de interferencia y de E/S concurrente entre los procesos de segundo plano LGWR y ARCH en dicho disco. En consecuencia, esta combinación no resulta adecuada para aquellos sistemas en los que haya un volumen muy elevado de transacciones y que se ejecuten en modo ARCHIVELOG. Dado que los discos de los registros de rehacer en línea se han combinado en uno sólo, hay que desplazar los archivos de control. En este ejemplo, los archivos de control coexisten con los tres espacios de tablas más importantes (SYSTEM, RBS y DATA). Como ya hemos mencionado, los archivos de control no tienen un gran volumen de operaciones de E/S, por lo que provocarán muy pocas contiendas. Otro cambio que hay en esta configuración es la combinación del tablespace TOOLS con el tablespace INDEX_2, que también generarán pocas contiendas por el tipo de información que contienen.

Disco	Contenido
1	Software de Oracle
2	Tablespace SYSTEM, Archivo de control 1
3	Tablespace RBS, Tablespace RBS_2, Archivo de control 2
4	Tablespace DATA, Archivo de control 3
5	Tablespace INDEXES
6	Tablespace TEMP, Tablespace TEMP_*USUARIO*
7	Tablespace TOOLS, Tablespace INDEXES_2
8	Registro de rehacer en línea 1,2 y 3
9	Software de aplicación
10	Tablespace DATA_2
11	Disco de destino del registro de rehacer archivado
12	Disco de destino del archivo de volcado de exportación

Tabla 3.8 Solución de 12 discos

Solución de 9 discos

La cuarta iteración combina los tres discos que tienen la numeración más alta (10, 11, 12) con los que mejor concuerdan con sus características (tabla 3.9). En primer lugar, el

tablespace DATA_2 se combina con los tablespaces TEMP, creando un disco que manejará aproximadamente el 4% de la E/S de archivos de datos. En segundo lugar se han desplazado los archivos de volcado de exportación al disco de registro de rehacer en línea. Los archivos de rehacer en línea nunca aumentan de tamaño y el proceso de exportación de una base de datos provoca un nivel de actividad de transacción muy bajo. La tercera combinación de esta iteración es la del software de la aplicación, con la del área de destino del archivo de registro de rehacer archivado. Se da por hecho que el software de la aplicación es estático y de tamaño pequeño, empleando menos del 10% del espacio del disco disponible. Esto deja un espacio muy amplio al proceso de segundo plano ARCH para escribir los archivos de registro, a la vez que se evitan los conflictos con el proceso DBWR.

Disco	Contenido
1	Software de Oracle
2	Tablespace SYSTEM, Archivo de control 1
3	Tablespace RBS, Tablespace RBS_2, Archivo de control 2
4	Tablespace DATA, Archivo de control 3
5	Tablespace INDEXES
6	Tablespace TEMP, Tablespace TEMP_*USUARIO*, Tablespace DATA_2
7	Tablespace TOOLS, Tablespace INDEXES_2
8	Registro de rehacer en línea 1,2 y 3, Disco de destino del archivo de volcado de exportación
9	Software de aplicación, Disco de destino del registro de rehacer archivado

Tabla 3.9 Solución de 9 discos

Solución de 7 discos

A partir se este punto, las combinaciones de espacios de tablas deben basarse en los pesos asignados durante un proceso de estimación de carga de E/S. En la tabla 3.10 se muestra la distribución de los pesos de E/S junto a los discos después de la cuarta iteración:

Disco	Peso	Contenido
1		Software de Oracle
2	35	Tablespace SYSTEM, Archivo de control 1
3	40	Tablespace RBS, Tablespace RBS_2, Archivo de control 2
4	100	Tablespace DATA, Archivo de control 3
5	33	Tablespace INDEXES
6	9	Tablespace TEMP, Tablespace TEMP_*USUARIO*, Tablespace DATA_2
7	3	Tablespace TOOLS, Tablespace INDEXES_2
8	40+	Registro de rehacer en línea 1,2 y 3, Disco de destino del archivo de volcado de exportación
9	40+	Software de aplicación, Disco de destino del registro de rehacer archivado

Tabla 3.10 Pesos de E/S estimados de la solución de 9 discos.

Se debe tomar en cuenta que el peso de 100 asignado al disco donde se encuentra el tablespace DATA, significa que es este tablespace el que más accesos va a tener y tomando como base a este se distribuye el peso en los demás discos. La razón de que no aparezca el peso del disco 1 reside en que se trata de un disco específico de la instalación, ya que las aplicaciones pueden variar considerablemente en tamaño y las versiones del manejador también. Los pesos del disco 8 y 9 se basan en el peso de los tablespaces de rollback, ya que las transacciones que se escriben en RBS también se escribirán en los archivos de registro de rollback. Si se está ejecutando la base de datos en modo ARCHIVELOG, los archivos de registro rehacer archivados del disco 9 tendrán la misma carga E/S que los archivos de registro de

rollback del disco 8. Dado que en estos discos hay otros archivos (los archivos de volcado de exportación y software de aplicación), se asigna a su peso aun valor algo superior al peso de E/S del disco RBS. A partir de los pesos que se muestran en la tabla 3.9, ya no se puede llevar a cabo ninguna otra combinación que sea buena. Para comprimir más el conjunto de discos habría que guardar los datos en el mismo disco de los índices asociados a ellos (combinando los discos 6 y 7, de DATA_2 e INDEXES_2, respectivamente), o bien habría que guardar más tablespaces en uno de los cuatro discos que tienen más peso (los discos 2,3,4 y 5). Los últimos dos discos, que se están usando para dar soporte a los archivos de rollbak, las exportaciones, el software de la aplicación y los archivos de registro de rollbak archivados, que son cruciales para la recuperación de la base de datos, no deberían soportar más carga. El peso de E/S de la quinta iteración da como resultado la distribución de archivos que se muestran en la tabla 3.11.

Disco	Peso	Contenido
1		Software de Oracle
2	38	**Tablespace SYSTEM, TOOLS, INDEXES_2, Archivo de control 1**
3	40	**Tablespace RBS, RBS_2, Archivo de control 2**
4	100	**Tablespace DATA, Archivo de control 3**
5	42	Tablespace INDEXES, TEMP, TEMP_USUARIO, DATA_2
6	40+	Registro de rehacer en línea 1,2 y 3, Disco de destino del archivo de volcado de exportación
7	40+	Software de aplicación, Disco de destino del registro de rollback archivado

Tabla 3.11 Solución intermedia de 11 discos.

En esta iteración, los tablespaces TOOLS e INDEXES_2 de desplazan desde el disco 7 al disco que contiene el tablespace SYSTEM. Los tablespaces TEMP, TEMP_*USUARIO* y DATA_2 se desplazan desde el disco 6 hasta aquel en el que se encuentra el tablespace INDEXES (dado que los segmentos temporales se amplían dinámicamente, hay que mantenerlos separados de los espacios de tablas SYSTEM). Los archivos de los tablespaces de la base de datos se muestran en negrita en la tabla 3.11 y ahora están repartidos en cuatro discos (los discos 2, 3, 4 y 5). Cada uno de estos cuatro discos contiene uno de los cuatro archivos que tienen más peso de E/S en la base de datos; sus pesos relativos serán los mismos en la mayoría de las bases de datos. Si los sistemas tienen un volumen de transacciones muy elevado, este diseño no cambiará, ya que los tablespaces de rollback (RBS y RBS_2) ya están aislados.

Para ir más allá de este nivel de combinación de archivos, el DBA tiene que llegar a compromisos adicionales. Dado que no se debe comprometer la recuperabilidad de la base de datos, los discos 6 y 7 deberían permanecer tal y como están. Cualquier combinación adicional de los archivos de tablespace afectaría al rendimiento. Por tanto, las combinaciones adicionales de archivos de datos tienen que basarse en medidas reales de carga de E/S de dichos archivos de datos. Una opción de distribución de la información en sistemas austeros, es simplemente utilizar tres discos y distribuir en uno de ellos las tablas e índices de los datos que cambian constantemente (dinámicos), en otro los objetos de los datos estáticos y en otro los programas y archivos de control, de esta manera se tiene una configuración mínima pero eficaz de los discos que utiliza el RDBMS.

3.3 Métodos de acceso a los datos

Los métodos de acceso son formas de obtener la información almacenada, que mediante un uso adecuado pueden mejorar el rendimiento. Se pueden usar sugerencias en las consultas para forzar el tipo de accesos. A continuación se describirán los métodos de acceso más comunes.

Uso de clusters (grupos)

Para utilizar una forma de accesar la información es necesario comprender su funcionamiento, sus ventajas y desventajas. En este caso se mencionan las consideraciones para crear un cluster:

- Se debe considerar agrupar una tabla si es accesada muchas veces por la aplicación en sentencias que incluyan joins.
- No agrupar tablas si la aplicación sólo realiza joins ocasionalmente o modifica sus columnas comunes constantemente. Modificar la llave de una tabla agrupada toma más tiempo que modificar la llave de una tabla no agrupada, porque Oracle tiene que migrar el renglón modificado a otro bloque para mantener el cluster (grupo).
- No agrupar tablas si la aplicación realiza muchas veces "full table scans" o accesos completos a la tabla de una de ellas. El acceso completo a la tabla agrupada toma más tiempo que en una tabla no agrupada. Es probable que Oracle lea más bloques cuando las tablas están agrupadas.
- Considerar la agrupación en tablas del tipo maestro detalle, si se seleccionan muchas veces registros del catalogo maestro y sus correspondientes registros detalle. Dado que los registros detalle están almacenados en el mismo bloque o bloques de datos con los registros maestros, son accesados de manera conjunta y se realizan menos operaciones de E/S.
- Considerar la agrupación en tablas del tipo maestro detalle, si comúnmente se seleccionan registros detalle del mismo maestro. Esta medida mejora el rendimiento de consultas que seleccionan registros detalle del maestro pero no disminuye el rendimiento de un acceso completo a la tabla maestra.
- No agrupar tablas, si los datos de todas las tablas con el mismo valor de la llave de grupo ("cluster key") exceden más de uno o dos bloques. Para accesar un registro de una tabla, Oracle lee todos los bloques que contengan registros con este valor de la llave. Si estos renglones toman múltiples bloques, accesar un registro requiere más lecturas que accesar el mismo registro de una tabla no agrupada.

  Se deben considerar los beneficios y las ventajas de los grupos con respecto a las necesidades de la aplicación. Por ejemplo, se puede decidir que el rendimiento obtenido por las sentencias del tipo join es mayor que el rendimiento perdido por las sentencias que modifican las llaves del grupo. Se puede buscar, experimentar y comparar los tiempos de los procesos con tablas agrupadas y con tablas almacenadas de manera separada. Para crear un grupo se utiliza el comando CREATE CLUSTER.

Uso de Hash clusters (grupos Hash)

Los grupos Hash se basan en el resultado de aplicar la función Hash a cada registro de la llave de grupo. Todos los registros con la misma llave de grupo son almacenados juntos en el disco. Considere las ventajas y desventajas de los grupos Hash con respecto a las necesidades de su aplicación. Se puede probar el rendimiento con un grupo de tipo Hash o sólo con una tabla y su índice; para observar cual se acerca más a nuestras necesidades.

Cuando usar un Hash cluster

- Cuando las tablas son continuamente accesadas por sentencias SQL con cláusulas WHERE que contengan condiciones de igualdad y usen la misma columna o combinación de columnas. Designar esta columna o combinación de columnas como la llave de grupo.
- Si es posible determinar cuanto espacio es requerido para contener todos los registros con la llave de grupo incluida., incluyendo los registros que van a ser insertados inmediatamente como los registros que se insertarán en el futuro.
- No utilizar si el espacio es escaso y no se puede proporcionar espacio adicional para insertar registros en el futuro.
- No utilizar en una tabla de constante crecimiento, dado que no es práctico utilizar un grupo Hash en una tabla demasiado grande que se tenga que actualizar constantemente.
- No utilizar si se realizan en las tablas frecuentemente accesos completos. Porque en un acceso completo se deben leer todos los bloques que integran el grupo Hash, aunque algunos bloques sólo contengan algunos registros. En una tabla almacenada independientemente se reduce el número de bloques leídos en un acceso completo a la tabla.
- No utilizar si la aplicación frecuentemente modifica los valores de la llave de grupo, esto es más costoso que actualizar los valores de la llave de tablas no agrupadas.

Como usar un Hash cluster

Para crear un hash cluster, se usa el comando CREATE CLUSTER con los parámetros HASH y HASHKEYS. Cuando se crea un hash cluster, se debe usar el parámetro HASHKEYS para especificar el número de valores del grupo hash. Para mejor rendimiento de las búsquedas de este tipo, se debe escoger un valor de HASHKEY tan grande como el número de valores de la llave de grupo.

Uso de Anti-Joins

Un anti-join es una forma de join (un join es la operación de reunión entre las tablas) con lógica inversa. En lugar de regresar los registros que cumplen con el predicado del join entre los lados izquierdo y derecho, un anti-join regresa los registros del lado izquierdo que no están en la tabla del lado derecho del join. Este comportamiento es exactamente el mismo de un NOT IN y una subconsulta, cuando el lado derecho del predicado del anti-join corresponde con la subconsulta.

Cuando usar un Anti-Join

Un anti-join usa sort-merge o hash joins para evaluar la subconsulta NOT IN. Se asume que el predicado de la subconsulta es de la siguiente forma: (colA1, colA2, ... colAn) NOT IN (SELECT colB1, colB2, ..., colBn FROM ...). Se deben cumplir las siguientes para que la subconsulta sea transformada en un anti-join.

- Todas las referencias a las columnas en A deben hacer referencia simples a columnas. Las referencias a las columnas en el B deben o ser referencias simples a columnas o funciones agregadas (MIN, MAX, SUMA, COUNT, o AVG) aplicadas directamente a una columna simple, si la subconsulta contiene una cláusula GROUP BY. No se permiten otras expresiones.
- Todas las referencias de columna no deben contener valores nulos.
- La cláusula WHERE no debe tener OR's en el nivel lógico más alto.
- Los Anti-joins sólo se usan con el enfoque basado en costos del optimizador.

Como usar un Anti-join

Oracle transforma las subconsultas NOT IN en anti-joins si se cumplen los puntos anteriores y si hay un parámetro de inicialización o sugerencia en la sentencia indicando que la transformación tenga lugar. Para una consulta especifica, poner las sugerencias: MERGE_AJ o HASH_AJ en la subconsulta. MERJE_AJ usa el anti-joing normal y HASH_AJ usa un hash anti-joing. Por ejemplo:

```
SELECT * FROM empleados
        WHERE nombre LIKE 'J%' AND
                dep_id IS NOT NULL AND
                dep_id NOT IN (SELECT /*+ HASH_AJ */ dep_id
                                FROM departamentos
                                WHERE dep_id IS NOT NULL AND
                                loc = 'TOLUCA');
```

Si se desea que la transformación anti-joing siempre ocurra cuando las condiciones previas se cumplen, se debe colocar el parámetro de inicialización ALWAYS_ANTI_JOIN con MERGE o HASH. La transformación con el correspondiente tipo de anti-joing tendrá lugar cuando sea posible.

Uso de índices

Cuando usar un índice

Los índices mejoran el desempeño de las sentencias que seleccionan un porcentaje pequeño de registros de una tabla. Como una regla general, se debería crear un índice sobre tablas que se requieran frecuentemente por menos de 2 a 4 por ciento de los registros de la tabla. Este valor puede ser más alto en situaciones donde todos los datos pueden recobrarse desde un índice, o donde las columnas indexadas pueden usarse para unir a otras tablas en los joins.

Esta regla es con base en estas suposiciones:

- Los registros con el mismo valor para la columna sobre la que se basa la sentencia se distribuyen uniformemente a lo largo de los bloques de datos destinados a la tabla.
- Los registros en la tabla se ordenan aleatoriamente con respecto a la columna sobre la cual se basa la sentencia.
- La tabla contiene un número relativamente pequeño de columnas.
- La mayoría de las sentencias sobre la tabla tienen relativamente cláusulas WHERE simples.
- El rendimiento del caché es bajo.

Si estas suposiciones no describen los datos de la tabla y las sentencias que accesan, el porcentaje de los registros de la tabla seleccionada bajo un índice es útil y puede aumentarse como mucho a 25%.

Optimizar la estructura lógica

Aunque la optimización basada en costos sea óptima en evitar el uso de índices no selectivos dentro de la ejecución de sentencias, el motor de SQL continua manteniendo todas las definiciones de los índices de las tablas sean usados o no. El mantenimiento de un índice representa una demanda importante de recursos de CPU y E/S en una aplicación intensa en operaciones de E/S. Por lo tanto los índices que se crean "por si a caso" no son buena idea,

estos se deberían construir solo si son requeridos. Los índices no utilizados deben ser borrados. Los índices no utilizados pueden ser detectados al realizar los planes de ejecución de todas las sentencias SQL que utiliza el sistema. Estos índices no son necesarios, típicamente son no selectivos, es decir, no tienen una buena selectividad, no tienen muchos valores distintos o no se selecciona menos del 25% de los registros. Sin embargo, los índices pueden tener usos que no son inmediatamente evidentes desde el resultado de un plan de ejecución, como en el caso de los índices únicos de las llaves primarias que sirven para cuidar la integridad referencial de la tabla o de llaves foráneas.

En muchas aplicaciones el índice de la llave foránea nunca o rara ves, es usado para apoyar una consulta SQL. Y si generar problemas de contienda, es importante revisar si vale la pena la existencia de un índice, para ello no hay más que probar con los datos que se presentan en el sistema en cuestión, tomando en cuenta la cantidad, la selectividad etc...

Cómo escoger las columnas para indexar

Se deben tomar en cuenta las siguientes reglas:

- Considerar las columnas que se usan frecuentemente en las cláusulas WHERE.
- Considerar las columnas que se usan frecuentemente para unir tablas en sentencias SQL.
- Sólo columnas con buena selectividad. La selectividad de un índice es el porcentaje de filas de una tabla que tienen el mismo valor, y esta es buena si pocos registros tienen el mismo valor, o sea que si todas fueran diferentes sería excelente.

Nota: Oracle crea implícitamente índices únicos sobre las columnas que son llaves primarias que se definen en la integridad referencial. Estos índices son los más selectivos y los más efectivos para mejorar el desempeño.

La selectividad de un índice se determina dividiendo el número de registros de la tabla entre el número de valores distintos. Se obtienen estos valores con el comando ANALYZE. La selectividad se calcula de esta manera y se debe interpretar como porcentaje.

Columnas que no se deben indexar.

- Columnas con pocos valores distintos. Algunas columnas tienen selectividad pobre y por lo tanto no perfeccionarían el desempeño, a menos que los valores frecuentemente seleccionados aparezcan con menor frecuencia que los otros valores de la columna. Por ejemplo, considerar una columna con igual número de valores 'SI' y 'NO'. Indexar esta columna no mejoraría normalmente el desempeño. Sin embargo si el valor 'SI' fuera escaso y la aplicación frecuentemente solicita este valor, entonces el índice en esta columna puede mejorar el desempeño.
- Columnas que se modifiquen constantemente. Las sentencias UPDATE que modifican las columnas indexadas y las sentencias INSERT y DELETE que modifican las tablas indexadas toman más tiempo que si la tabla no tuviera índice. Las sentencias SQL deben modificar datos de las tablas e índices si los tienen.
- Columnas que solo aparezcan en cláusulas WHERE con funciones u operadores. Una cláusula WHERE que utiliza una función (como MIN o MAX) o un operador con una columna indexada no dispone de la ruta de acceso del índice, es decir, no utiliza el índice.

Considerar indexar llaves foráneas en caso de que un gran número de sentencias concurrentes del tipo INSERT, UPDATE o DELETE utilicen las tablas maestras y las detalle. El índice permite a Oracle modificar datos en la tabla detalle sin cerrar la tabla maestra.

Cuando se elige indexar una columna, es importante considerar la ganancia de desempeño para las sentencias que la utilizan, por la pérdida de desempeño para las sentencias INSERT, DELETE y UPDATE y el uso del espacio requerido para almacenar el índice. Es bueno experimentar y comparar las sentencias SQL con y sin índices para determinar si vale la pena su creación o no.

Cómo escoger los índices compuestos

Un índice compuesto es un índice que se construye de más de una columna, los índices compuestos pueden proveer de ventajas adicionales sobre un índice de una sola columna.

- Mejor selectividad. A veces dos o más columnas, cada una con selectividad pobre, pueden combinarse en un índice compuesto y producir buena selectividad.

- Almacenamiento adicional de datos. Si todas las columnas seleccionadas por una consulta están en un índice, Oracle regresa estos valores del índice sin accesar la tabla.

Una sentencia SQL puede usar una ruta de acceso que involucra a un índice compuesto si la sentencia contiene la parte principal del índice. La parte principal del índice es una o más columnas que forman el índice, en el orden en el que se creó. Ejemplo, de la sentencia CREATE INDEX:

```
CREATE INDEX comp_ind
   ON tabl(x, y, z);
```

Estas combinaciones de columnas conducen a las porciones de índice: X, XY y XYZ no se consideran porciones de índice combinaciones como: YZ y Z. Por eso se deben seguir las siguientes reglas para elegir columnas que integren índices compuestos:

- Columnas que se usan juntas frecuentemente en condiciones de cláusulas WHERE combinadas con operadores AND, especialmente si su selectividad combinada es mejor que cada una individualmente.
- Si varias sentencias seleccionan el mismo conjunto de columnas con base en uno o más valores de columna, considerar crear un índice compuesto conteniendo todas estas columnas.

Por supuesto, considerar las reglas de generales de balances comparativos y ventajas de desempeño de los índices descritas anteriormente. Siga estas reglas para columnas ordenadas en índices compuestos:

- Crear el índice compuesto para las columnas que se usen en cláusulas WHERE constituidas por una porción principal.
- Si algunas columnas se usan en sentencias WHERE más frecuentemente, se debe crear el índice para aquellas columnas más frecuentemente seleccionadas
- Si todas las columnas se usan en cláusulas WHERE con igual frecuencia, se deben ordenar las columnas desde la más selectiva hasta la menos selectiva en la creación del índice, para mejorar el desempeño.
- Si todas las columnas se usan en cláusulas WHERE con igual frecuencia, pero los datos se ordenan físicamente sobre alguna columna, se debe considerar poner al principio a esta columna.

Como escribir sentencias para aprovechar los índices

Después de crear un índice, el optimizador no puede usar la trayectoria de acceso que usa el índice simplemente porque el índice existe. El optimizador puede usar el índice para una sentencia SQL si esta contiene elementos que lo requieran, para asegurarse de que una sentencia usa un índice, es necesario hacerlo disponible para su uso. Si se usa la optimización basada en costos, se deben generar estadísticas para el índice, una vez que se ha hecho disponible el uso del índice, el optimizador puede o no escoger la trayectoria de acceso del índice u otras rutas de acceso a la información.

Si se crean nuevos índices para mejorar sentencias, se puede usar el comando EXPLAIN PLAN para determinar si el optimizador utilizará los índices cuando la aplicación se encuentre corriendo. Cuando se crean índices para mejorar una sentencia que ha sido analizada, la sentencia se invalida, cuando se vuelve a ejecutar el optimizador selecciona un nuevo plan de ejecución que potencialmente puede utilizar el nuevo índice. Cuando se crean nuevos índices en una base de datos remota para mejorar una sentencia distribuida, el optimizador considera estos índices cuando la sentencia se vuelve a analizar. Es importante recordar que al mejorar el rendimiento de una sentencia se pueden afectar los planes de ejecución de otras. Por ejemplo, si se crea un índice para ser usado en una sentencia, el optimizador puede tomar ese índice para otras sentencias relacionadas. Por esta razón se debe reexaminar el desempeño de la aplicación para revisar que se ha realizado una buena inversión con los índices y no se ha afectado en otras áreas. Además se pueden utilizar sugerencias en la sentencia para "forzarla" a utilizar un índice, esto se mostrará en el capítulo siguiente, ejemplo:

Usar las sugerencias "INDEX" o "AND_EQUAL", obliga al optimizador a utilizar el índice que este disponible para la sentencia en cuestión. Ejemplo, si la tabla empleados tiene un índice en el código de departamento será utilizado:

```
Select /*+ INDEX*/
From empleados
Where cod_dep = 2;
```

Como escribir sentencias que no usen los índices

En algunos casos se requerirá que no se utilice el índice. Esto se puede hacer si el índice no es muy selectivo y un acceso completo sería más eficiente (full table scan). Si la sentencia hace que el uso del índice sea viable, se puede forzar al optimizador a utilizar un acceso completo a la tabla de la siguiente manera.

Se puede hacer que la ruta de acceso del índice no se habilite, modificando la sentencia de tal forma que no cambie su objetivo. Utilizando la sugerencia "FULL" en la sentencia (se muestra más a detalle en el capítulo siguiente) se obliga a la sentencia a realizar una exploración completa a la tabla en lugar de una exploración al índice, ejemplo:

```
Select /*+ FULL*/ campo_1, campo_2
From tabla
Where campo1 > 1;
```

La mejor forma de validar si se usa o no un índice es realizando el plan de ejecución de la sentencia, y la mejor forma de saber si funciona o no el índice es comparando el rendimiento de las sentencias con y sin el índice. Otra forma de no utilizar un índice es utilizando una función sobre la columna que tiene el índice o realizando una operación sobre la misma, ejemplo, si hay un índice en el campo: *campo1*, que es de tipo numérico:

```
Select campo_1, campo_2
From tabla
Where (campo1 * 1) > 1;

Select campo_1, campo_2
From tabla
Where Round(campo1,2) > 1;
```

Para el caso de las columnas de tipo *char* o *varchar* (alfanuméricas), con sólo concatenas un espacio en blanco ya no se utilizaría el índice si esa columna esta indexada.

Fast Full Scan

La exploración completa rápida (fast full scan) es una alternativa a la exploración completa (full scan), cuando todas las columnas que se requieren en la sentencia se encuentran en el índice. La exploración completa rápida es más rápida que la exploración normal al índice porque utiliza operaciones E/S multibloque y se puede paralelizar. Para indicar al optimizador el uso de este tipo de acceso se debe especificar la sugerencia: "INDEX_FFS" en la sentencia.

Regenerar un índice

Es conveniente regenerar un índice a fin de compactar y limpiar su espacio fragmentado, o cambiar sus características de almacenamiento. Cuando se crea un nuevo índice que es un subconjunto de un índice existente o cuando se regenera un índice existente con nuevas características Oracle utiliza el índice existente en vez de la tabla para realizar estas tareas y mejorar el rendimiento. Se utiliza el comando ALTER INDEX REBUILD para reorganizar, compactar o cambiar las características de almacenamiento de un índice existente. Esto es sano cuando el índice pertenece a una tabla que constantemente es actualizada, para mantener al índice en buenas condiciones.

Uso de índices Bitmap

Qué es un índice bitmap

Oracle provee de cuatro esquemas de índice: índices árbol-B, cluster índice árbol-B, cluster índice hash e índice bitmap (mapa de bits). Estos esquemas de índice proveen una funcionalidad complementaria de rendimiento. Los índices contienen una lista de elementos; cada uno de ellos consta de un valor clave y un identificador de fila (RowID), este identificador de fila indica a la base de datos la ubicación exacta de la fila (archivo, bloque de archivo y fila del bloque). Las operaciones de E/S necesarias para localizar un valor de la llave son mínimas y, una vez encontrado, se utiliza el identificador de fila para acceder directamente a la fila.

Cuando usar un índice bitmap.

Los índices bitmap son muy útiles cuando los datos no son muy selectivos (es decir, no hay muchos valores distintos en una columna), los índices bitmap aceleran las búsquedas en las que se utilizan esas columnas poco selectivas como base para eliminar los registros del conjunto de registros devuelto. Estos índices son muy efectivos con los datos muy estáticos.

Como usar un índice bitmap.

Si hubiera muy pocos valores distintos para un campo de *status*, por ejemplo; en una tabla de *ventas* de gran tamaño, normalmente no se crearía un índice de árbol binario para esta columna, incluso aunque se utilice con frecuencia en las cláusulas WHERE. Sin embargo esta columna podría beneficiarse de un índice bitmap. Internamente un índice de mapa de bits establece una correspondencia entre los distintos valores de las columnas y los distintos registros. Para este ejemplo, suponemos que sólo hay dos valores del *status*: "L" y "C" en la tabla de *ventas* de gran tamaño. Dado que hay dos valores, hay dos elementos del bitmap diferentes para el índice bitmap del campo *status*. Si las primeras cinco filas tienen un valor de *status* de "L" y las cinco siguientes tienen un valor de "C", los elementos del bitmap serán similares a los que se muestran en el siguiente listado:

```
status bitmaps:
L:      <1111100000>
C:      <0000011111>
```

En la lista anterior, cada número representa a un registro de la tabla *ventas*. Y dado que se han considerado 10 registros, se muestran 10 valores del bitmap. Si se lee el bitmap de *status*, se nota que los primeros cinco registros tienen un valor de "L" (los valores 1) y los siguientes cinco no (los valores 0). Se podrían tener más de dos valores posibles para la columna, en este caso habría una fila de bitmap diferente para cada valor posible. El optimizador puede convertir dinámicamente los elementos del índice bitmap en valores de RowID durante el procesamiento de consultas. Esta capacidad de conversión permite utilizar índices tanto en columnas que contienen muchos valores distintos (por medio de árboles binarios) como aquellas que tienen pocos valores distintos (mediante índices bitmap). Para crear un índice de mapa de bits, se utiliza la cláusula BITMAP en la orden CREATE INDEX, tal y como se muestra en el siguiente ejemplo, en el nombre del índice se debería indicar su carácter de índice bitmap, de tal forma que sea fácil detectarlo durante las tareas de optimización.

```
create bitmap index idx_ventas_status_bmap
on ventas(status);
```

Si se opta por utilizar índices de este tipo, habrá que estudiar si la mejora en el rendimiento de las consultas compensa la disminución del rendimiento en las sentencias de manipulación de datos. Cuantos más índices haya en una tabla, mayor será el costo en una transacción. No es recomendable utilizar un índice de mapa de bits en una tabla a la que constantemente se le añadan nuevos valores. Cada adición de un nuevo valor, requerirá que se cree una nueva fila de bitmap para el nuevo valor. Cuando se crean índices bitmap, Oracle comprime los bitmaps para almacenarlos. Como resultado, el espacio que necesita un índice bitmap es de sólo del 5 al 10 por ciento del espacio que necesita un índice normal. Por lo tanto se debería pensar en la posibilidad de utilizar índices de mapa de bits para cualquier columna no selectiva que se utilice frecuentemente en las cláusulas WHERE, siempre y cuando el conjunto de valores distintos de la columna sea limitado.

Capítulo IV. Optimización de aplicaciones.

4.1 Optimización del diseño de aplicaciones

Optimización durante el diseño

Es por eso que durante el diseño de cada proceso de la aplicación se debe pensar en la mejor forma de realizarlo en el desarrollo, por ejemplo, si es necesario interactuar con un cursor que recorra un conjunto de registros, debemos pensar en evitar el tráfico de la red y no desarrollar el proceso del cursor en la aplicación (en el cliente); para ponerlo en un store procedure (procedimiento almacenado), que realiza todas las operaciones en el RDBMS (en el servidor), por lo tanto es más rápido. Como en este ejemplo, todos los procesos, además de que realicen su objetivo, lo deben hacer de la mejor manera, siempre aprovechando al máximo los recursos disponibles.

Diseño adecuado de tablas

El diseño de las tablas, es parte fundamental en el rendimiento de las aplicaciones, si un campo debe ser de un tipo de dato, de una determinada longitud, si debe dividirse en más campos o deben estar unidos, en el caso de las llaves compuestas. Es importante que las tablas queden bien diseñadas, a veces hay que considerar a la tabla en cuestión no sólo como algo individual, sino en conjunto con las tablas que más interacción tienen con ella. Debido a que la información sólo tiene sentido cuando se muestran los datos que se requieren. Así, si una consulta de un reporte que es el más importante del sistema, necesita de ciertas características de las tablas y estas no las tienen, estas se deben adaptar a las necesidades que tienen más prioridad. Aunque no es la forma más ortodoxa de hacerlo, se justifica si se mejora el rendimiento de las consultas más importantes del sistema, aunque se perjudiquen otras que no lo sean tanto. Cabe mencionar que a veces es necesario incluir tablas que no se producen en el diseño "normal" del sistema, y por lo tanto se llaman "tablas desnormalizadas" que contienen campos necesarios para consultas importantes y que para mejorar su rendimiento y evitar joins, se crean en procesos independientes que las limpian y llenan cuando sea necesario refrescarlas.

Distribución de las necesidades del procesador

Un punto que aparentemente es trivial, a veces es pasado por alto es el de darle el tiempo adecuado a los procesos del RDBMS. Por ejemplo, la regeneración de tablas desnormalizadas, la ejecución de procesos fuertes, la regeneración de índices, etc., debe realizarse en horarios que no interfieran entre sí, en la medida de lo posible. Por ejemplo, un proceso de cierre, que realice muchos cálculos sobre tablas de captura de información, no se debe realizar al momento de la captura, porque va a ser muy lento, tanto la captura como el proceso de cierre, porque la captura constantemente bloquea la tabla en el momento de actualizarla y hacer el commit y si el proceso necesita la tabla para obtener información, también lo hará, hasta que termine la consulta que la necesite. Si programan tiempos para las dos cosas, las dos cosas se realizarán más rápido.

Plan de ejecución de una sentencia SQL

Es la forma en que Oracle resolverá la sentencia SQL que le indicamos. Para poder conocerla necesitamos crear una tabla llamada PLAN_TABLE, que contendrá el plan de ejecución de la sentencia que le indiquemos. La tabla se encuentra localizada en el script

UTLXPLAN.SQL que se incluye en la instalación de Oracle y comúnmente se localiza en el directorio principal del software. Para el caso del cliente de Oracle para Windows, se encuentra en: C:\Orawin95\Rdbms73\Admin\Utlxplan.sql que en suma contiene el siguiente código para crear la "PLAN_TABLE" o tabla de los planes de ejecución:

```
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);
```

La "PLAN_TABLE" es usada por el comando EXPLAIN PLAN y sus columnas indican lo siguiente (Tabla 4.1).

Columna	Uso
STATEMENT_ID	El valor especificado del parámetro: STATEMENT_ID de la sentencia EXPLAIN PLAN. Que es el identificador de la sentencia.
TIMESTAMP	La fecha y la hora en que se ejecuto la sentencia.
REMARKS	Cualquier comentario (menor a 80 bytes) que se asocie con cada paso del plan de ejecución. Si se necesita agregar o cambiar las anotaciones de cualquier renglón de la tabla, usamos la sentencia UPDATE para modificarlos.
OPERATION	El nombre de la operación interna realizada en este paso. En el primer renglón generado por esta sentencia, la columna contiene uno de los siguientes valores: 'DELETE STATEMENT' 'INSERT STATEMENT' 'SELECT STATEMENT' 'UPDATE STATEMENT'
OPTIONS	Una variación de la operación descrita en la columna OPERATION.
OBJECT_NODE	El nombre de la liga a la base de datos usada como referencia al objeto (nombre de la tabla o vista). Para consultas locales que utilizan la opción de consulta en paralelo, esta columna describe el orden con el cual las operaciones son realizadas.
OBJECT_OWNER	Nombre del usuario dueño del esquema que contiene la tabla o índice.
OBJECT_NAME	El nombre de la tabla o índice.
OBJECT_INSTANCE	Número correspondiente a la posición del objeto tal como aparece en la sentencia original.

Tabla 4.1. Campos de la "Plan table".

Columna	Uso
OBJECT_TYPE	Un indicador que contiene información descriptiva acerca del objeto. Por ejemplo: NON-UNIQUE para los índices.
OPTIMIZER	El modo actual del optimizador.
SEARCH_COLUMNS	No utilizado.
ID	Número asignada a cada paso del plan de ejecución.
PARENT_ID	El ID del siguiente paso en el plan de ejecución que opera con la salida del paso actual.
POSITION	Orden de ejecución de los pasos que tienen el mismo PARENT_ID.
OTHER	Otra información que se especifica en cada paso de la ejecución que el usuario considera necesaria.
OTHER_TAG	Describe el contenido de la columna OTHER.
COST	El costo de la operación es estimado por el optimizador en el modo de acceso basado en costos. Para las sentencias que usan el acceso basado en reglas esta columna adquiere un valor nulo.
CARDINALITY	Número estimado de registros accesados en la operación.
BYTES	Número estimado de bytes accesados en la operación.

Tabla 4.1. Campos de la "Plan table" (continuación).

La tabla 4.2 muestra cada combinación de operación y opción producida por el comando EXPLAIN PLAN y el significado en el plan de ejecución.

OPERATION	OPTION	Description
AND-EQUAL		Una operación que acepta múltiples grupos de ROWIDS y regresa la intersección de los grupos, eliminando duplicados. Esta operación es usada por columnas únicas indexadas.
CONNECT BY		Recuperación de registros en orden jerárquico para una consulta que contenga la cláusula CONNECT BY.
CONCATENATION		Operación que acepta múltiples grupos de registros y regresa la unión completa de los grupos.
COUNT		Operación que cuenta el número de registros seleccionados para una tabla.
	STOPKEY	Operación que cuenta el número de registros recuperados que es limitada por la expresión ROWNUM en la cláusula WHERE.
FILTER		Operación que acepta un grupo de registros, elimina algunos de ellos y regresa el resto.
FIRST ROW		Recupera solo el primer registro seleccionado por una consulta.
FOR UPDATE		Recupera y bloquea los registros seleccionados por una consulta que contenga una cláusula FOR UPDATE.
INDEX*	UNIQUE SCAN	Recupera un único ROWID de un índice
	RANGE SCAN	Recupera uno o más ROWIDs de un índice. Los valores del índice son explorados en orden ascendente.
	RANGE SCAN DESCENDING	Recupera uno o más ROWIDs de un índice. Los valores del índice son explorados en orden descendente.

Tabla 4.2. Operaciones del comando "Explain Plan".

OPERATION	OPTION	Description
INTERSECTION		Operación que acepta dos grupos de registros y regresa la intersección de los grupos, eliminando duplicados.
MERGE JOIN+		Esta operación acepta dos grupos de registros, cada ordenación por un valor específico, combina cada registro de un grupo con los registros que hacen pareja del otro y regresa el resultado.
	OUTER	Una combinación de la operación MERGE JOIN que realiza una sentencia OUTER JOIN.
CONNECT BY		Recupera los registros en un orden jerárquico de una consulta que contiene la cláusula CONNECT BY.
MINUS		Una operación que acepta dos grupos de registros y regresa los registros que aparecen en el primer grupo pero no en el segundo, eliminando duplicados.
NESTED LOOPS*		Operación que acepta dos grupos de registros, un grupo externo y un grupo interno. Oracle compara cada registro del grupo externo con cada registro del grupo interno y regresa aquellos que satisfagan la condición.
	OUTER	Una operación NESTED LOOPS que realiza una sentencia OUTER JOIN.
PROJECTION		Una operación interna.
REMOTE		Regresa los datos de una base de datos remota.
SEQUENCE		Operación que implica accesar valores de una secuencia.
SORT	AGGREGATE	Recupera un solo registro que es resultado de aplicar una función de grupo a un grupo de registros seleccionados.
	UNIQUE	Operación que ordena un grupo de registros eliminando duplicados.
	GROUP BY	Esta operación ordena los registros en grupos, en base a las columnas especificadas en la cláusula GROUP BY.
	JOIN	Ordena un conjunto de registros antes de realizar una operación "merge-join".
	ORDER BY	Ordena el conjunto de registros de una consulta con la cláusula ORDER BY.
TABLE ACCESS*	FULL	Recupera todos los registros de la tabla.
	CLUSTER	Recupera registros de la tabla basados en el valor de la llave de un índice de grupo (cluster).
	HASH	Recupera registros de la tabla basados en el valor de la llave de un grupo hash (hash cluster).
	BY ROWID	Recupera un registro de la tabla basado en el ROWID.
UNION		Una operación que acepta dos conjuntos de registros y regresa la unión de los conjuntos, eliminando duplicados.
VIEW		Operación que realiza una consulta a una vista y regresa los registros resultantes a otra operación.

Tabla 4.2. Operaciones del comando "Explain Plan" (continuación).

* Estas operaciones son métodos de acceso.
\+ Estas operaciones son operaciones de cruce (joins).

Como se realiza un plan de ejecución

El siguiente ejemplo muestra una sentencia SQL y su correspondiente plan de ejecución generado por la sentencia EXPLAIN PLAN. La consulta de ejemplo recupera los nombres e

información relacionada de los empleados cuyo salario no este dentro de ningún rango de la tabla SALGRADE. La consulta es al siguiente:

```
SELECT ename, job, sal, dname
   FROM emp, dept
   WHERE emp.deptno = dept.deptno
      AND NOT EXISTS
         (SELECT *
            FROM salgrade
            WHERE emp.sal BETWEEN losal AND hisal);
```

Esta sentencia EXPLAIN PLAN genera un plan de ejecución y deja la salida en la "PLAN_TABLE":

```
EXPLAIN PLAN
   SET STATEMENT_ID = 'Emp_Sal' FOR
    SELECT ename, job, sal, dname
      FROM emp, dept
      WHERE emp.deptno = dept.deptno
         AND NOT EXISTS
            (SELECT *
               FROM salgrade
               WHERE emp.sal BETWEEN losal AND hisal);
```

Observemos que lo que se encuentra en negritas en la sentencia anterior es sintaxis de la instrucción EXPLAIN PLAN, a partir de la palabra FOR, se escribe la sentencia que se desee analizar.

Obtener el plan de ejecución de la "PLAN_TABLE"

Necesitamos realizar una consulta para conocer el plan de ejecución, esta consulta puede tener todos los campos de la "PLAN_TABLE" o sólo los más representativos, como en el siguiente ejemplo:

```
SELECT operation, options, object_name, id, parent_id, position
    FROM plan_table
    WHERE statement_id = 'Emp_Sal'
   ORDER BY id;
```

OPERATION	OPTIONS	OBJECT_NAME	ID	PARENT_ID	POSITION	COST	CARDINALITY	BYTES	OTHER_TAG	OPTIMIZER
SELECT STATEMENT			0		2	2	1	62		CHOOSE
FILTER			1	0	1					
NESTED LOOPS			2	1	1	2	1	62		
TABLE ACCESS	FULL	EMP	3	2	1	1	1	40		ANALYZED
TABLE ACCESS	FULL	DEPT	4	2	2		4	88		ANALYZED
TABLE ACCESS	FULL	SALGRADE	5	1	2	1	1	13		ANALYZED

La cláusula ORDER BY regresa los pasos del plan de ejecución ordenados secuencialmente por el valor del ID. Sin embargo, Oracle no realiza los pasos en este orden. PARENT ID recibe información del ID, hay más de un ID que es necesario para un PARENT ID. Por ejemplo, el paso 2, un "merge join", y el paso 7, un acceso a tabla, dependen del paso 1. Una representación visual de la secuencia del proceso se muestra en la figura 4.1.

El valor de la columna POSITION del primer registro de la salida indica el costo estimado por el optimizador de la ejecución de la sentencia, con este plan será de 5. Para los otros renglones, este indica la posición relativa de los otros procesos hijos del mismo padre.

Mostrar la salida del plan de ejecución de una forma anidada

Este tipo de sentencia SELECT genera una representación anidada de la salida que muestra el orden de procesamiento usado para la sentencia SQL.

```
SELECT LPAD(' ',2*(LEVEL-1))||operation||' '||options
   ||' '||object_name
   ||' '||DECODE(id, 0, 'Cost = '||position) "Query Plan"
   FROM plan_table
   START WITH id = 0 AND statement_id = 'Emp_Sal'
   CONNECT BY PRIOR id = parent_id AND statement_id ='Emp_Sal';

Query Plan
------------------------------
SELECT STATEMENT   Cost = 5
   FILTER
      NESTED LOOPS
         TABLE ACCESS FULL EMP
         TABLE ACCESS FULL DEPT
      TABLE ACCESS FULL SALGRADE
```

El orden se asemeja a una estructura de árbol, ilustrada en la figura 4.1:

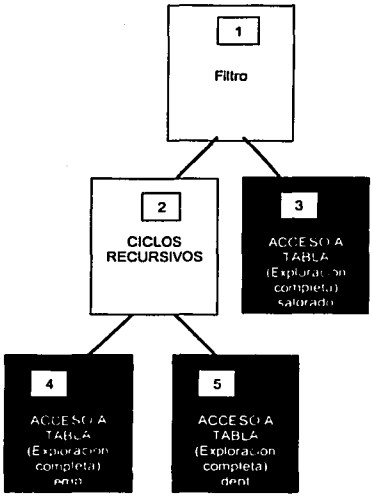


Figura 4.1 Estructura de árbol del plan de ejecución

La estructura de árbol muestra como las operaciones van ocurriendo durante la ejecución de la sentencia SQL. A cada paso del plan de ejecución se le asigna un número (representado por la columna ID de la tabla del plan de ejecución "plan_table") y es descrito por un nodo, el resultado de cada operación del nodo pasa a su nodo padre el cual utiliza como entrada.

4.2 Optimización de las sentencias SQL

Hay dos formas ejecutar una sentencia por parte del RDBMS, a través de reglas o costos. La optimización basada en reglas tiende a ya no ser utilizada y la optimización basada en costos tiende a consolidarse como la mejor forma de ejecutar una sentencia SQL. Algo muy importante antes de iniciar la optimización de las sentencias SQL, es asegurarse que las sentencias que se utilizan para realizar algún proceso, sean las sentencias correctas, es decir, que estas, sean la mejor manera de resolver el problema, se ser así, entonces si deben ser optimizadas. De lo contrario, se debe ajustar el proceso, para que se realice de la mejor manera, una vez, ajustado, estas sentencias, si deben ser optimizadas. Esto es, que no se debe iniciar a optimizar sentencias SQL, si no se esta seguro de que estas son la mejor manera de resolver los procesos en cuestión.

Optimización basada en reglas (RBO)

La optimización basada en reglas como su nombre lo índica utiliza reglas lógicas para realizar las operaciones que ejecuten una sentencia SQL, aunque estas no sean las más optimas, ver tabla 4.3. Por ejemplo, si una tabla tiene un índice no selectivo y una sentencia usa ese campo para hacer una restricción en una cláusula WHERE, el optimizador basado en reglas utilizará el índice, como la lógica indica; pero sabemos que esta no es la mejor elección porque el índice no es selectivo, en cambio el optimizador basado en costos no lo haría, y buscaría una alternativa que tuviera un menor costo de trabajo. Si se han desarrollado aplicaciones que usan la versión 6 de Oracle y ha afinado cuidadosamente sus sentencias SQL con base en las reglas del optimizador, se puede continuar usando la optimización basada reglas cuando se actualizan estas aplicaciones a Oracle 7 y así no tener que modificar todas las sentencias para adaptarlas a la optimización basada en costos.

Si re realizan estadísticas a los objetos de la base de datos utilizando la optimización basada en reglas, aunque se le indique a la sentencia SQL una sugerencia para aprovechar las estadísticas, el optimizador seguirá utilizando el enfoque basado en reglas, aunque estos datos de las estadísticas nos pueden servir a nosotros para otros propósitos informativos. Sin embargo se debería cambiar el enfoque de optimización basado en reglas al de costos, porque el enfoque de optimización basado en reglas no estará disponible n futuras versiones de Oracle. Para hacer este proceso se puede realizar una base de datos de prueba y en ella realizar estadísticas y pruebas de rendimiento. Para modificar el valor del parámetro de inicialización: OPTIMIZER_MODE o el parámetro OPTIMIZER_GOAL del comando ALTER SESSION y dejar el modo de optimización basado en reglas se especifica el valor: RULES.

#	Regla	Descripción
1	Single row by ROWID	Un registro por índice.
2	Single row by cluster join	Un registro por join agrupado.
3	Single row by hash cluster key with unique or primary key	Un registro por Llave de grupo hash.
4	Single row by unique or primary key	Un registro por Llave primaria.
5	Cluster join	Join de grupo.
6	Hash cluster key	Llave de grupo hash.
7	Indexed cluster key	Llave de grupo indexada.
8	Composite index	Índice compuesto.
9	Single-column index	Índice de una sola columna.
10	Bounded range search on indexed columns	Búsqueda de rango limitado sobre columnas indexadas.

Tabla 4.3. Rango de las reglas en RBO.

#	Regla	Descripcion
11	Unbounded range search on indexed columns	Búsqueda de rango ilimitado sobre columnas indexadas.
12	Sort-merge join	Join ordenar/combinar.
13	MAX or MIN of indexed column	MAX o MIN sobre columnas indexadas.
14	ORDER BY indexed columns	ORDER BY columnas indexadas.
15	Full table scan	Exploración completa de la tabla.

Tabla 4.3. Rango de las reglas en RBO (continuación).

Optimización basada en costos (CBO)

El acceso basado en costos generalmente elige un plan de ejecución tan bueno o mejor que el plan de ejecución seleccionado por el acceso basado en reglas, especialmente para consultas grandes con múltiples joins o múltiples índices. El acceso basado en costos además mejora la productividad, eliminando la necesidad de que se mejoren las sentencias SQL por nosotros. Finalmente muchas características de rendimiento están disponible solo a través del acceso basado en costos.

La optimización basada en costos debe ser usada para realizar consultas con un rendimiento eficiente. La optimización basada en costos es usada siempre con consultas en paralelo y con vistas particionadas. Es decir en consultas complejas.

Como usar el acceso basado en costos.

Para habilitar la optimización basada en costos para una sentencia, se reúnen estadísticas para las tablas accesadas por la sentencia y si parámetro de inicialización del OPTIZER_MODE (forma de optimización) se encuentra en el valor por defecto: COSE, debe hacer lo siguiente:

- Si es solo para una sesión de algún usuario, se utiliza la sentencia ALTER SESSION con la opción OPTIZER_GOAL y el valor ALL_ROWS o FIRST_ROWS.
- Si es para una sentencia SQL individual, usar cualquier sugerencia diferente a RULE (reglas).

Los planes de ejecución generados por el optimizador basado en costos, depende del tamaño de las tablas. Cuando se usa la optimización basada en costos con pequeñas cantidades de datos, no se asume que cuando la base de datos esta llena el plan de ejecución es el mismo que el del prototipo.

Generar estadísticas

Cuando se use la optimización basada en costos se deben reunir estadísticas de los objetos que utilizará el optimizador. Estas estadísticas se realizan con la sentencia ANALYZE. No se deben realizar estadísticas sobre el esquema SYS. Ya que estos objetos están hechos tal manera que trabajan satisfactoriamente sin necesidad de analizarlos para recoger estadísticas. Si se realizan estadísticas sobre estos objetos en algunas situaciones se disminuye el rendimiento, las estadísticas se deben realizar solo con las tablas que se utilizan en las sentencias que utiliza la aplicación, por lo general las tablas de datos principales.

Se realiza este análisis de dos maneras: calculando y estimando. Para calcular estadísticas sólo se codifica el nombre del objeto seguido por las palabras clave: COMPUTE STATICS, tal y como se muestra en el siguiente listado:

```
ANALYZE INDEX idx_mi_indice COMPUTE STATICS;

ANALYZE TABLE mi_tabla COMPUTE STATICS;
```

Indicadores (sugerencias)

Como diseñador de la aplicación, se tienen información sobre los datos que el optimizador no puede. Por ejemplo, un diseñador debe saber que un determinado índice es más selectivo para determinadas preguntas que el optimizador no puede determinar. Con base a esta información, se puede escoger un plan de ejecución más eficiente al que el optimizador elegirá. En tal caso, se utilizan indicadores para forzar al optimizador a usar el plan elegido de ejecución.

Los indicadores son sugerencias que se dan al optimizador para mejorar las sentencias SQL, permiten tomar decisiones comúnmente hechas por el optimizador. Se utilizan indicadores para especificar:

- La forma de optimización de una sentencia SQL.
- El objetivo del enfoque basado en costos para una sentencia SQL.
- La ruta de acceso para una tabla accesada por la sentencia SQL.
- El orden de los joins.
- El tipo de join.

Como especificar los indicadores

Los indicadores sólo afectan al bloque de la sentencia donde aparecen. Un bloque de sentencias es cualquiera de las partes o sentencias siguientes:

- Una sentencia simple de SELECT, UPDATE o DELETE.
- Una sentencia padre o una subconsulta de una sentencia compleja.
- Una parte de una consulta compuesta.

Una consulta compuesta consiste, en dos sentencias combinadas por el operador UNION, cada consulta puede ser afectada por indicadores distintos. Esto significa que si la primera sentencia tiene un indicador su optimización sólo se modifica para esta, no afectara la optimización de la segunda. Los indicadores se emplean adjuntando un comentario dentro de la sentencia.

Un bloque de sentencias puede tener un solo comentario que contenga los indicadores. Este comentario debe seguir a la palabra clave SELECT, UPDATE o DELETE. Si se especifican incorrectamente las sugerencias, Oracle las ignora y no regresa error.

- Oracle ignora los indicadores, si estos no se encuentran enseguida de las palabras clave: SELECT, INSERT y UPDATE.
- Oracle ignora los indicadores que contienen errores de sintaxis, pero si toma en cuenta otros que se encuentren en el mismo comentario.
- Oracle ignora los indicadores que estén en conflicto, pero si toma en cuenta otros que se encuentren en el mismo comentario.

El optimizador sólo reconoce los indicadores cuando se utiliza el acceso basado en costos. Si se incluye cualquier indicador en una sentencia (excepto el indicador "RULE"), el optimizador automáticamente usa el acceso basado en costos.

Indicadores para la optimización de accesos

Los indicadores descritos en esta sección permiten escoger entre los enfoques de optimización basados en reglas o en costos y, con el enfoque basado en costos, entre las metas de mejor salida y mejor tiempo de respuesta. Si una sentencia SQL contiene un indicador que especifica una meta y enfoque de optimización, el optimizador usa el enfoque especificado sin considerar la presencia o ausencia de estadísticas, el valor del OPTIMIZER_MODE parámetro de inicialización, y el OPTIMIZER_GOAL de parámetro del comando ALTER SESSION.

ALL_ROWS

El indicador ALL_ROWS explícitamente escoge el enfoque basado en costos para perfeccionar una sentencia con la meta que utilice el mínimo de recursos. Por ejemplo el optimizador el acceso basado en costos para optimizar la siguiente sentencia con una utilización mínima de recursos.

```
SELECT /*+ ALL_ROWS */ campo1, campo2, campo3
   FROM  tabla
   WHERE campo1 = 7566;
```

FIRST_ROWS

El indicador FIRST_ROWS explícitamente escoge el enfoque basado en costos para perfeccionar una sentencia con una meta de mejor tiempo de respuesta (uso mínimo de recursos para traer la primera fila). Este indicador ocasiona que el optimizador haga las siguientes elecciones:

- Si hay un índice disponible, el optimizador puede seleccionarlo sobre un análisis completo de la tabla.
- Si hay un índice disponible, el optimizador puede seleccionar nested loops join sobre un sort-merge join, cuando la tabla asociada sea la tabla interior potencial de los nested loops.
- Si hay un índice se hace disponible por una cláusula ORDER BY, el optimizador puede evitar realizar la operación de ordenamiento.

Por ejemplo, el optimizador usa el enfoque basado en costos para perfeccionar esta declaración para mejorar el tiempo de respuesta:

```
SELECT /*+ FIRST_ROWS */ campo1, campo2, campo3
   FROM  tabla1
   WHERE campo1 = 7566;
```

El optimizador ignora este que indicador en las sentencias DELETE, UPDATE y en las sentencias SELECT que contengan una de las siguientes sintaxis:

- Un operador de conjunto (UNION, INTERSECT, MINUS, UNION ALL)
- Una cláusula GROUP BY
- Una cláusula UPDATE
- Funciones de grupo
- Operador DISTINCT

Estas sentencias no pueden ser optimizadas con este indicador porque Oracle debe recuperar todos los registros accesados por la sentencia antes de regresar el primer registro. Si se especifica este indicador en cualquiera de estas sentencias el optimizador usa el enfoque basado en costos y optimiza para una utilización mínima de recursos. Si se especifica el indicador de ALL_ROWS o FIRST_ROWS en alguna sentencia y el diccionario de datos no contiene estadísticas para ninguna de las tablas accesadas por la sentencia, el optimizador usa los valores de las estadísticas por defecto (como el espacio utilizado para las tablas) para estimar las estadísticas faltantes y subsecuentemente elegir un plan de ejecución. Esta estimación no puede ser tan precisa como la que se genera con el comando ANALYZE, se debe utilizar el comando ANALIZE para generar estadísticas para todas las tablas por las sentencias que usan optimización basada en costos. Si se especifican indicadores para las rutas de acceso u operaciones en los joins conjuntamente con los indicadores ALL_ROWS o FIRST_ROWS, el optimizador da precedencia a las rutas de acceso y a las operaciones de joins especificadas en los indicadores.

CHOSSE

El indicador CHOOSE ocasiona que el optimizador escoja entre la enfoque basado en reglas y el enfoque basado en costos para una sentencia SQL con base en la presencia de estadísticas para las tablas accesadas por la sentencia. Si el diccionario de datos contiene estadísticas para por lo menos una de estas tablas, el optimizador usa el enfoque basado en costos y perfecciona con la sentencia con el objetivo de utilizar el menor uso de recursos. Si el diccionario de datos no contiene estadísticas para ninguna de las tablas que se involucran en la sentencia, el optimizador usa el enfoque basado en reglas. Ejemplo:

```
SELECT /*+ CHOOSE */
campo1, campo2, campo3
      FROM emp
      WHERE empno = 7566;
```

RULE

El indicador RULE explícitamente escoge la optimización basada en reglas para el bloque de la sentencia. Este indicador también ocasiona que el optimizador ignore cualquier otro indicador especificado para el bloque de la sentencia. Por ejemplo, el optimizador usa el enfoque basado en reglas para esta declaración:

```
SELECT      --+ RULE
empno, ename, sal, job
   FROM emp
   WHERE empno = 7566;
```

El indicador RULE, conjuntamente con el enfoque basado en reglas, no será disponible en futuras versiones de Oracle (de la 7.3 en adelante). Por lo tanto lo más recomendable es que no se utilice.

Indicadores para los métodos de acceso

Cada indicador descrito en esta sección sugiere un método de acceso para una tabla. Especificando uno de estos indicadores causa que el optimizador elija la ruta de acceso especificada sólo si la ruta de acceso se encuentra disponible basada en la existencia de un índice o un cluster y la construcción sintáctica de una sentencia SQL. Si el indicador especifica una ruta de acceso no disponible, el optimizador lo ignora.

FULL

El indicador FULL explícitamente elige una exploración completa de la tabla especificada. La sintaxis del indicador FULL es:

FULL (*tabla*)

Donde *tabla* especifica el nombre o el alias de la tabla en la cual la exploración completa será realizada. Por ejemplo, Oracle realiza una exploración completa en la tabla PROD el ejecutarse la siguiente sentencia, aún cuando hay un índice en la columna *cod_id* que se hace disponible por la condición en la cláusula WHERE.

```
SELECT /*+ FULL(a) No usa el índice de cod_id*/ cod_id, nom_prod
   FROM prod a
   WHERE cod_id = 7086854;
```

Porque la tabla PROD tiene un alias, a, el indicador se debe referir a la tabla por este alias, más que por su nombre. También, no se especifica el esquema, este en su caso se especifica en la cláusula FROM.

ROWID

El indicador ROWID escoge explícitamente una tabla y realiza una exploración por rowid para la tabla especificada. La sintaxis para el indicador de ROWID es:

ROWID(*tabla*)

Donde *tabla* especifica el nombre o el alias de la tabla sobre la cual el acceso por ROWID será realizado.

CLUSTER

El indicador CLUSTER explícitamente escoge una exploración de grupo para accesar la tabla especificada. Su sintaxis es:

CLUSTER(*tabla*)

Donde *tabla* especifica el nombre o el alias de la tabla para ser accesada por una exploración de grupo.

El ejemplo siguiente ilustra el uso del indicador CLUSTER.

```
SELECT --+ CLUSTER tabla1
a.campo1, b.campo2
        FROM tabla1 a, tabla2 b
        WHERE campo1   = 10 AND
              a.campo1 = b.campo1;
```

HASH

El indicador HASH explícitamente elige una exploración HASH para accesar a la tabla especificada. La sintaxis es la siguiente:

HASH(*tabla*)

Donde *tabla* especifica el nombre o el alias de la tabla para ser accesada por una exploración de tipo HASH.

HASH_AJ

El indicador HASH_AJ transforma una subconsulta del tipo NOT IN en un hash anti-join para accesar la tabla especificada. La sintaxis es la siguiente:

HASH_AJ(*tabla*)

Donde *tabla* especifica el nombre o el alias de la tabla a ser accesada.

INDEX

La sugerencia INDEX elige una exploración de índice a la tabla especificada. La sintaxis para el indicador INDEX es:

INDEX(*tabla índice*)

tabla: especifica el nombre o alias de la tabla asociada con el índice a ser explorado.
índice: especifica el índice sobre el cual se realizará la exploración.

En este indicador se pueden especificar uno o más índices:

- Si se especifica un solo índice, el optimizador realiza la exploración sobre este índice, no considera realizar una exploración completa o sobre otro índice de la tabla.
- Si se especifica una lista de índices disponibles, el optimizador considera el costo de explorar cada uno de los índices y realiza la exploración en el índice que produzca un menor costo. El optimizador no considera realizar una exploración completa o sobre otro índice de la tabla no especificado en el indicador.
- Si el indicador no especifica algún índice, el optimizador considera el costo de explorar todos los índices disponibles de la tabla y realiza la exploración en el índice que produzca un menor costo. El optimizador puede elegir explorar múltiples índices y combinar los resultados; si tal trayectoria de acceso tiene el menor costo. El optimizador no considera una exploración completa sobre la tabla.

```
SELECT /*+ INDEX(pacientes sex_index) */
   nombre, estatura, peso
   FROM pacientes
   WHERE sex = 'M';
```

La consulta anterior considera el índice sex_index para obtener los pacientes de sexo masculino. Se debe tener en cuenta que si el número de registros que sé van a obtener es de hasta 25% del total, esto es conveniente, si no es así, es preferible que realice una exploración completa sobre la tabla.

INDEX_ASC

Tiene la misma sintaxis que indicador anterior y realiza un exploración sobre el índice especificado de forma ascendente. Esto es útil cuando el índice se guarde de forma descendente y el rango de valores que se necesitan están hasta el final.

INDEX_ASC

Realiza una combinación de los índices disponibles, con el menor costo de operación, si se especifican índices, sólo combina los que se listan. (Misma sintaxis que el indicador INDEX).

INDEX_FFS

Efectúa una rápida exploración completa del índice, más que una exploración completa de la tabla. (Misma sintaxis que el indicador INDEX). Esto es útil cuando el índice contiene todos los campos que se requieren en la consulta, con este indicador, se toman directamente del índice.

MERGE_AJ

Este indicador transforma una subconsulta contenida en una cláusula NOT IN en un anti join comninado para accesar la tabla.

MERGE_AJ(*tabla*)

Donde *tabla* es el nombre de la tabla o alias de la tabla a ser accesada,

AND_EQUAL

Elige un plan de ejecución que usa una ruta de acceso que combina exploraciones de varias columnas indexadas únicas, figura 4.2.

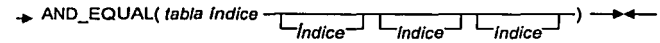


Figura 4.2. Sintaxis del indicador AND_EQUAL

tabla: especifica el nombre o alias de la tabla asociada con el índice a ser explorado.
índice: especifica el índice sobre el cual se realizará la exploración.

USE_CONCAT

El uso de este hint forza a combinar las condiciones del tipo OR de una cláusula WHERE de una sentencia a ser transformadas en una sentencia compuesta usando el operador de conjuntos UNION ALL. Normalmente esta transformación ocurre solo si el costo de la sentencia usando la concatenación es menor que él costo sin utilizarla. Y sigue la misma sintaxis, aunque no se especifica la tabla sobre la que se realizarán las operaciones, porque se realiza sobre las condiciones independientemente de las tablas.

Indicadores para el orden de los Joins

Estos indicadores sugieren un orden en los joins.

ORDERED

El indicador ORDERED provoca que Oracle haga el join de las tablas en el orden que aparecen en la cláusula FROM. Por ejemplo, en la siguiente sentencia realiza el join entre la tabla1 y la tabla2, y luego el resultado con la tabla3.

```
SELECT /*+ ORDERED */ tabla1.col1, tabla2.col2, tabla3.col3
FROM tabla1, tabla2, tabla3
WHERE tabla1.col1 = tabla2.col1
AND   tabla2.col1 = tabla3.col1;
```

Si no se especifica el orden en este indicador, el optimizador selecciona el orden en el que se realizarán los joins que considere conveniente. Se puede utilizar este indicador para dar un orden en la secuencia de los joins, si conocemos la cantidad de registros que se seleccionan de las tablas que participan en la sentencia.

STAR

Cuando se utiliza el indicador STAR, se sugiere al optimizador que la tabla más grande que realiza el join al último, use "nested loops" (ciclos anidados) para hacer el join sobre el índice. El optimizador considerará diferentes permutaciones con las tablas pequeñas. Usualmente, si se analizan las tablas el optimizador elige un plan de ejecución eficiente, se pueden utilizar los indicadores para mejorar el plan. El método más preciso es ordenar las tablas en la cláusula FROM en el orden de las llaves de los índices, con la tabla más grande al último. Se pueden utilizar los siguientes indicadores:

```
/*+ ORDERED USE_NL(facts) INDEX(facts fact_concat) */
```

Un método más general es utilizar el indicador STAR:

```
/*+ STAR */
```

Indicadores para operaciones Join

Se debe especificar la tabla combinada exactamente como aparece en la sentencia. Si la tabla usa un alias se debe utilizar el alias en el indicador. Los indicadores USE_NL y USE_MERGE deben ser utilizados con el indicador: ORDERED. Oracle usa estos indicadores cuando la tabla referenciada es forzada a ser la tabla interior del Join y son ignorados si la tabla referenciada es la tabla exterior del Join.

USE_NL

El uso del indicador USE_NL causa que Oracle realice el join de las tablas especificadas, con otra tabla utilizando ciclos anidados utilizando la tabla especificada como tabla interna, figura 4.3.

Figura 4.3. Sintaxis del indicador USE_NL.

Donde *table* es el nombre de la tabla o el alias de la tabla a ser usada como tabla interna del join con ciclos anidados.

Por ejemplo, en la siguiente sentencia se realiza un join entre las tablas Cuentas y Clientes. Se asume que estas tablas no están almacenadas juntas en un cluster.

```
SELECT a.balance, b.nombre, b.apellido
   FROM cuentas a, clientes b
   WHERE a. cliente_id = b. cliente_id;
```

Después de seleccionar el acceso basado en costos como default, el optimizador elige realizar la operación del join entre ciclos anidados o combinación ordenada (sort-merge) entre las tablas, dependiendo sobre la cual probablemente regrese todos los registros seleccionados por la sentencia, más rápido. Sin embargo, se puede optimizar la sentencia para un mejor tiempo de respuesta, o un mínimo tiempo necesario para regresar el primer registro seleccionado

por la consulta, obteniendo la mejor salida. Se puede forzar al optimizador a escoger un join del tipo nested loops al utilizar el indicador USE_NL. Es la siguiente sentencia, el uso del indicador USE_NL explícitamente elige un join nested loops con la tabla CLIENTES como la tabla interior:

```
SELECT /*+ ORDERED USE_NL(clientes) Usa N-L para conseguir el primer renglón
        más rápido */
    a.balance, b.nombre, b.apellido
    FROM cuentas a, clientes b
    WHERE a.cliente_id = b. cliente_id;
```

En muchos casos, los joins nested loops regresan el primer renglón más rápido que un join del tipo sort-merge. Un join nested loops puede regresar el primer renglón después de leer el primer seleccionado de una de las tablas y el primer renglón que hace match de la otra y combinarlos, mientras un join short-merge no puede regresar el primer registro hasta después de leer y ordenar todos los registros de las tablas y combinar los primeros registros de cada fuenre de registros ordenados.

USE_MERGE

El uso de este indicador hace que Oracle realice el join de la tabla especificada con otra fuente de registros usando un join del tipo sort-merje, figura 4.4.

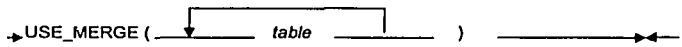


Figura 4.4. Sintaxis del indicador USE_MERGE.

Donde *table* es el nombre de la tabla que se combinará con el resultado de los joins de las tablas previas en el orden de los joins utilizando un join sort-merge.

NO_MERGE

El hint no merge ocasiona que Oracle no combine las vistas combinables, figura 4.5.

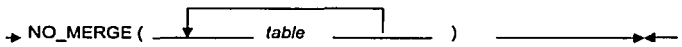


Figura 4.5. Sintaxis del indicador USE_MERGE.

Este hint es más utilizado para reducir el número de posibles permutaciones de una sentencia y hace la optimización más rápido. Este hint no tiene argumentos:

```
SELECT * FROM t1, (SELECT /*+ NO_MERGE */ * from t2) v ...
```

Ocasiona que la vista v no sea combinada.

USE_HASH

Este hint hace que Oracle combine cada tabla especificada con otra fuente de registros mediante un join hash, figura 4.6.

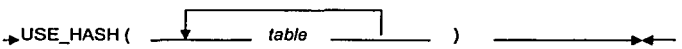


Figura 4.6. Sintaxis del indicador USE_HASH.

Donde *table* es el nombre de la tabla que se combinará con el resultado de los joins de las tablas previas en el orden de los joins utilizando un join sort-merge.

Indicadores Adicionales

CACHE

Este indicador especifica que los bloques recuperados de la tabla, sean colocados primero en la lista de las sentencias más recientemente utilizadas (LRU), de la memoria cache, cuando se realiza una exploración completa de la tabla.
Esta opción es útil para tablas pequeñas. Ejemplo:

```
SELECT /*+ FULL (tabla1) CACHE(tabla1) */
ename
FROM Juan.empleados tabla1;
```

NO CACHE

Este indicador especifica que los bloques recuperados de la tabla, sean colocados al último en la lista de las sentencias más recientemente utilizadas, cuando se realiza una exploración completa de la tabla. Este es el manejo normal de las sentencias.Ejemplo:

```
SELECT /*+ FULL (tabla1) CACHE(tabla1) */
ename
FROM Juan.empleados tabla1;
```

PUSH SUBQ

El indicador PUSH_SUBQ provoca que las subconsultas no combinadas, se realicen lo más pronto posible dentro del plan de ejecución. Normalmente las subconsultas no combinadas en el último paso del plan de ejecución. Si la subconsulta es muy importante y reduce el número de renglones significativamente, se mejoraría el rendimiento, si primero se evalúa la subconsulta.

Considerar una sintaxis alternativa de sentencias SQL

Porque SQL es un lenguaje flexible, más de una sentencia puede satisfacer las necesidades de alguna aplicación. Si bien, dos sentencias SQL producen el mismo resultado, Oracle puede procesar más rápido una que la otra. Se puede utilizar el resultado del comando EXPLAIN PLAN para comparar los planes de ejecución de las dos sentencias y determinar cual es más eficiente. El siguiente ejemplo muestra los planes de ejecución para dos sentencias SQL que realizan la misma función. Ambas sentencias regresan todos los departamentos de la tabla DEPT que no tienen empleados en la tabla EMP. Cada sentencia realiza una búsqueda en la tabla EMP con una subconsulta. Asume que hay un índice, DEPTNO_INDEX, sobre la columna DEPTNO de la tabla EMP.

Esta es la primera sentencia y su plan de ejecución, ver figura 4.7:

```
SELECT dname, deptno
FROM   dept
WHERE  deptno NOT IN
(SELECT deptno FROM emp);
```

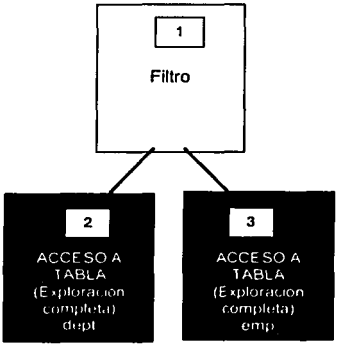


Figura 4.7. Plan de ejecución con dos full table scan.

El paso tres de la salida indica que Oracle ejecuta esta sentencia mediante una consulta completa de la tabla EMP desaprovechando el índice de la columna DEPTNO. Esta consulta completa a la tabla puede ser una operación de consumo de tiempo innecesario. Oracle no usa el índice porque la subconsulta que busca en la tabla EMP no tiene una cláusula WHERE que haga disponible al índice. Sin embargo, esta sentencia SQL selecciona los mismos renglones pero utilizando el índice, ver figura 4.8.

```
SELECT dname, deptno
FROM    dept
WHERE NOT EXISTS  (SELECT deptno
  FROM emp
  WHERE dept.deptno = emp.deptno);
```

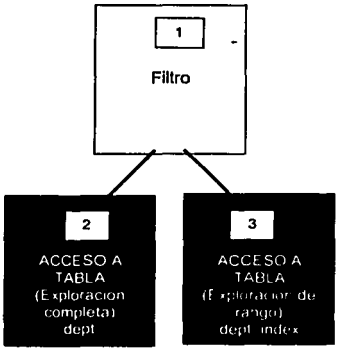


Figura 4.8 Plan de ejecución con una consulta completa a la tabla y una consulta al índice.

La cláusula WHERE de la subconsulta se refiere a la columna DEPTNO de la tabla EMP, así el índice DEPTNO_INDEX es usado. El uso del índice se refleja en el paso tres del plan de ejecución. La consulta de rango al índice DEPTNO_INDEX toma menos tiempo que la consulta completa a la tabla de la primera sentencia. Además, la primera sentencia realiza una consulta completa a la tabla por cada DEPTO de la tabla DEPT. Por estas razones, la segunda sentencia es más rápida que la primera.

Si se presentan sentencias que contengan el operador NOT IN, como el primer query del ejemplo, se debe considerar reescribirla utilizando el operador NOT EXIST. Esto debe habilitar el uso de los índices en algunas sentencias, si este existe, y por lo tanto mejorar el rendimiento.

4.3 Técnicas de diseño y programación de aplicaciones.

Para obtener un buen rendimiento de las aplicaciones, es conveniente manejar una serie de conceptos sobre cómo funciona el optimizador de Oracle, cuándo se pueden usar índices y cuándo interesa evitarlos, o cómo se tratan sentencias en las que aparecen vistas, etc. Es muy importante para el mantenimiento de una aplicación que el código que se ejecuta aproveche la *estrategia* que sigue el optimizador a la hora de elegir un plan óptimo. El impacto que tiene en el rendimiento de una base de datos un buen código es mucho mayor que el que pueda tener cualquiera de los parámetros que se usan para configurar la instancia. Además, el coste que supone el optimizar una aplicación en el momento en que se está desarrollando es también muy inferior al que supondrá mejorar algo cuando esa aplicación esté ya en su fase de producción.

Evitar FULL TABLE SCAN no planificados

Un full table scan lee secuencialmente todos los datos de una tabla, tanto si son relevantes para la sentencia que se ejecuta como si no. Hay dos razones importantes para evitar full table scans innecesarios.

- No son selectivos. Aunque operaciones no selectivas puedan ser apropiadas para grandes tareas por lotes, que manejan muchos datos, son poco apropiadas para aplicaciones online.
- Los datos leídos en un full table scan desaparecen de la SGA muy rápido, ya que se sitúan en la parte más volátil del buffer de datos.

Cuándo se realiza un full table scan

En RBO (Optimización Basada en Reglas), se realizará un full table scan sobre una tabla cuando en una sentencia SQL se dé alguna de las siguientes condiciones:

- No existen índices en la tabla.
- En la sentencia no hay ninguna condición sobre las filas que devuelve.
- No hay ninguna condición sobre la primera columna de algún índice de la tabla.
- Hay alguna condición sobre las primeras columnas de algún índice de la tabla, pero estas condiciones aparecen dentro de expresiones o requieren de alguna conversión implícita.
- Hay alguna condición sobre las primeras columnas de algún índice de la tabla, pero estas condiciones son desigualdades o comprobaciones por NULL o NOT NULL.

Si se está usando CBO(Optimización Basada en Costos), Oracle usará un full table scan en todos los casos anteriores y, además, puede decidir hacerlo en alguno de los siguientes:

- La tabla no ha sido analizada.
- La tabla es pequeña.

- Las columnas indexadas no son selectivas.
- El modo de optimización está puesto a ALL_ROWS.

Recomendaciones para crear índices

Para asegurar que una sentencia puede hacer uso de índices, lo primero que debe considerarse es indexar todas las primary key y foreign key. En una aplicación, la mayor parte de sentencias y joins se hacen usando estas columnas. El orden de las columnas en un índice es una decisión muy importante. Deben considerarse dos factores para decidir sobre este orden. Primero, la columna usada más frecuentemente en las condiciones WHERE debería ser la primera del índice. Segundo, colocar primero la columna más selectiva (aquella con mayor número de valores distintos). La selectividad de la primera columna de un índice es muy importante si se está usando CBO. Hay que asegurarse de que, en las sentencias, las columnas que se están indexando no aparecen dentro de expresiones (funciones TO_CHAR, UPPER, etc.) o en conversiones implícitas (la columna es un VARCHAR2 y se está comparando con un NUMBER). Es frecuente el uso de sentencias en las que se pregunta por un campo nulo, para actualizarlo a continuación. Si la tabla es muy grande y van a recuperarse pocos registros, interesa que ese campo tenga inicialmente un valor por defecto (no nulo), y se pregunte luego por ese valor. Por ejemplo, una tabla CLIENTES con un campo TELEFONO. Este campo se deja inicialmente con NULL y se ejecutan sentencias del tipo

```
select * from clientes where telefono is null;
```

Esta sentencia siempre hará un FULL TABLE SCAN de CLIENTES. Si se ejecuta frecuentemente, y la tabla es muy grande, el impacto sobre la base de datos puede ser considerable. Para evitar esto, puede definirse este campo con un valor por defecto, por ejemplo, '000000000'. La sentencia anterior se puede reescribir de esta forma:

```
select * from clientes where telefono = '000000000';
```

Si hay un índice por el campo TELEFONO, el optimizador puede decidir hacer uso de él (RBO siempre lo usará).

Usar índices selectivos

La selectividad de un índice es la relación entre el número de valores distintos de una columna indexada y el número de registros de la tabla. Si una tabla tiene 1000 registros, y una columna indexada de la tabla tiene 950 valores diferentes, la selectividad del índice es 0.95 (950/1000). La mejor selectividad es 1. Los índices únicos sobre columnas no nulas siempre tienen selectividad 1.

La selectividad de un índice nos da una medida de su utilidad para evitar I/O en la ejecución de sentencias contra la tabla. Si un índice sobre una tabla de1000 registros tiene sólo 5 valores diferentes, entonces su selectividad es muy pobre (5/1000 = 0.005). Para cada posible valor de este índice, habrá un promedio de 200 filas. En este caso, podría ser más interesante realizar un full table scan que acceder vía índice a esta tabla.

Si se está usando CBO, el optimizador no realizará accesos a tablas a través de índices con una selectividad muy baja.

Elección de la primera columna en un índice concatenado

- La primera columna de un índice debería ser la columna más selectiva y también debería ser la más usada en las condiciones que aparezcan para las sentencias sobre esa tabla. Si una columna cumple las dos condiciones, entonces ésa debe ser la primera del índice.

- Si una columna no selectiva es la más frecuentemente utilizada en sentencias sobre tablas grandes, posiblemente sería necesario revisar el diseño de la aplicación o de los datos.

- Si una columna muy selectiva no es muy usada en las condiciones de sentencias, sólo es recomendable crear un índice sobre ella si se trata de una foreign key.

Índices concatenados vs varios índices con una sola columna

Cuando se va a crear un índice concatenado, debe valorarse si la selectividad de ese índice va a ser considerablemente mayor con varias columnas que con una. Por ejemplo, para una tabla de POBLACIONES, un índice por el campo NOMBRE_POBLACION tendrá la misma selectividad que otro por los campos NOMBRE_POBLACION y PROVINCIA (habrá muy pocas poblaciones que se llamen igual en distintas provincias). En este caso, no interesará un índice concatenado, ya que con una sola columna puede localizarse el registro consultado.

Si se tienen varias columnas muy selectivas, podemos pensar en crear varios índices, cada uno de ellos con una columna, si las consultas de nuestra aplicación tienen condiciones de igualdad para esas columnas unidas por AND. Por ejemplo, para sentencias del tipo:

```
SELECT *
FROM emp
WHERE job = 'ANALISTA'
AND deptno = 20;
```

Podría interesar tener dos índices, uno para cada columna. El plan de ejecución sería el siguiente:

```
TABLE ACCESS BY ROWID EMP
AND-EQUAL
INDEX RANGE SCAN EMP$JOB
INDEX RANGE SCAN EMP$DEPTNO
```

Se hace un RANGE SCAN de cada índice y, a continuación, la operación AND-EQUAL, que consiste en obtener la intersección de los RowID que han devuelto los INDEX RANGE SCAN anteriores. Aquí debe valorarse cómo se harán menos lecturas: si en un INDEX RANGE SCAN del índice concatenado, seguido del acceso a tabla por RowID; o con los dos INDEX RANGE SCAN seguidos del acceso a tabla por RowID.

Joins de varias tablas: NESTED LOOPS y MERGE JOINS

En Oracle hay dos tipos de operaciones para resolver un join: NESTED LOOPS y MERGE JOIN (en Oracle 7.3 se añadió una nueva opción, HASH JOIN). MERGE JOIN es una operación de conjuntos, no devuelve registros a la siguiente operación hasta que todas las filas han sido procesadas. Por el contrario, NESTED LOOPS es una operación de fila, devuelve los primeros registros a la siguiente operación tan pronto como hayan sido procesados. Es importante conocer cómo trabajan cada una de estas dos opciones para determinar cuál es la idónea en función del tamaño de las tablas y la naturaleza de la sentencia.

MERGE JOINS

Una operación MERGE JOIN enfrenta los datos resultantes de dos scans. Implica tres pasos:

- TABLE ACCESS FULL de cada tabla del join.
- SORT JOIN para ordenar los dos conjuntos resultado del paso anterior.
- MERGE JOIN para mezclar los resultados del SORT JOIN.

El uso de un MERGE JOIN indica que no existen índices sobre las tablas o que la sintaxis de la sentencia imposibilita el uso de índices. Este tipo de operación no es apropiado para aplicaciones online por varios motivos:

- Lentitud al devolver la primera fila de la sentencia. Debido a que es una operación de conjunto (no de fila) no devuelve filas hasta que han sido procesadas todas.
- Los bloques se cargan en la SGA haciendo un FULL TABLE SCAN, lo que hace que sean los primeros en ser eliminados cuando se necesite espacio en el buffer de datos.
- Puede ser necesario el uso de segmentos temporales para resolver la sentencia, lo que supone un riesgo potencial de contención entre usuarios.

Sin embargo, pueden darse situaciones en que un MERGE JOIN sea la operación más eficiente para resolver un join, como tareas por lotes que traten muchos registros. Un MERGE JOIN será la mejor opción en los mismos casos en que un FULL TABLE SCAN sea la mejor opción: cuando la tabla involucrada sea muy pequeña y cuando la tabla sea extremadamente grande.

Si la tabla es muy pequeña, entonces puede ser más rápido realizar un full table scan que un index range scan seguido de un table access by RowID. Esto es así cuando, por ejemplo, la tabla entera puede ser leída en una única petición de I/O al sistema.

Si la tabla es muy grande, también puede interesar realizar un full table scan por ciertos motivos:

- Si estamos seleccionando un rango de valores de una columna y los registros no se encuentran ordenados físicamente según esta columna. En este caso, una combinación index range scan seguido de table access by RowID puede leer más bloques que un full table scan.

- Los bloques cargados en la SGA no se mantienen ahí durante mucho tiempo, no perjudicando el que se compartan de datos entre procesos.

NESTED LOOPS

Esta es la operación más habitual en que Oracle resuelve los joins. Indica que un índice está disponible para hacer el join. Es una operación de fila: cada fila procesada se devuelve a la siguiente operación, en vez de esperar a que se procese el resultado completo. NESTED LOOPS es una operación muy efectiva para aplicaciones online. Cuando se realiza un NESTED LOOPS, se siguen los siguientes pasos:

- Elección de la tabla sobre la que se hará el FULL SCAN. Esta tabla se llamará directora.
- FULL TABLE SCAN de la tabla directora.
- INDEX RANGE SCAN de la otra tabla.

Si se encuentra alguna coincidencia, un table access by RowID de la segunda tabla.

La elección de la tabla directora en un NESTED LOOPS es crítica para la ejecución de la sentencia. El número de lecturas realizadas por la sentencia depende de qué tabla se tome para realizar el full table scan.

Implicaciones de la tabla directora en un NESTED LOOPS

La clave del buen rendimiento de una operación NESTED LOOPS es el orden en que se realiza el join de las tablas. El número de repeticiones del bucle es el producto del número de registros resultante de la primera tabla por el número de registros de la segunda tabla a los que se accede después. Si se añaden más tablas en el join, la elección de la primera tabla es aún más crítica. Lo conveniente es minimizar el número de registros leídos en los primeros pasos del bucle.

Consideremos un ejemplo. Tenemos una sentencia en la que aparecen 4 tablas (A, B, C y D), todas ellas del mismo tamaño, con las siguientes cláusulas FROM y WHERE:

```
from D, C, B, A
where A.cod = B.cod
and B.cod = C.cod
and C.cod = D.cod
and A.cod = 123
and D.val = 'VALOR'
```

Si las tablas A, B, C y D tienen índices para sus columnas cod, un NESTED LOOP será la operación que usará Oracle para resolver la sentencia. Se hará un join de A con B, del resultado de este join con C y del resultado de este último con D. Finalmente, se aplicará la condición que hay sobre D.val.

Si D.val es una columna selectiva, el rendimiento de la sentencia será mejor si hacemos que D entre en el primer NESTED LOOP de la ejecución. De esta forma, menos registros serán devueltos al siguiente NESTED LOOP y así sucesivamente. La anterior cláusula WHERE puede reescribirse así:

```
from D, C, B, A
where A.cod = B.cod
and B.cod = C.cod
and C.cod = D.cod
and D.cod = 123
and D.val = 'VALOR'
```

Ahora, se hará un join de D con C, del resultado con B y, finalmente, con A.

Veamos qué puede significar esto con números reales. Supongamos que cada tabla tiene exactamente 100 registros y que solamente hay un registro en D con val = 'VALOR'. Un acceso por índice genera un promedio de 2 accesos a las ramas del índice, más 1 acceso a la hoja por cada fila, además de un acceso por RowID a la tabla por cada fila. Por tanto, para leer 100 filas en un acceso por índice se necesitan 2 lecturas para las ramas del índice, más 100 lecturas del índice, más 100 lecturas de la tabla, haciendo un total de 202 lecturas. Para el join de A con B, se necesitan 100 lecturas por cada una de las 202 lecturas que se hicieron para la tabla A.

Para el join original, tendríamos el siguiente número de lecturas:

Operación	Tabla accedida	Lecturas	Lecturas acumuladas
1ª	A	2 + 2 * 100	202
1º join	B	100 * 202	20,402
2ª join	C	100 * 100 * 202	2,040,402
3º join	D	100 * 100 * 100 * 202	204,040,402

Como se ve, un incremento significativo en cada paso penaliza los pasos posteriores. Veamos qué sucede en el caso de la sentencia modificada:

Operación	Tabla accedida	Lecturas	Lecturas acumuladas
1ª	D	2 + 2 * 100	202 (devuelve 1 fila)
1º join	C	1 * 202	404
2ª join	B	100 * 1 * 202	20,604
3º join	A	100 * 100 * 1 * 202	2,040,604

El orden en que se realizan los joins tienen una implicación decisiva en el rendimiento del NESTED LOOP. En este ejemplo, se puede reducir a una centésima parte el número de lecturas realizadas para resolver la sentencia.

Cómo alterar el orden de los joins

En RBO, si se tiene igual oportunidad de usar índice en todas las tablas de un join, la tabla sobre la que se hará FULL SCAN será la que aparezca en último lugar en la cláusula FROM. Si estamos usando CBO, el optimizador tendrá en cuenta el tamaño de las tablas y la selectividad de los índices para elegir la primera tabla.

Si se conoce la naturaleza de los datos, puede interesar decirle al optimizador que siga un orden concreto para realizar los joins. Esto se hace añadiendo hints (pistas) a la sentencia. Algunos de estos hints son:

- ORDERED Se hace el join de las tablas en el orden en que aparecen en la cláusula FROM.
- INDEX Proporciona una lista de índices a utilizar.
- FULL Da una tabla sobre la que se hará FULL TABLE SCAN. Esta tabla será usada como la tabla directora.
- USE_NL Lista las tablas de las que se hará el join usando NESTED LOOP.

Si no es posible mejorar el rendimiento de los joins cambiando el orden de las tablas, puede que interese estudiar la posibilidad de eliminar alguno de estos joins desnormalizando nuestras tablas.

Sentencias SQL que usan vistas

Si una sentencia contiene una vista, el optimizador tiene dos modos de resolverla: resolver primero la vista y después la sentencia, o integrar el texto de la vista con la sentencia. Si se resuelve primero la vista, el conjunto resultado de la vista es calculado primero, y el resto de las condiciones de la sentencia se aplican después como un filtro.

Dependiendo de los tamaños de las tablas involucradas, resolver primero la vista puede degradar el rendimiento de la sentencia; si la vista se integra dentro de la sentencia, las

condiciones de la sentencia pueden aplicarse dentro de la vista y el conjunto resultado que se obtiene será más pequeño. Sin embargo, en determinados casos puede mejorarse el rendimiento separando operaciones de grupo mediante vistas. Cuando se usen vistas en un join, es conveniente tener en cuenta cómo se van a resolver. Si la vista contiene funciones de grupo (GROUP BY, SUM, COUNT, DISTINCT), no podrá integrarse dentro de la sentencia, sino que ha de resolverse primero.

Integración de la vista en la sentencia

Si la vista devuelve un número de filas muy grande, o si este número de filas va a ser filtrado con condiciones adicionales de la sentencia que usa la vista, el rendimiento se verá mejorado permitiendo que la vista se integre dentro de la sentencia. El optimizador realizará esta operación automáticamente siempre que sea posible.

Para evitar tener vistas que no puedan integrarse en la sentencia, deberán evitarse las funciones de grupo en el SQL de la vista. Estas funciones se añadirán posteriormente a la sentencia.

Forzar a que la vista se ejecute por separado

En ciertos casos, puede interesar que el SQL de la vista no se integre con el resto de la sentencia. Por ejemplo, si se está realizando un GROUP BY en un NESTED LOOPS join de dos tablas, la operación de grupo no se completará hasta que haya finalizado el join de las dos tablas. Esto es así incluso si las columnas que se agrupan son todas de la misma tabla. En este caso, interesaría realizar el GROUP BY antes que el join, para que el NESTED LOOP procese menos filas. Esto se consigue añadiendo al SQL de la vista la función GROUP BY. De este modo, el optimizador no podrá integrarla dentro de la sentencia y la ejecuta por separado.

Para una vista que no tiene funciones de grupo, también se puede forzar que no se integre dentro de la sentencia, usando el hint NO_MERGE. Con este hint, el optimizador resolverá la vista primero y después el resto de la sentencia.

No "reciclar" las vistas

Cuando se está trabajando con vistas, es muy importante darle el uso para el que fueron creadas. Nunca crear una vista con un propósito y utilizarlas posteriormente en sentencias en las que su rendimiento puede no ser el adecuado. Una vista no debe ser utilizada por el mero hecho de que los registros que devuelve cumplen unas condiciones; hay que tener muy en cuenta con qué otras vistas / tablas se va a hacer un join. Este mal uso de las vistas puede ser crítico en el rendimiento de las sentencias, ya que es el optimizador quien decide cómo y cuando van a resolverse si forman parte de una sentencia más compleja.

Sentencias SQL con subselects (subconsultas)

Cuando aparecen subselects dentro de una sentencia, podemos encontrarnos con problemas similares a los que se plantean con las vistas, ya que el optimizador decide cuándo resuelve la subselect y si la integra o no con el resto de la sentencia. Además, ciertas subselects, como comprobaciones de existencia, pueden realizarse de un modo muy eficiente usando la cláusula EXISTS en vez del operador IN.

Cómo se resuelve una consulta anidada

Si una sentencia tiene una subselect, el optimizador puede resolverla de dos modos:

- resolver primero la subselect y a continuación el resto de la sentencia (tipo vista), o
- integrar la subselect dentro de la sentencia (tipo join).

Si se resuelve primero la subselect, se calcula primero el resultado de ésta y, a continuación, el resto de las condiciones de la sentencia se aplican como un filtro. Si se integra la subselect con la sentencia, las condiciones de la subselect se unen a las del resto de la sentencia. Si se usan las subselects para comprobaciones de existencia, el rendimiento es mejor si se resuelve por separado. En otro caso, es mejor que la subselect se integre dentro de la sentencia. Cuando la subselect tenga funciones de grupo, siempre se resolverá por separado, limitando las opciones que pudiera tener el optimizador para elegir un plan de ejecución.

Por ejemplo, supongamos que tenemos la siguiente sentencia con una subselect que contiene un DISTINCT:

```
select A.valor_A
from A
where A.cod in
(select DISTINCT B.cod
from B
where valor_B1 = 1
and valor_B2 > 1000);
```

Esta subselect se dice que es de tipo vista, porque tiene una función de grupo y ha de resolverse por separado. Suponiendo que tenemos una clave primaria en A para el campo cod (A_PK), el plan de ejecución sería el siguiente:

```
NESTED LOOPS
VIEW
SORT UNIQUE
TABLE ACCESS FULL B
TABLE ACCESS BY ROWID A
INDEX UNIQUE SCAN A_PK
```

En esta sentencia, la subselect actúa como la tabla directora de un NESTED LOOPS join. El optimizador no tiene opción de elegir qué tabla interesa tomar como la directora del NESTED LOOPS. Sin embargo, la sentencia anterior puede reescribirse para evitar la subselect y tener un join de dos tablas:

```
select A.valor_A
from A, B
where A.cod = B.cod
and B.valor_B1 = 1
and B.valor_B2 > 1000;
```

El nuevo plan de ejecución sería el siguiente:

```
NESTED LOOPS
TABLE ACCESS FULL B
TABLE ACCESS BY ROWID A
INDEX UNIQUE SCAN A_PK
```

La tabla directora del NESTED LOOPS sigue siendo B, pero con la nueva sintaxis el optimizador puede decidir qué tabla elige, en función de los tamaños que tengan en cada momento.

Sugerencias (Hints) para subselects que devuelvan un valor máximo

Es muy frecuente el uso de sentencias con subselects que devuelven el valor máximo (o mínimo) de un campo. En estos casos, puede reducirse considerablemente el número de lecturas si se utilizan hints (sugerencias) para decir al optimizador cómo debe ejecutar la sentencia.

Supongamos una tabla A, que entre otros tiene los campos valor, cod y fecha. Creamos un índice concatenado por los campos cod y fecha. Tenemos la siguiente sentencia:

```
select valor
from A A1
where cod = 1
and fecha = (select max(A2.fecha)
from A A2
where A2.cod = A1.cod);
```

El plan de ejecución para esta sentencia es el siguiente:

```
FILTER
TABLE ACCESS FULL A
SORT AGGREGATE
INDEX RANGE SCAN A$COD_FECHA
```

Para resolver la sentencia, se recorre el índice A$COD_FECHA, y ordena todos los valores para extraer el máximo. Ahora bien, los valores que se están ordenando sabemos que ya están ordenados en el índice, luego podríamos indicarle al optimizador que no realice ese paso y que el índice lo recorra de mayor a menor (por defecto, lo hace de menor a mayor). Reescribimos la sentencia y añadimos un hint para mostrarle al optimizador el camino que debe seguir:

```
select /*+ INDEX_DESC(A A$COD_FECHA) */ valor
from A
where cod = 1
and fecha < sysdate
and rownum = 1;
```

El plan de ejecución ahora queda así:

```
COUNT STOPKEY
TABLE ACCESS BY ROWID A
INDEX FULL SCAN DESCENDING A$COD_FECHA
```

El número de filas devuelto es uno, y esa fila es la que tiene el máximo valor de la fecha para el código solicitado.

Cómo combinar subselects

Una única sentencia puede contener más de una subselect. Cuanto mayor sea el número de subselects, más difícil será integrarlas dentro de un único join. Para evitar esto, es conveniente combinar varias subselects siempre que sea posible. Supongamos que tenemos tres tablas A, B y C. La tabla A es muy grande y vamos a hacer una actualización masiva sobre ella. Las tablas B y C son tablas pequeñas, que contienen sólo códigos y descripciones. Tenemos la siguiente sentencia:

```
update A
set valor = 123
where cod1 in
(select cod from B
where desc = 'COD1')
and cod2 in
(select cod from C
where desc = 'COD2')
and fecha < sysdate;
```

Para cada registro de la tabla A, se ejecutarán las dos subselects. Puede evitarse realizar varias subselects por registro combinando las dos subselects en una sola:

```
update A
set valor = 123
where (cod1, cod2) = ANY
(select B.cod, C.cod
from B, C
where B.desc = 'COD1'
and C.desc = 'COD2')
and fecha < sysdate;
```

El resultado de este subselect es el producto cartesiano de los registros de B y C que cumplen las condiciones sobre los campos desc. Si B y C son tablas pequeñas, la sobrecarga que supone este join es despreciable si la comparamos con la mejora que se consigue haciendo únicamente un subselect.

Validaciones de existencia

Una subselect no siempre devuelve filas, en ocasiones sólo se usan para validar si un dato existe o no. En estos casos, pueden usarse los operadores EXISTS y NOT EXISTS, en vez de IN y NOT IN, para mejorar el rendimiento de la sentencia. De esta forma, se evitan accesos innecesarios a la tabla (no se devuelve la fila que tenemos en la tabla, sólo la condición de que exista o no). La siguiente sentencia:

```
select valor
from A
where cod IN
(select cod from B);
```

puede reescribirse de la siguiente manera, usando el operador EXISTS:

```
select valor
from A
where EXISTS
(select 1 from B
where B.cod = A.cod);
```

El operador EXISTS sólo necesita validar que exista un registro devuelto por la subselect. Una sentencia con NOT IN puede ser reescrita para usar el operador NOT EXISTS. La mejora que se puede conseguir usando (NOT) EXISTS puede ser enorme. El operador (NOT) IN hace FULL TABLE SCANS anidados. Sin embargo, usando (NOT) EXISTS, la sentencia puede hacer uso de índices en los campos que aparezcan en el WHERE de la subselect.

Transacciones grandes

Una transacción es un conjunto de operaciones que se ejecutan todas (o no) como una unidad. Para terminar la transacción, se puede ejecutar cualquiera de los comandos commit o rollback. Cuando una sesión comienza, una transacción Oracle le asigna un segmento de rollback. La información necesaria para deshacer esa transacción se va guardando en el segmento de rollback asociado. Varias transacciones pueden estar escribiendo simultáneamente en el mismo segmento de rollback, incluso en el mismo extent, pero nunca en el mismo bloque. Una vez terminada la transacción (commit o rollback), el espacio que ocupa su información de rollback queda marcado como reutilizable. Sin embargo, Oracle no lo utiliza de inmediato, intenta maximizar el tiempo que se conserva esa información.

Los segmentos de rollback no sólo se utilizan para escribir información que se utilizaría para dar marcha atrás a la transacción. También permiten construir una vista consistente de los datos que están siendo modificados por una transacción, tal como estaban en un momento en el tiempo anterior al inicio de esa transacción. Supongamos que se ejecuta una SELECT que tarda mucho tiempo en completarse. Antes de que se termine de ejecutar, se comienza una transacción que modifica bloques que necesita leer la SELECT. Si estos bloques no pueden ser reconstruidos (con información de los segmentos de rollback) a la situación en que estaban en el momento en que empezó la SELECT, se produce un error ORA-01555: Snapshot too old, y la SELECT es cancelada. Esta imposibilidad de recrear el estado anterior de los bloques se debe a que la información de los segmentos de rollback ha sido sobrescrita (la transacción terminó, y ese espacio quedó disponible para ser usado).

Este doble uso que hace Oracle de los segmentos de rollback nos obliga a considerar de una forma especial los procesos que se vayan a ejecutar en la base de datos y que realizan actualizaciones masivas (tardan mucho tiempo en completarse y modifican un gran número de bloques). Si el proceso sólo hace commit (o rollback) al final y genera mucha información de rollback, puede que aborte porque su segmento de rollback no tenga espacio suficiente. Esto supone que todo el trabajo que había realizado la transacción hasta ese punto se pierde y hay que volver a empezar (podría demorar un proceso muchas horas). Por el contrario, si el proceso hace commit (o rollback) con mucha frecuencia, estamos permitiendo reutilizar el espacio que ocupa su información en los segmentos de rollback, con lo que la probabilidad de un ORA-01555 es mayor. Este error también lo puede dar la propia transacción, si es frecuente que visite repetidamente los mismos bloques.

Conocer este funcionamiento nos puede ayudar en el momento de codificar los procesos que realizan grandes transacciones. Si el rollback que genera cabe en un único segmento, podemos asignárselo al comienzo de la transacción (SET TRANSACTION USE ROLLBACK SEGMENT ...) y no hacer commit hasta el final. De todos modos, no debe suponerse que en ese segmento sólo va a escribir una transacción. Si el espacio que hay para segmentos de rollback es insuficiente, habrá que fraccionar la transacción y hacer commit periódicamente. Esto se consigue definiendo un cursor que seleccione los registros a modificar y recorrerlos en un bucle, tal como muestra el siguiente código:

```
DECLARE
CURSOR C IS select rowid from A
where FECHA > sysdate - 365;
cont NUMBER := 0;
reg C%rowtype;
BEGIN
FOR reg IN C LOOP
  cont := cont + 1;
  update A set CAMPO = 'VALOR' where rowid = reg.rowid;
  IF cont > 10000 THEN
    COMMIT;
```

```
      cont := 0;
    END IF;
  END LOOP;
  COMMIT;
  END;
```

Aquí se hacen FETCH's entre COMMIT's. Esta técnica no es ANSI-SQL, pero Oracle la permite si el cursor no es un SELECT ... FOR UPDATE. El inconveniente es que los registros que se van a modificar no son bloqueados previamente, y otro proceso podría sobreescribir las modificaciones. Si no se tiene la certeza de que no haya otros procesos que puedan interferir, es conveniente realizar un bloqueo antes de modificar los datos. Esto se puede hacer reescribiendo el código anterior del siguiente modo:

```
DECLARE
CURSOR C1 IS select rowid from A
where FECHA > sysdate - 365;
CURSOR C2(r ROWID) IS select rowid from A
where rowid = r FOR UPDATE;
cont NUMBER := 0;
reg1 C1%rowtype;
reg2 C2%rowtype;
BEGIN
FOR reg1 IN C1 LOOP
   cont := cont + 1;
   FOR reg2 IN C2(reg1.rowid) LOOP
    update A set CAMPO = 'VALOR'
    where rowid = reg2.rowid;
   END LOOP;
   IF cont > 10000 THEN
      COMMIT;
      cont := 0;
   END IF;
END LOOP;
COMMIT;
END;
```

Aquí, los registros son bloqueados de uno en uno (se podría hacer con un rango). El COMMIT se hace después de cerrar el cursor que tiene el SELECT ... FOR UPDATE.

Si el cursor no devuelve los registros en el mismo orden en que se encuentran físicamente, la probabilidad de que se intente leer en distintos momentos el mismo bloque aumenta (aumentando la probabilidad del error ORA-01555). Si forzamos a usar siempre el mismo segmento de rollback, será el propio proceso quien sobreescriba la información de rollback que va generando, aumentando también la probabilidad de este error.

En resumen, a la hora de programar grandes transacciones hay que elegir con qué problema nos vamos a encontrar. A priori, podremos conocer el volumen de nuestra transacción. Si es desmesuradamente grande, se bebería decidir fraccionarla programando un cursor. Esto evitará que, después de varias horas, nuestra transacción termine mal y haga rollback de todo el trabajo. Pero nos podremos encontrar con el ORA-01555. No se puede garantizar que este error no va a darse, sólo puede minimizarse la probabilidad de que aparezca. Si nuestro proceso está bien diseñado (así como los datos), que se dé este error no hará que se pierda todo el tiempo (y trabajo) transcurrido: debería bastar con relanzarlo y que continúe por donde se quedó.

Accesos a tablas muy grandes

Cuando nuestra aplicación hace accesos a tablas pequeñas, los bloques leídos en el buffer de datos de la SGA tienen una probabilidad muy alta de ser compartidos por varios procesos. La situación ideal sería aquella en que toda nuestra aplicación, con sus datos, pudiese estar cargada en ese buffer: los procesos nunca necesitarían leer de los ficheros de datos.

Según van creciendo las tablas de nuestra aplicación, esta situación ideal ya no es posible. La posibilidad de reutilizar bloques ya leídos decrece considerablemente. Además, el tener cargados en memoria muchos bloques de una tabla muy grande, y que permanezcan ahí mucho tiempo, puede perjudicar al resto de usuarios. Esto es lo que sucede cuando se leen bloques vía un INDEX RANGE SCAN poco selectivo. Aunque pueda parecer contradictorio, en ciertos casos, un acceso por índice resulta perjudicial para la gestión de la SGA.

Esto hace que las lecturas de tablas muy grandes necesiten un enfoque diferente cuando se quiera mejorar el rendimiento de la aplicación.

Proximidad de los datos

Si se pretende acceder a grandes tablas mediante índices, interesa que los datos estén físicamente ordenados según el criterio en que se vayan a recuperar. Esto no es posible controlarlo totalmente, pero sí podemos aumentar la probabilidad de encontrar los registros relacionados dentro del mismo bloque. Esto se consigue ordenando los registros que se desea insertar antes de ser insertados.

Supongamos una aplicación que recupera filas de una tabla muy grande, y que siempre interesa recuperar aquellos datos que se encuentran dentro de un rango de fechas, con una sentencia del tipo:

```
select * from tabla where fecha between :fecha1 and :fecha2;
```

Si hay un índice por el campo FECHA, el optimizador puede decidir utilizarlo para ejecutar esta sentencia (si se tiene RBO, siempre lo usará). Si los datos están muy disgregados, en el peor de los casos se necesitará leer tantos bloques como registros recupera el SELECT. Sin embargo, si cuando se insertaron los datos estaban ordenados por la fecha, la posibilidad de que con un mismo bloque se recupere más de un sólo registro es considerablemente mayor. La necesidad de lecturas físicas en este caso será mucho menor.

INDEX SCANS vs FULL TABLE SCANS

Si se va a leer de tablas muy grandes, no siempre debe asumirse que un acceso por índice va a dar un rendimiento mejor que un full table scan. UNIQUE SCANS y RANGE SCANS de índices que no sean seguidos de accesos a tabla suelen funcionar bien, pero un RANGE SCAN de un índice, seguido de acceso a tabla por RowID puede resultar peor que un FULL TABLE SCAN. Esto es más cierto cuanto más grande sea la tabla.

Supongamos una tabla de 10,000,000 de filas, y una sentencia del tipo:

```
select * from tabla where campo between :min and :max;
```

que va a recuperar 1,000,000 de filas (el 10% de la tabla). Para esta tabla, tenemos un índice por CAMPO. Con RBO, siempre se va a usar ese índice; CBO puede decidir que es mejor acceder directamente a la tabla.

Supongamos que en cada bloque del índice caben 100 registros y que en cada bloque de la tabla caben 10. Del índice habría que leer 10,000 bloques (1,000,000 filas / 100 filas por bloque).

Suponiendo que la tabla estuviese físicamente ordenada por CAMPO (este sería el mejor de los casos), tendríamos que leer 100,000 bloques (1,000,000 filas / 10 filas por bloque). Si la tabla no está ordenada, en el peor de los casos habría que leer 1,000,000 de bloques (1 bloque por fila). En total, si la tabla está ordenada, habrá que leer 110,000 bloques (10,000 del índice, más 100,000 de la tabla); si no está ordenada, se necesita leer 1,010,000 bloques (10,000 del índice, más 1,000,000 de la tabla).

Si se accede directamente a la tabla, un full table scan leerá 1,000,000 de bloques. Esto es mejor que el caso menos favorable accediendo por índice. Además, si se accedió por índice, los bloques se mantendrán en la SGA tanto tiempo como sea posible, posiblemente perjudicando al resto de procesos. Si tenemos un tamaño de bloque de 2K, en el mejor de los casos se intentaría conservar en memoria 220M (2K * 110,000 bloques). En cambio, si se hizo un full table scan, sólo se intenta mantener en la SGA tantos bloques como indique el parámetro db_file_multiblock_read_count (suele tener un valor entre 8 y 64). Los datos que cargaron el resto de procesos pueden seguir conviviendo en la memoria.

Crear tablas completamente indexadas

Cuando se tiene accesos a tablas muy grandes, es interesante no realizar dos operaciones para leer los datos (index range scans y table access by rowid), si se puede resolver la sentencia con una sola operación (index only scans). Si las columnas más frecuentemente seleccionadas de una tabla son relativamente estáticas, podemos considerar crear un índice que contenga todas las columnas que aparezcan en la cláusula WHERE, seguidas de las columnas que aparezcan en la SELECT.

Una sentencia del tipo:

```
select campo1, campo2, campo3 from tabla
where campo4 = :var1 and campo5 = :var2;
```

puede resolverse leyendo sólo del índice, si éste contiene las columnas (por este orden) CAMPO4, CAMPO5, CAMPO1, CAMPO2 y CAMPO3.

Crear Hash Clusters

En un hash cluster, el orden en que se insertan los registros en la tabla no importa; la ubicación física del registro se determina en base a los valores de columnas clave. Se aplica una función hash a los valores de la clave de una fila, y Oracle usa el resultado para saber en qué bloque debe ser almacenada. El uso de un hash cluster hace innecesario un índice cuando se busca por los valores de la clave.

Puede ser conveniente usar hash clusters en las siguientes circunstancias:

- Se usan sentencias con condiciones de igualdad sobre los campos de la clave (where ID = :valor). Cuando se ejecuta la sentencia, el optimizador aplica la función hash sobre la clave y calcula directamente el bloque en el que se encuentra el registro.
- No hay posibilidad de forzar a que los datos estén físicamente ordenados. Si continuamente se recuperan rangos de valores para una columna, y los datos no están ordenados, la posibilidad de reutilizar bloques ya cargados en la SGA es muy baja. En este caso, el uso de un hash cluster puede ser apropiado.

El espacio de almacenamiento no supone un problema para nuestro sistema, ya que un hash cluster necesita, aproximadamente, un 50% más de espacio que una tabla indexada.

Crear tablas particionadas

Si una tabla muy grande puede verse como varias particiones lógicas (registros que comparten el mismo valor para una columna clave), puede considerarse la posibilidad de partirla en varias tablas más pequeñas. Esas tablas pequeñas pueden consultarse juntas haciendo un UNION ALL de varias selects. Esto es lo que se llama particionamiento horizontal.

El uso de tablas particionadas es interesante, por ejemplo, en grandes tablas históricas que tienen que almacenar datos de varios años. Puede definirse como una tabla particionada, cada partición guardando un trimestre. En este caso, la clave del particionamiento sería un campo fecha. Si la aplicación suele consultar esta tabla restringiendo la columna clave a un valor o un rango, eliminar todas las particiones de la tabla menos una puede suponer una drástica disminución del número de bloques necesarios para resolver la consulta. En cambio, si frecuentemente es necesario leer de más de una partición, puede que la mejora no sea considerable.

Por otra parte, el uso de tablas particionadas supone un beneficio importante para la administración ya que:

- en caso de desastre, la recuperación puede ser más rápida que si la tabla no está particionada;
- es más fácil hacer backup de una partición que de toda la tabla;
- puede balancearse la I/O creando cada partición en un disco diferente;
- cuando se realizan borrados masivos de registros basados en la columna clave del particionamiento, puede diseñarse para que baste borrar la partición (esto afecta sólo al diccionario de datos, es muy eficiente y no genera log ni rollback).

Implementar Parallel Query

Cuando se están realizando full scans de tablas muy grandes, puede mejorarse el rendimiento si se hace uso de la opción Parallel Query de Oracle. En Oracle7, no se puede paralelizar un INDEX SCAN.

4.4 Ejemplos y técnicas de optimización de sentencias SQL.

Como hemos visto el RDBMS puede obtener la misma información de diferentes formas, y unas son más rápidas que otras, la tarea del desarrollador es encontrar la correcta para cada caso. Los siguientes ejemplos son muestra de cómo una sentencia SQL puede ser reescrita para obtener los mismos resultados en tiempos distintos, no necesariamente, modificar una sentencia reducirá el tiempo de respuesta, si se hace mal, tal vez lo incremente.

- > versus >=

Dada la siguiente sentencia:

```
select * from tab where x > 3
```

cúando hay un índice en la columna x. Esta sentencia trabaja utilizando el índice para encontrar el primer valor de donde x = 3, y continua explorando hacia adelante, ya que todos los registros que siguen son mayores a 3.

Suponga que hay muchos registros en la tabla *tab* donde x es igual a 3, en este caso, el RDBMS tiene que explorar varias páginas antes de encontrar el primer registro donde x sea mayor a 3. Es más eficiente escribir la sentencia así:

```
select * from tab where x >= 4
```

De esta manera se evitará el RDBMS el explorar valores innecesarios antes de encontrar el primer registro mayor a tres, porque ahora tiene que encontrar primero el primer registro igual a cuatro y por lo tanto este y los siguientes son mayores a 3.

- Ejemplos de la utilización de las cláusulas "AND" y "OR" en el rendimiento de las aplicaciones, los tiempo son relativos, ya que la cantidad de información y recursos es variable, sólo se muestran como referencia de como es que pueden variar:

```
    SELECT  . . . .
    FROM    EMP E
    WHERE   EMP_SALARIO > 50,000
    AND     EMP_TIPO    = 'GERENTE'
    AND     25          < ( SELECT  COUNT(*)
                            FROM    EMP
                            WHERE   EMP_GTE = E.EMP_NO );
Total CPU = 156.3 Sec


    SELECT  . . . .
    FROM    EMP E
    WHERE   25          < ( SELECT  COUNT(*)
                            FROM    EMP
                            WHERE   EMP_GTE = E.EMP_NO )
    AND     EMP_SALARIO > 50,000
    AND     EMP_TIPO    = 'GERENTE;

Total CPU = 10.6 Sec
```

```
    SELECT  . . . .
    FROM    EMP E
    WHERE ( EMP_SALARIO > 50,000
        AND EMP_TIPO    = 'GERENTE' )
    OR      25          < ( SELECT  COUNT(*)
                            FROM    EMP
                            WHERE   EMP_GTE = E.EMP_NO );
Total CPU = 28.3 Sec

    SELECT  . . . .
    FROM    EMP E
    WHERE   25          < ( SELECT  COUNT(*)
                            FROM    EMP
                            WHERE   EMP_GTR = E.EMP_NO )
    OR    ( EMP_SALARIO > 50,000
        AND EMP_TIPO    = 'GERENTE' );
Total CPU = 101.6 Sec
```

- Una columna de tipo numérico puede causar problemas, la siguiente sentencia:

```
SELECT . . .
FROM   EMP
WHERE  EMP_NO = '123';
```

Será procesada como:

```
SELECT . . .
FROM   EMP
WHERE  EMP_NO = TO_NUMBER('123');
```

Y esta otra:

```
SELECT . . .
FROM   EMP
WHERE  EMP_TYPE = 123;
```

Será procesada como:

```
SELECT . . .
FROM   EMP
WHERE  TO_NUMBER(EMP_TYPE) = 123;
```

Por lo tanto se debe convertir el valor constante que se va a comparar al mismo tipo del campo, para evitar problemas de conversión de tipo.

- Combinar sentencias SELECT no relacionadas. De las siguientes tres sentencias podemos generar una sola:

```
SELECT NAME
FROM   EMP
WHERE  EMP_NO = 1234;

SELECT NAME
FROM   DPT
WHERE  DPT_NO = 10;
```

```
SELECT  NAME
FROM    CAT
WHERE   CAT_TYPE = 'RD'   ;
```

De las sentencias anteriores podemos generar la siguiente:

```
SELECT E.NAME, D.NAME, C.NAME
FROM   CAT  C,
       DPT  D,
       EMP  E,
       DUAL X
WHERE  NVL('X', X.DUMMY) = NVL('X', E.ROWID (+))
AND    NV Language L('X', X.DUMMY) = NVL('X', D.ROWID (+))
AND    NVL('X', X.DUMMY) = NVL('X', C.ROWID (+))
AND    E.EMP_NO    (+)     = 1234
AND    D.DEPT_NO   (+)     = 10
AND    C.CAT_TYPE (+)      = 'RD';
```

De esta manera si las sentencias se repitieran 1000 de veces, se realizarían 3000 accesos a la base de datos, pero si se realizará la unión sólo se accesaría 1000 veces.

- Ejemplo de como combinar dos cláusulas "Like" utilizando un "decode".

```
SELECT COUNT(*), SUM(SALARY)
FROM   EMP
WHERE  DEPT_NO   =  0020
AND    EMP_NAME LIKE 'SMITH%' ;

SELECT COUNT(*), SUM(SALARY)
FROM   EMP
WHERE  DEPT_NO   =  0030
AND    EMP_NAME LIKE 'SMITH%' ;
```

De las sentencias anteriores se puede "armar" una

```
SELECT COUNT(DECODE(DEPT_NO, 0020, 'X',     NULL)) D0020_KOUNT,
       COUNT(DECODE(DEPT_NO, 0030, 'X',     NULL)) D0030_KOUNT,
       SUM   (DECODE(DEPT_NO, 0020, SALARY, NULL)) D0020_SAL,
       SUM   (DECODE(DEPT_NO, 0030, SALARY, NULL)) D0030_SAL
FROM   EMP
WHERE  EMP_NAME LIKE 'SMITH%';
```

Hay que tomar en cuenta que en la sentencia anterior, se realiza una exploración completa de la tabla EMP, por lo tanto se recomienda utilizarla en tablas que no sean de gran tamaño, y se produce el mismo efecto de reducción de accesos a la base de datos que en el ejemplo anterior.

- Usando "decode" en las cláusulas "order by" o "group by" puede habilitar el uso compartido de sentencias SELECT similares. Como en los siguientes ejemplos:

```
SELECT . . .
FROM   EMP
WHERE  EMP_NAME LIKE 'SMITH%'
ORDER
    BY DECODE(:VAR1 ,'E', EMP_NO, 'D', DEPT_NO);
```

```
SELECT . . .
FROM   EMP
WHERE  EMP_NAME  LIKE  'SMITH%'
GROUP  BY DECODE(:VAR2,'E',EMP_NO,'D',DEPT_NO);
```

En la primer sentencia se usa la variable: PAR1 para cambiar el orden de los registros obtenidos por la columna EMP_NO o DEPT_NO, y en el segundo para agrupar por una u otra columna.

- Borrar registros duplicados de una tabla:

```
DELETE FROM EMP E
WHERE  E.ROWID > ( SELECT MIN(X.ROWID)
                   FROM   EMP X
                   WHERE  X.EMP_NO = E.EMP_NO );
```

Haciendo uso del identificador de registro, si hay dos valores repetidos, no hay dos identificadores iguales, por lo tanto uno de ellos se borra.

- Probar EXIST en lugar de JOINS entre tablas. En las siguientes sentencias se utilizan joins:

```
SELECT . . .
FROM   DEPT D,
       EMP  E
WHERE  E.DEPT_NO  = D.DEPT_NO
AND    E.EMP_TYPE = 'MANAGER'
AND    D.DEPT_CAT = 'A';

 SELECT . . .
 FROM   DEPT D,
        EMP E
 WHERE  E.DEPT_NO = D.DEPT_NO
 AND   ( E.EMP_TYPE = 'MANAGER'
 OR D.DEPT_CAT = 'A'      );
```

Para mejorar las sentencias anteriores podemos cambiarlas para implementar el EXIST de la siguiente manera:

```
SELECT . . .
FROM   EMP E
WHERE  EXISTS      ( SELECT 'X'
                     FROM   DEPT
                     WHERE  DEPT_NO  = E.DEPT_NO
                     AND    DEPT_CAT = 'A' )
AND    E.EMP_TYPE = 'MANAGER'


SELECT . . .
FROM   EMP E
WHERE  E.EMP_TYPE = 'MANAGER'
OR     EXISTS      ( SELECT 'X'
                     FROM   DEPT
                     WHERE  DEPT_NO  = E.DEPT_NO
                     AND    DEPT_CAT = 'A' )
```

El EXIST es sólo una alternativa, para un JOIN, se debe probar de ambas formas para saber cual es más adecuada para el diseño de tablas e índices del sistema en cuestión.

- Probar el uso de EXISTS en lugar de DISTINCT. Ejemplo: uso de DISTINCT:

```
SELECT DISTINCT DEPT_NO, DEPT_NAME
FROM   DEPT D,
       EMP E
WHERE  D.DEPT_NO = E.DEPT_NO ;
```

Uso de EXIST

```
SELECT DEPT_NO, DEPT_NAME
FROM   DEPT D
WHERE  EXISTS ( SELECT 'X'
                FROM   EMP E
                WHERE  E.DEPT_NO = D.DEPT_NO );
```

Una alternativa del DISTINCT dentro de un JOIN es el EXIST que accesa de forma diferente a la base de datos.

- Reducción de costos generales del SQL, utilizando funciones almacenadas "en línea". En el siguiente ejemplo se realiza un join entre tres tablas, con el empleo de dos funciones podemos eliminarlos y obtener una sentencia más simple:

```
SELECT H.emp_no,     E.emp_name,
       H.hist_type, T.type_desc,
       count(*)
From   history_type T,
       emp E,
       emp_history H
Where  H.emp_no      = E.emp_no
And    H.hist_type   = T.hist_type
Group  By H.emp_no, E.emp_name, H.hist_type, T.type_desc ;
```

Esta sentencia puede ser mejorada utilizando funciones, como las siguientes:

```
FUNCTION Lookup_Hist_Type (typ IN number) return varchar2
AS
  tdesc   varchar2(30);
  CURSOR  C1 IS
  SELECT  type_desc
  FROM    history_type
  WHERE   hist_type = typ;
BEGIN
  Open  C1;
  Fetch C1 into tdesc;
  Close C1;
  return (nvl(tdesc,'?'));
END;
```

```
FUNCTION Lookup_Emp (emp IN number) return varchar2
AS
  ename    varchar2(30);
  CURSOR   C1 IS
  SELECT   emp_name
  FROM     emp
  WHERE    emp_no = emp;
BEGIN
  Open  C1;
  Fetch C1 into ename;
  Close C1;
  return (nvl(ename,'?'));
END;
```

Utilizando las funciones anteriores, la sentencia original se cambiaría por la siguiente:

```
Select H.emp_no,     Lookup_Emp(H.emp_no),
       H.hist_type, Lookup_Hist_Type(H.hist_type),
       Count(*)
From    emp_history H
Geoup by H.emp_no, H.hist_type;
```

- Reducir joins utilizando funciones almacenadas "en línea"

```
Select E.emp_no, E.emp_name,
       sum(S.days) sick_days,
       sum(H.days) holidays
From   holiday_leave H,
       sick_leave S,
       emp E
Where  E.emp_no = :emp_no
And    E.emp_no = S.emp_no (+)
And    E.emp_no = H.emp_no (+)
Group By E.emp_no, E.emp_name
```

Las columnas sumadas son ineficientes, porque se generan operaciones de reunión (joins) innecesarias, en lugar de utilizar múltiples sentencias SQL para cambiar este SQL, podemos utilizar funciones, como las siguientes:

```
FUNCTION Sum_Sick_Leave (emp  IN number) return number
AS
  tot_days  number := 0;
  CURSOR C1 IS
  SELECT sum(days)
  FROM   sick_leave
  WHERE  emp_no = emp;
BEGIN
  Open  C1;
  Fetch C1 into tot_days;
  Close C1;
  return (tot_days);
END;

FUNCTION Sum_Holiday_Leave (emp IN number) return number
AS
  tot_days  number := 0;
```

```
  CURSOR C1 IS
  SELECT sum(days)
  FROM   holiday_leave
  WHERE  emp_no = emp;
BEGIN
  Open  C1;
  Fetch C1 into tot_days;
  Close C1;
  return (tot_days);
END;
```

El SQL original quedaría de la siguiente manera, como observamos, es correcto y más sencillo.

```
SELECT E.emp_no, E.emp_name,
       sum_sick_leave(E.emp_no) sick_days,
       sum_holiday_leave(E.emp_no) holidays
FROM   emp E
WHERE  E.emp_no = :emp_no
```

- Utilizando funciones pueden solucionarse las limitaciones del "outer join", como en el siguiente ejemplo:

```
SELECT E.emp_no, E.emp_name, H.hist_date, H.Hist_Type
From   emp_historty H,
       emp E
Where  E.emp_no         = :emp_no
And    E.emp_no         = H.emp_no (+)
And    H.hist_date (+) = ( SELECT MAX(hist_date)
And                          From   emp_history
                             Where  emp_no = E.emp_no );
```

Esta sentencia es ilegal, se puede corregir usando funciones.

```
FUNCTION Max_Emp_History (emp IN number) return date
AS
  max_dte date;
  CURSOR  C1 IS
  SELECT  MAX(hist_date)
  FROM    emp_history
  WHERE   emp_no = emp;
BEGIN
  Open  C1;
  Fetch C1 into max_dte;
  Close C1;
  return (max_dte);
END;

SELECT E.emp_no, E.emp_name, H.hist_date, H.Hist_Type
From   emp_historty H,
       emp E
Where  E.emp_no         = :emp_no
And    E.emp_no         = H.emp_no (+)
And    H.hist_date (+) = Max_Emp_History (E.emp_no);
```

- Minimizar el número de subconsultas. Observe el siguiente ejemplo:

```
SELECT EMP_NAME
FROM   EMP
WHERE  EMP_CAT    = ( SELECT MAX(CATEGORY)
                      FROM   EMP_CATEGORIES )
AND    SAL_RANGE  = ( SELECT MAX(SAL_RANGE)
                      FROM   EMP_CATEGORIES )
AND    EMP_DEPT   =   0020;
```

Es mejor especificar los dos valores en una sola subconsulta, de la siguiente manera:

```
SELECT EMP_NAME
FROM   EMP
WHERE (EMP_CAT, SAL_RANGE)
                = ( SELECT MAX(CATEGORY), MAX(SAL_RANGE)
                    FROM   EMP_CATEGORIES )
AND    EMP_DEPT = 0020;
```

Para sentencias UPDATE multicolumna, también es mejor especificar también un sola subconsulta:

```
UPDATE EMP
SET    EMP_CAT   = ( SELECT MAX(CATEGORY)
                     FROM   EMP_CATEGORIES ),
       SAL_RANGE = ( SELECT MAX(SAL_RANGE)
                     FROM   EMP_CATEGORIES )
WHERE  EMP_DEPT  = 0020;
```

Probar:

```
UPDATE EMP
SET   (EMP_CAT, SAL_RANGE)
                = ( SELECT MAX(CATEGORY), MAX(SAL_RANGE)
                    FROM   EMP_CATEGORIES )
WHERE  EMP_DEPT = 0020;
```

- Probar el uso de NOT EXIST en lugar de NOT IN. Observe el siguiente ejemplo:

```
SELECT . . .
FROM   EMP
WHERE  DEPT_NO NOT IN ( SELECT DEPT_NO FROM   DEPT
                        WHERE  DEPT_CAT = 'A');
```

Para mejorar el rendimiento reemplace este código por el siguiente:

```
SELECT . . .
FROM   EMP E
WHERE  NOT EXISTS ( SELECT  X'
                    FROM   DEPT
                    WHERE  DEPT_NO  = E.DEPT_NO
                    AND    DEPT_CAT = 'A'   );
```

- Un ejemplo de una "consulta en paralelo". Los indicadores full y parallel, indican al optimizador a realizarla en paralelo y no seguir el camino tradicional.

```
SELECT /*+ full(H) parallel(H, 8) */
       H.emp_no,     Lookup_Emp(H.emp_no),
       H.hist_type, Lookup_Hist_Type(H.hist_type),
       count(*)
FROM   emp_history H
GROUP  BY H.emp_no, H.hist_type;
```

- Ejemplo de como compartir el mismo cursor para dos sentencias parecidas. Observe estas sentencias:

```
SELECT EMP_NAME, SALARY, GRADE
FROM   EMP
WHERE  EMP_NO = 0342;

SELECT EMP_NAME, SALARY, GRADE
FROM   EMP
WHERE  EMP_NO = 0291;
```

Ahora se encuentran compartiendo un mismo cursor en un procedimiento almacenado o una función, se reducen accesos:

```
DECLARE
   CURSOR C1 (E_NO  NUMBER)  IS
   SELECT EMP_NAME, SALARY, GRADE
   FROM   EMP
   WHERE  EMP_NO = E_NO;
BEGIN
   OPEN   C1 (342);
   FETCH  C1 INTO ..., ..., ...;
   .
   .
   .
   OPEN   C1 (291);
   FETCH  C1 INTO ..., ..., ...;
   CLOSE  C1;
END;
```

Y aquí se encuentran en un mismo query o consulta:

```
SELECT A.EMP_NAME, A.SALARY, A.GRADE,
       B.EMP_NAME, B,SALARY, B.GRADE
FROM   EMP A, EMP B
WHERE  A.EMP_NO = 0342
AND    B.EMP_NO = 0291 ;
```

- Ejemplos de sintaxis de las sugerencias o indicadores de ejecución SQL:

Utilizar la optimización basada en reglas:

```
SELECT  /*+ RULE */  . . . .
FROM    EMP, DEPT
WHERE   . . .
```

Los indicadores se pueden utilizar en las sentencias SELECT, DELETE y UPDATE:

```
SELECT  /*+ hint text */  . . . .
DELETE  /*+ hint text */  . . . .
UPDATE  /*+ hint text */  . . . .
```

Para cada sentencia se puede usar un indicador distinto, en el siguiente ejemplo hay una sentencia incluida dentro de otra y las dos tienen diferente indicador de ejecución.

```
SELECT   /*+ RULE */  . . . .
FROM     EMP
WHERE    EMP_STATUS = 'PART-TIME'
AND      EXISTS      ( SELECT  /*+ FIRST_ROWS */  'x'
                       FROM    EMP_HISTORY
                       WHERE   EMP_NO = E.EMP_NO
                       AND     EMP_STATUS != 'PART-TIME' )
```

En este ejemplo se realiza la primera sentencia de la unión por reglas y la segunda por costos optimizando la ejecución para obtener todos los renglones lo más rápido posible.

```
SELECT   /*+ RULE */  . . . .
FROM     EMP
WHERE    EMP_STATUS = 'PART-TIME'
UNION
SELECT   /*+ ALL_ROWS */  . . . .
FROM     EMP_HISTORY
WHERE    EMP_STATUS != 'PART-TIME'
```

- Query's usados para identificar la pobreza del rendimiento de las sentencias SQL.

El siguiente ejemplo muestra las sentencias que tienen un porcentaje de efectividad menor a 80%, para saber cuales son correctas sólo tenemos que invertir el signo de menor a mayor y asignar un porcentaje tan elevado como queramos. Lo ideal es que el "hit_ratio" sea de uno. Nos muestra: el número de ejecuciones, lecturas a disco, bufers utilizó y el porcentaje de efectividad en base a los bufers leídos y los accesos a disco, entre menos accesos a disco haya mejor será el rendimiento.

```
SELECT executions, disk_reads, buffer_gets,
       round((buffer_gets - disk_reads)
            / buffer_gets, 2) hit_ratio,
       sql_text
FROM   v$sqlarea
WHERE  executions  > 0
AND    buffer_gets > 0
AND    (buffer_gets - disk_reads) / buffer_gets < 0.80
ORDER
    BY 4 desc ;
```

Ejemplo:

```
SQL_TEXT   EXECUTIONS  READS_PER_RUN  DISK_READS BUFFER_GET HIT_RATIO  SQL_TEXT
-----------------------------------
SELECT …        1           442           442        763        .42  SELECT …
SELECT …        1           380           380        408        .07  SELECT …
select …        1           366           366        367          0  select …
select …        1           196           196        371        .47  select …
```

En los puntos suspensivos continua realmente el texto de la sentencia. Los porcentajes de rendimiento son realmente bajos en este caso. En este ejemplo, se muestra también las lecturas a disco promedio, en base al número de ejecuciones y el número de accesos a disco totales, aquí se visualizarán las sentencias con un rendimiento superior al 95 %:

```
SELECT sql_text, executions,
       round(disk_reads / executions, 2) reads_per_run,
       disk_reads, buffer_gets,
       round((buffer_gets - disk_reads)
           / buffer_gets, 2) hit_ratio,
       sql_text
FROM   v$sqlarea
WHERE  executions  > 0
AND    buffer_gets > 0
AND    (buffer_gets - disk_reads) / buffer_gets > 0.950
ORDER  BY 3 desc ;
```

Ejemplo:

```
EXECUTIONS DISK_READS BUFFER_GET HIT_RATIO  SQL_TEXT
---------- ---------- ---------- ---------- ---------
1          1          46         .98        select …
111        40         1703       .98        select …
627        57         2386       .98        begin  sp_cierre …
22        158       6506         .98        SELECT …
```

en los puntos suspensivos continua realmente el texto de la sentencia. Se puede observar un rendimiento del 98% en estos ejemplos, vemos además, que se incluyen procedimientos almacenados, como el de la tercera línea.

- Probar usar joins (cruce de tablas) en lugar de la cláusula: EXIST. Observe el siguiente ejemplo:

```
SELECT EMP_NAME
FROM   EMP E
WHERE  EXISTS ( SELECT 'X'
                FROM   DEPT
                WHERE  DEPT_NO  = E.DEPT_NO
                AND    DEPT_CAT = 'A');
```

Para mejorar el rendimiento probar con la siguiente sentencia:

```
SELECT EMP_NAME
FROM   DEPT D,
       EMP  E
WHERE  E.DEPT_NO  = D.DEPT_NO
AND    D.DEPT_CAT = 'A';
```

Esto depende del tamaño de las tablas, los índices y de la forma de optimización que se va a utilizar, es por eso que se recomienda obtener el plan de ejecución de las posibilidades para obtener así la mejor.

- Probar el uso de la UNIÓN o el IN en lugar del OR. En los siguientes ejemplos se utiliza la cláusula OR para obtener información de una u otra condición, pero no siempre es la mejor opción...

```
SELECT . . .
  FROM   LOCATION
  WHERE  LOC_ID = 10
  OR     REGION = 'MELBOURNE'
```

```
SELECT . . .
  FROM   LOCATION
  WHERE  LOC_ID = 10
  OR     LOC_ID = 20
  OR     LOC_ID = 30
```

Para mejorar el rendimiento se pueden reestructurar las sentencias para hacer lo mismo de una manera más óptima:

```
  SELECT . . .
    FROM   LOCATION
    WHERE  LOC_ID = 10
UNION
  SELECT . . .
    FROM   LOCATION
    WHERE  REGION = 'MELBOURNE'

SELECT . . .
  FROM   LOCATION
  WHERE  LOC_IN IN (10,20,30)
```

- Probar usar UNIÓN ALL en lugar de UNIÓN cuando sea posible (UNON ALL no filtra registros repetidos):

```
SELECT ACCT_NUM, BALANCE_AMT
FROM   DEBIT_TRANSACTIONS
WHERE  TRAN_DATE = '31-DEC-95'
UNION
SELECT ACCT_NUM, BALANCE_AMT
FROM   CREDIT_TRANSACTIONS
WHERE  TRAN_DATE = '31-DEC-95'
```

Para mejorar el rendimiento se puede sustituir la sentencia anterior por la siguiente:

```
SELECT ACCT_NUM, BALANCE_AMT
FROM   DEBIT_TRANSACTIONS
WHERE  TRAN_DATE = '31-DEC-95'
UNION ALL
SELECT ACCT_NUM, BALANCE_AMT
FROM   CREDIT_TRANSACTIONS
WHERE  TRAN_DATE = '31-DEC-95'
```

- Ejemplos de cláusulas WHERE alternativas. Observamos que hay muchas formas de hacer lo mismo pero algunas son más optimas que otras, un buen camino para saberlo es probándolas...

Sustituir la función de cadenas SUBSTR por una cláusula LIKE. Ejemplo: que el número de cuenta empiece con 'CAPITAL':

```
SELECT ACCOUNT_NAME, TRANS_DATE, AMOUNT
FROM   TRANSACTION
WHERE  SUBSTR(ACCOUNT_NAME,1,7) = 'CAPITAL';

SELECT ACCOUNT_NAME, TRANS_DATE, AMOUNT
FROM   TRANSACTION
WHERE  ACCOUNT_NAME LIKE 'CAPITAL%';
```

- Que la cantidad de la transacción sea diferente de cero (no puede ser menor):

```
SELECT ACCOUNT_NAME,TRANS_DATE,AMOUNT
FROM   TRANSACTION
WHERE  AMOUNT != 0;
SELECT ACCOUNT_NAME,TRANS_DATE,AMOUNT
FROM   TRANSACTION
WHERE  AMOUNT > 0 ;
```

- Observar las transacciones que se realizaron durante el día actual:

```
SELECT ACCOUNT_NAME, TRANS_DATE, AMOUNT
FROM   TRANSACTION
WHERE  TRUNC(TRANS_DATE) = TRUNC(SYSDATE);

SELECT ACCOUNT_NAME, TRANS_DATE, AMOUNT
FROM   TRANSACTION
WHERE  TRANS_DATE BETWEEN TRUNC(SYSDATE)
AND TRUNC(SYSDATE) + .99999;
```

- Para entender la segunda sentencia en los siguientes ejemplos se muestra el resultado se sumar a un dato de tipo fecha las cantidades: .99999 y .999999:

```
SELECT TO_DATE('01-JAN-93') + .99999
FROM   DUAL;
```

regresa: '01-JAN-93 23:59:59'

```
SELECT TO_DATE('01-JAN-93') + .999999
FROM   DUAL;
```

regresa: '02-JAN-93 00:00:00'

- Como evitar una concatenación:

```
SELECT ACCOUNT_NAME, TRANS_DATE, AMOUNT
FROM   TRANSACTION
WHERE  ACCOUNT_NAME || ACCOUNT_TYPE = 'AMEXA';

SELECT ACCOUNT_NAME, TRANS_DATE, AMOUNT
FROM   TRANSACTION
WHERE  ACCOUNT_NAME ='AMEX'
AND    ACCOUNT_TYPE = 'A' ;
```

- Quitar operaciones redundantes:

```
SELECT ACCOUNT_NAME, TRANS_DATE, AMOUNT
FROM   TRANSACTION
WHERE  AMOUNT + 3000 < 5000;

SELECT ACCOUNT_NAME, TRANS_DATE, AMOUNT
FROM   TRANSACTION
WHERE  AMOUNT < 2000;
```

- Cambiar una igualdad por un LIKE, cuando se utiliza la función NVL y se espere un posible valor nulo:

```
SELECT ACCOUNT_NAME, TRANS_DATE, AMOUNT
FROM  TRANSACTION
WHERE ACCOUNT_NAME = NVL(:ACC_NAME, ACCOUNT_NAME);


SELECT ACCOUNT_NAME, TRANS_DATE, AMOUNT
FROM  TRANSACTION
WHERE ACCOUNT_NAME LIKE NVL(:ACC_NAME, '%');
```

- Usar procedimientos almacenados, además de mejorar el rendimiento de la aplicación, también reduce costos de mantenimiento, porque si hay que cambiar el proceso, sólo se modifica el procedimiento almacenado, se recompila y la aplicación no sufre cambios, no es necesario generar un nuevo ejecutable, ni instalar nuevamente a los usuarios del sistema. Además de evitar tráfico en la red, en aplicaciones remotas esto se nota más, porque la información no viaja del cliente al servidor, la información se procesa en el servidor y al final sólo se muestran resultados al cliente.

Tabas desnormalizadas:

El uso de tablas desnormalizadas es útil cuando la información que contengan no tiene que ser actualizada de forma constante, para evitar problemas de desactualización y de rendimiento al llenarlas. En el caso que se necesiten realizar múltiples joins para obtener la información requerida, entre muchas tablas, y más cuando se necesita desarrollar, por ejemplo: una aplicación de reportes dinámicos, que tiene que accesar y dejar de accesar diferentes tablas constantemente, lo mejor es hacer una tabla desnormalizada que contenga todos los elementos informativos que necesitamos, incluyendo campos calculados, si es necesario. En la figura 4.9 se muestran los campos que se requieren y enseguida se muestran las tablas que dan origen a esta.

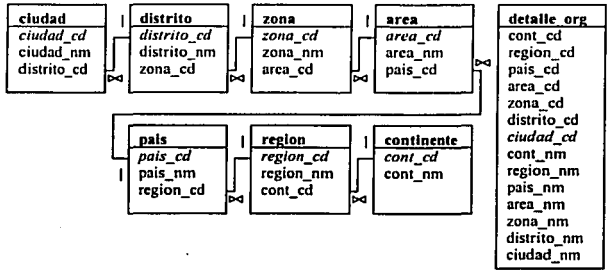


Figura 4.9. Tabla desnormalizada de organización: detalle_org y tablas que le dan origen: ciudad, distrito, zona, area, pais, region, continente.

Sentencias SQL para llenar esta tabla:

```
Delete from nivel ;
commit;

Insert into nivel
select  h.cont_cd,  g.region_cd,  f.pais_cd,  a.area_cd,     b.zona_cd,
d.distrito_cd,    e.ciudad_cd,    h.cont_nm,    g.region_nm,    f.pais_nm,
a.area_nm, b.zona_nm, d.distrito_nm, e.ciudad_nm
from area a, zona b, distrito d, ciudad e, pais f, region g, division h
where a.area_cd=b.area_cd and
      b.zona_cd=d.zona_cd and
      d.distrito_cd=e.distrito_cd and
      a.pais_cd = f.pais_cd and
      f.region_cd = g.region_cd and
     g.cont_cd = h.cont_cd        ;
commit;
```

De esta manera al realizar reportes o procesos en los que se involucre el el nivel de organización no se tiene que hacer join para cada tabla que se necesite. Se tendría que hacer un SQL dinámico muy complicado o varios SQLs que contemplen todos los casos. Además de complicado, tomaría más tiempo ejecutar una sentencia con muchos joins que una sentencia con uno que contemple todas las caracteristicas del nivel de organización. Al utilizar tablas desnormalizadas debemos tener en cuenta que tienen un tiempo de actualización diferente al de cada catálogo de forma individual, estas tablas se actualizan diariamente para este ejemplo, porque se utilizan para construir reportes y procesos diarios. No es necesario que los datos se

actualicen en linea. Si crearamos otra tabla desnormalizada, por ejemplo, para productos y en ella incluyeramos todas las características del producto y se incluyera también un factor de conversión para cada producto que al multiplicarse por su venta la conviertiera en otras unidades de volumen (ej. Litros, galones, etc..), esto es mucho más rápido que si se realiza una función para convertir el volumen normal a otras unidades, porque la función accesaría a otras tablas para realizar esta operación y al utilizar la tabla de productos solo se realiza una multiplicación entre el factor y el campo del volumen de la tabla de ventas sin accesar a otras tablas por registro para obtener el mismo resultado. Las tablas desnormalizadas tienen muchas aplicaciones, pero se deben manejar con mucho cuidado para mantener actualizado su contenido según sean los requerimientos de tiempo, además se debe tomar en cuenta el espacio que ocupa, sólo se recomienda que sea para consolidar catálogos que no contienen un gran volumen de registros.

Manejo de imagenes.

Una forma alternativa de manipular las imagenes en una base dedatos es almacenando sólo el nombre de la imagen en un campo de texto, la ruta de almacenamiento se guarda en un archivo de tipo "ini", si la aplicación ya tiene uno, solo se debe incluir. Así las operaciones de altas y bajas serán mucho más rápidas.

Límites de la configuración de la base de datos.

Cuando se realizan consultas con un volumen elevado de información, aveces se quedan "colgadas", es decir no se ejecutan rápidamente, esto es debido a que si la consulta realiza cálculos de agrupamiento, ordenación u otros, los espacios de memoria reservados para estas operaciónes pueden estar llenos y el RDBMS, empieza a realizar operaciones de subir y bajar información de la memoria al disco. Lo que hace que el proceso en un volumen grande de información se vea bastante afectado.Una alternativa de solución, es reduciendo el conjunto de registros sobre los cuales se opera, hasta llegar a la unidad óptima de ejecución. Se puede iniciar con una cantidad minima, despues se incrementa en intervalos adecuados al volumen de información hasta llegar al volumen máximo, en que la sentencia se ejecuta sin problemas. Una vez encontrado un conjunto de registros óptimo se "arma" una sentencia con una union de varias consultas que agrupen en conjunto la totalidad de registros a procesar. De esta manera se procesarán las consultas por separado de forma rápida y al final se unirán los resultados; obteniendose así el resultado deseado de una forma mucho más rápida.Ejemplo: considerando las siguientes tablas (Figura 4.10).

Ventas	***Productos***
*fecha	*prod_cd
*ciudad_cd	*marca_cd
*prod_cd	
*venta	

Figura 4.10 Ventas y Productos.

Tomando en cuenta que la tabla de ventas tiene un volumen importante de información de más de 1 millón de registros. Si se tiene que realizar una consulta para obtener los códigos de las ciudades que tuvieron ventas para todo el año, agrupados por mes y solo para las marcas con código 99, se realizaría una consulta como la siguiente:

```
Select /*+ ALL_ROWS*/ distinct a.ciudad_cd, a.mes
from   Ventas a
where  fecha >= '01/jan/00'
and     fecha <= '31/dec/00'
and       exists ( Select '' from productos b where b.marca_cd = '99' and
b.prod_cd = a.prod_cd );
```

Esta consulta seguramente se tardaría demaciado porque los espacios de memoria destinados para hacer las operaciones del distinct se llenan, debido a que tiene que contener muchísimos datos para luego seleccionar los distintos. Por lo tanto la mejor opción es encontrar la cantidad óptima para es sistema particular que se utilice. Una solución, es iniciar con la mínima cantidad, y luego agregar más hasta que la consulta se tarde más de lo "normal", lo normal pueden ser algunos segundos o minutos, deende de los recursos del sistema. En el ejemplo se encontró que el tamaño óptimo para manejar esta consulta es un rango de 11 días, por lo que se optó por realizar una consulta con varias uniones. Cada consulta se realiza independientemente y no satura la memoria reservada para relizar las operaciones, por lo tanto no realiza operaciones de I/O que alenten la operación y la consulta se ejecuratá mucho más rápido. La siguiente consulta, resuelve el problema, aunque no parece muy estética, es muy efectiva, para mejorar el tiempo de respuesta. Hay una unión de tres consultas para cada mes. Y son 12 consultas en total, para todo el año.

```
Select /*+ ALL_ROWS*/ distinct a.ciudad_cd   , '1'
from   ventas a
where  fecha >= '01/jan/00'
and      fecha <= '10/jan/00'
and      exists ( Select '' from productos b where b.marca_cd= '99' and    b.prod_cd =
a.prod_cd )
union
Select /*+ ALL_ROWS*/ distinct a.ciudad_cd   , '1'
from   ventas a
where  fecha >= '11/jan/00'
and      fecha <= '20/jan/00'
and      exists ( Select '' from productos b where b.marca_cd= '99' and    b.prod_cd =
a.prod_cd )
union
Select /*+ ALL_ROWS*/ distinct a.ciudad_cd   , '1'
from   ventas a
where  fecha >= '21/jan/00'
and      fecha <= '31/jan/00'
and      exists ( Select '' from productos b where b.marca_cd= '99' and    b.prod_cd =
a.prod_cd ) ;
```

Y así, hasta el último mes, diciembre:

```
Select /*+ ALL_ROWS*/ distinct a.ciudad_cd   , '12'
from   ventas a
where  fecha >= '01/dec/00'
and      fecha <= '10/dec/00'
and      exists ( Select '' from productos b where b.marca_cd= '99' and    b.prod_cd =
a.prod_cd )
union
Select /*+ ALL_ROWS*/ distinct a.ciudad_cd   , '12'
from   ventas a
where  fecha >= '11/dec/00'
and      fecha <= '20/dec/00'
and      exists ( Select '' from productos b where b.marca_cd= '99' and    b.prod_cd =
a.prod_cd )
union
Select /*+ ALL_ROWS*/ distinct a.ciudad_cd   , '12'
from   ventas a
where  fecha >= '21/dec/00'
and      fecha <= '31/dec/00'
and      exists ( Select '' from productos b where b.marca_cd= '99' and    b.prod_cd =
a.prod_cd ) ;
```

De esta manera se ejecutaría la consulta por partes, que se pueden menejar sin problemas en memoria, luego se unen los resultados al final por medio de uniones y se obtiene el mismo resultado, pero mucho más rápido.

Convertir sumas verticales en horizontales.

Es útil utilizar la función DECODE, para realizar varias operaciones de sustitución de valores, funciona de manera similara un "IF" de lenguaje "C" u otro lenguage que utilice esta estructura de control. Puede utilizarce de la siguientes maneras:

1. Select DECODE (**valorX, valorC, valorr1, valorr2**) From tabla.

Donde:
valorX, puede ser el nombre de un campo de la tabla o una constante.
valorC, es el valor contra el que se compara *valorX.*
valor1, si el valorX es igual al valorC, éste valor es el resultado de la función.
valor2, si el valorX es diferente al valorC, éste valor es el resultado de la función.

2. Select DECODE (**valorX, valorC1, valorR1, valorC2,valorR2 ,...**) From tabla.

Donde:
valorX, puede ser el nombre de un campo de la tabla o una constante.
ValorC1, es el primer valor contra el que se compara *valorX.*
valorR1, si el valorX es igual al valorC1, éste valor es el resultado de la función.
ValorC2, es el segundo valor contra el que se compara *valorX.*
ValorR2, si el valorX es igual al valorC2, éste valor es el resultado de la función.

Pueden colocarse varias comparaciones, como sea necesario. Una aplicación práctica de esto es, por ejemplo, si para algún reporte se necesita mostrar las ventas de todo un año, de forma horizontal, de enero a diciembre, y en nuestras tablas tenemos guardada la información e la siguiente manera, figura 4.11:

VentasMensuales
*Anio
*Mes
*ciudad_cd
*prod_cd
*venta

Figura 4.11 Ventas Mensuales.

Es decir, la información de las ventas por mes esta guardada de forma vertical, una sóla columna para todas las ventas de todos los meses, no una columna de venta para cada mes, que sería lo ideal para este caso del reporte. Una sentencia "natural" para obtener la información sería:

```
Select anio, mes, ciudad_cd,prod_cd, venta
From VentasMensuales
Where anio = 2002
Order by mes;
```

Que nos arrojaría resultados de las ventas de forma vertical, es decir, primero mostraría las ventas de enero, febrero, hasta diciembre. Y se tendía que realizar un proceso para que en el reporte se visualizara la información de forma horizontal. Para solucionar este problema puede utilizarse la función DECODE, que nos serviría para separar las ventas de cada mes y mostrarlas al mismo tiempo en una columna diferente, esto se realiza de la siguiente manera:

```
Select mes, ciudad_cd, prod_cd, sum( decode( mes,1,venta,0 )),
sum( decode( mes,2,venta,0 )), sum( decode( mes,3,venta,0 )),
sum( decode( mes,4,venta,0 )), sum( decode( mes,5,venta,0 )),
sum( decode( mes,6,venta,0 )), sum( decode( mes,7,venta,0 )),
sum( decode( mes,8,venta,0 )), sum( decode( mes,9,venta,0 )),
sum( decode( mes,10,venta,0 )), sum( decode( mes,11,venta,0 )),
sum( decode( mes,12,venta,0 ))
From VentasMensuales
Where anio = 2002
Group by mes, ciudad_cd, prod_cd;
```

Con esta sentencia, se generan 12 columnas para las ventas, y sólo se muestra en cada una de ellas la suma del mes correspondiente, es decir, en enero sólo el volumen de enero; hasta diciembre, sólo el volumen de este mes. En la sentencia no se realiza una restricción para algún mes, específico, se selecciona todo el año, para incluir todos los meses, y como estamos realizando sumas es necesario agrupar el resultado. De esta manera llevamos las ventas que se guardan de forma vertical a un despliegue horizontal útil en ciertos casos. Otra aplicación es la de realizar sustituciones de valor, cuando por alguna razón se necesita mostrar un valor por otro.

Caso práctico.

En el siguiente ejemplo se verifica la diferencia de tiempos entre tomar una u otra alternativa como solución para resolver un problema. Se tomarán en cuenta las siguientes tablas, creadas en Oracle 8.0:

Tabla: PRODUCTOS

Campo	Tipo
PRODUCTO_CD	NUMBER(4)
PRODUCTO_NM	VARCHAR2(20)

Tabla: TIENDAS

Campo	Tipo
TIENDA_CD	NUMBER(4)
TIENDA_NM	VARCHAR2(20)

Tabla: PRD_X_TND (productos por tienda)

Campo	Tipo
PRODUCTO_CD	NUMBER(4)
TIENDA_CD	NUMBER(4)
STATUS	VARCHAR2(1)

Tabla: VENTAS

Campo	Tipo
FECHA	DATE
PRODUCTO_CD	NUMBER(4)
TIENDA_CD	NUMBER(4)
VENTA	NUMBER(10,2)

Estas tablas tienen 100, 30, 3000 y 93000 registros respectivamente. Las pruebas se realizaron en una maquina con las siguientes características:

Procesador:	Pentium III a 750 Mhz
Sistema Operativo:	Windows 98
Memoria RAM:	128 MB
Disco Duro:	30 GB

La consulta siguiente:

```
SQL> Select to_char(sysdate,'hh:mi:ss') from dual;

TO_CHAR(
--------
02:07:11

SQL> Select a.producto_cd, sum(a.venta)
  2  From   ventas a, prd_x_tnd b
  3  where  fecha between '01-01-2001' and '10-01-2001'
  4  and    a.tienda_cd   = b.tienda_cd
  5  and    a.producto_cd = b.producto_cd
  6  and    b.status      = 'N'
  7  group by a.producto_cd;

PRODUCTO_CD SUM(A.VENTA)
----------- ------------
          1        30000
          2        30000
          3        30000
          4        30000
          5        30000

SQL> Select to_char(sysdate,'hh:mi:ss') from dual;

TO_CHAR(
--------
02:07:24
```

toma 13s en ejecutarse. Obteniendo el mismo resultado, pero ajustando la sentencia SQL:

```
SQL> Select to_char(sysdate,'hh:mi:ss') from dual;

TO_CHAR(
--------
02:10:11

SQL> select /*+ ALL_ROWS */ a.producto_cd, sum(a.venta)
  2  from   ventas a
  3  where  fecha >= '01-01-2001'
  4  and    fecha <= '10-01-2001'
  5  and    exists ( select /*+ FIRST_ROWS */ 'x'
  6    from   prd_x_tnd b
  7    where  b.status      = 'N'
  8    and    b.producto_cd = a.producto_cd
  9    and    b.tienda_cd   = a.tienda_cd )
 10  group by a.producto_cd;

PRODUCTO_CD SUM(A.VENTA)
----------- ------------
          1        30000
          2        30000
          3        30000
          4        30000
          5        30000

SQL> Select to_char(sysdate,'hh:mi:ss') from dual;

TO_CHAR(
--------
02:10:23
```

Se necesitan 12 segundos para resolver esta sentencia.

Si cambiamos la condición del status, como hay un mayor número de registros que cumplen la condición se obtienen los siguientes tiempos:

Query1: 14 s.

```
SQL> Select to_char(sysdate,'hh:mi:ss') from dual;

TO_CHAR(
--------
02:12:05

SQL> Select a.producto_cd, sum(a.venta)
  2  From   ventas a, prd_x_tnd b
  3  where  fecha between '01-01-2001' and '10-01-2001'
  4  and    a.tienda_cd   = b.tienda_cd
  5  and    a.producto_cd = b.producto_cd
  6  and    b.status      = 'Y'
  7  group by a.producto_cd;

no rows selected

SQL> Select to_char(sysdate,'hh:mi:ss') from dual;

TO_CHAR(
--------
02:12:19
```

Query2: 9s

```
SQL> Select to_char(sysdate,'hh:mi:ss') from dual;

TO_CHAR(
--------
02:12:24

SQL> select /*+ ALL_ROWS */ a.producto_cd, sum(a.venta)
  2  from   ventas a
  3  where  fecha >= '01-01-2001'
  4  and    fecha <= '10-01-2001'
  5  and    exists ( select /*+ FIRST_ROWS */ 'x'
  6    from   prd_x_tnd b
  7    where  b.status      = 'Y'
  8    and    b.producto_cd = a.producto_cd
  9    and    b.tienda_cd   = a.tienda_cd )
 10  group by a.producto_cd;

no rows selected

SQL> Select to_char(sysdate,'hh:mi:ss') from dual;

TO_CHAR(
--------
02:12:33
```

De esta manera verificamos que efectivamente se obtienen tiempos diferentes entre una y otra sentencia, que al final entregan el mismo resultado, y uno de estos tiempos es menor (Query 2), lo que nos lleva a afirmar que si se optimizan todas las sentencias que integran un sistema, este mejorará su velocidad de respuesta. Sólo es cuestión de identificar las sentencias que puedan transformarse y probar diferentes soluciones para estar seguros que es la mejor.

Conclusiones.

El lenguaje estándar para los RDBMS, SQL, también se encuentra en proceso de evolución y se esta desarrollando la tercera versión del lenguaje, que integra elementos para la tecnología orientada a objetos, con lo que se confirma la evolución de los RDBMS, en Oracle 8i ya se incluyen los primeros elementos de esta tecnología. La arquitectura cliente servidor, también ha mejorado para beneficio de los sistemas de información y así poder tener un mejor acceso a los datos, además, de forma más abierta entre plataformas y más rápido, ayudando un poco con la concurrencia de los múltiples usuarios de la base de datos. Con esto se puede concluir que la tecnología evoluciona de forma muy rápida y no se debe perder el paso de sus cambios para saber utilizar y aprovechar al máximo sus avances.

Se demuestra la importancia de conocer que es la optimización de aplicaciones y los conceptos que esta envuelve, ya que al entenderlos se debe aprovechar mejor el empleo de las herramientas para mejorar el tiempo de respuesta de los sistemas. Es indispensable saber que elementos debemos de tomar en cuenta y hasta que punto, para evitar trabajar de más y en deterioro del sistema más que en su beneficio, desafortunadamente la gran mayoría de los sistemas que se encuentran operando funcionan "correctamente", pero no de la mejor manera, esto significa que no se utilizan los recursos al máximo, para obtener mejor beneficio, porque la prioridad de muchas empresas es que sus sistemas funcionen, no que funcionen de la mejor manera posible.

Sería importante incluir en los libros de análisis, diseño y desarrollo de sistemas un apartado en cada etapa que tocara el tema de la optimización, que se ajustaran las etapas para aprovechar al máximo los recursos. Y que esto se aplicara en el desarrollo de los sistemas, y al terminar el desarrollo se diera un tiempo de optimización o "tunning"; de esta manera se estaría asegurando que el sistema funciona aprovechando correctamente todos los recursos y que su velocidad de respuesta es el óptimo.

Algunas empresas, para mejorar el rendimiento de los sistemas invierten en equipo de computo más robusto y obviamente el desempeño mejora, pero a veces con el mismo equipo y algunos cambios en la forma de obtener la información de la base de datos puede igualar, incluso superar el rendimiento de un equipo más robusto. Por lo tanto, conocer lo que implica y lo que es la optimización abre un panorama más amplio al momento de mejorar el rendimiento de los sistemas, porque hay muchos caminos para obtener lo mismo, solo hay que buscarlos y probarlos para encontrar el que más se adecue al que necesitamos.

Se a visto en el presente trabajo conceptos muy importantes en el proceso de optimizar una aplicación, como la forma en que funciona el optimizador de Oracle, las reglas que se proponen para llevarla a cabo y de forma especial se trata el aspecto de la optimización de la memoria, que aunque es trabajo del DBA, es importante la colaboración de quién desarrolla la aplicación para su mejor rendimiento, ya que un uso correcto de la memoria ayuda en gran medida en el rendimiento de las sentencias y por lo tanto de la aplicación.

Es un hecho que el diseño y configuración de la base de datos es un factor determinante en el rendimiento de la aplicación independientemente del desarrollo de la misma, es necesario que esta etapa se realice de la mejor manera para no enfrentar dificultades a la hora de desarrollar la aplicación. Si la base de datos esta mal configurada o mal diseñada provocará problemas de rendimiento a la aplicación, aunque esta sea bien desarrollada, lo ideal es que la configuración y el diseño de la base de datos sean lo más adecuado para el tipo de aplicación que se desee realizar.

Para asegurar un correcto dimensionamiento de los objetos, una correcta configuración y una correcta conjunción entre todos los objetos, las pruebas de rendimiento se deben efectuar

en un ambiente lo más parecido al real, es decir, con una cantidad de información suficiente a la que se enfrentará el sistema en producción (esto es, cuando los usuarios lo utilicen normalmente), ya que si se realizan pruebas con poca información no se asegurará que el rendimiento será óptimo cuando la cantidad de información aumente.

De esta manera, con una construcción óptima de la aplicación se obtendrá el máximo provecho de las herramientas que se tienen y el rendimiento general de la aplicación será mejor, esto es pensando en que se desarrollará una aplicación desde cero, pero si toda la parte del diseño y construcción de tablas, índices, etc., ya existe, entonces tenemos que explotar de la mejor manera el diseño existente.

Después de analizar los conceptos anteriores y de analizar como se desarrollan los sistemas en el mundo real, se puede concluir que la mayoría los sistemas que sólo se desarrollan para cumplir con el objetivo de funcionar, pueden mejorar su rendimiento, accesando su información de la forma más óptima, sin incrementar recursos de hardware. Mejorando los accesos a la información, una correcta configuración y administración de la base de datos, pueden mejorar notablemente el rendimiento de las aplicaciones que trabajan sobre un RDBMS, independientemente del RDBMS de que se trate, ya sea ORACLE, SQL Server, Informix, Sybase, etc. Como se observó en el ejemplo (página 127, *caso práctico*), la reducción de tiempo al sustituir una sentencia por otra, es muy significativa, podría ser casi inperceptible. Pero, cuando son varias sentencias las que se mejoran (o es una sola, pero que se repite muchas veces), sumando el poco tiempo que este pueda ser, se alcanza a disminuir un tiempo significativo, que al final se refleja en la velocidad de respuesta de las aplicaciones.

No existe una cultura de la optimización en México en la mayoría de los sistemas de información desarrollados, y creo que no hay nada más valiosos que el tiempo, y si los procesos que funcionan se tardan por ejemplo 6 horas, quizas se tarden menos de una hora si se realizan de forma diferente; en la que el RDBMS realice menos trabajo, es sólo cuestión de probar sentencias SQL alternativas (dependiendo de los datos) y de haber realizado un buen diseño de la base de datos, por lo tanto de las tablas e índices. Así será más fácil optimizar las aplicaciones, porque ya esta implícito en el diseño; si no hay un buen diseño, sera mucho más difícil y tardado hacer que un sistema funcione con un buen rendimiento. En general si se aprovechan las características del RDBMS en base a la información almacenada, y se desarrollan sentencias SQL que recuperen la información por el camino más corto, las aplicaciones reflejarán el mejor tiempo de respuesta que se puede obtener y no solo un tiempo de respuesta del que no estamos seguros si es el mejor.

Bibliografía

Groff, James R. **"Aplique, SQL."**, España 1991, Osborne/Mc Graw Hill. Páginas 619.

Newcomer, Larry R. **"SELECT... SQL. The relational database languaje."** U.S.A. 1992, Mac millan Publishing Companym, Páginas 446.

ORACLE **"SQL Language. Reference Manual Version 7.0"** U.S.A. 1992, ORACLE, Páginas 366.

Mahler, Paul. **"Power Builder Desarrollo de Aplicaciones Cliente Servidor"** Traductor: Joaquín Delgado Medina , España 1996, Prentice may ,Páginas 410

Corey Michael J., Abbey Michael, Dechichio Daniel J., Abramson Ian. **"Puesta a punto de Oracle8"** Traducción: Jorge Rodríguez Vega , España 1998, Osborne/Mc Graw Hill, Páginas 505

Loney Kevin. **"Oracle8. Manual del Administrador."** España 2000, Osborne/Mc Graw Hill, Páginas 708

Keesling Donna. Nathan Priya **"Oracle 8i: New Features for Developers"** U.S.A. 2000, Oracle Corporation, Páginas 346.

De Haan Lex. Greenberg Nancy **"Oracle 8i: SQL Statement Tuning Workshop"** U.S.A. 2000, Oracle Corporation, Páginas 308.

WEB

http://www.lobocom.es/~claudio/sql.html
http://kb.indiana.edu/data/ahux.html
http://www.jcc.com/SQLPages/jccs_sql.htm
http://www.computerbits.com/archive/19960500/sql1.htm
http://vision.ucsd.edu/~deborah/132b/db2/doc/html/db2s0/db2s003.htm#ToC
http://www.mcjones.org/System_R/
http://www.computer.org/annals/
http://www.redbooks.ibm.com/cgi-bin/bookmgr/BOOKS/QBKSQ900/CCONTENTS
http://www.recursos-as400.com/
http://lobos.itlp.edu.mx/publica/tutoriales/basedat1/
http://www.oracle.com/
http://www.oracle.com/es
http://www.microsoft.com
ftp://ftp.ora.com/pub/examples/oracle/tuning_2/
http://www.sybase.com/products/whitepapers/performance_tips.html
http://msdn.microsoft.com/library/techart/msdn_sql7perftune.htm
http://www.mcjones.org/System_R/SQL_reunion_95/index.html

Referencias Bibliografías

[AESB2000], **"Sistemas de Bases de Datos"**, Armes Elmasari, Shamkant B. Navathe, Addison, Wesley, 2000, México, Página 100.

Glosario.

ANSI. American National Standards Institute
Árbol. Conjunto formado por los elementos de información cuando, partiendo del principal (la raíz) pueden ser alcanzados todos los demás mediante una enumeración, dado que cada uno de ellos, como las personas en un árbol genealógico, deriva de otro o es el punto de partida de nuevas ramificaciones (hijos); en el caso contrario, o sea si no tiene descendencia, constituye una hoja. Un árbol binario, es aquel cuyos elementos tienen cero, uno o, como máximo, dos hijos; es calificado de árbol completo si todos sus elementos, salvo las hojas, tienen dos hijos.
Árbol-B. Permiten realizar búsquedas y actualizaciones de forma eficiente, a pesar de mantener el índice en un archivo. Es una forma de estructurar índices de gran tamaño que no caben en la memoria RAM. Un árbol B de orden ω cumple con las siguientes propiedades:

a. Toda página excepto la raíz contiene al menos ω claves.
b. La raíz contiene al menos una clave
c. La raíz tiene al menos 2 descendientes
d. Toda página contiene al menos 2 ω claves
e. Toda página excepto las hojas tiene m + 1 decendientes, donde m es el numero de claves en la página
f. Las hojas no tienen descendientes
g. Todas las hojas están en el mismo nivel

CODASYL. Conference on Data Systems Languages.
Commit. Aceptar la transacción.
Cursor. Conexión a una tabla o consulta de la base de datos, con la que se puede recorrer cada elemento de manera secuencial.
DBA. Administrador de la base de datos (Database Administrator)
DBTG. Database Task Group
FETCH. Es la acción de recuperar un registro de la base de datos.
Full Table Scan. Exploración completa de la tabla.
Hash. Las técnicas de HASH se basan en el proceso de extraer el índice de un elemento de un arreglo directamente a partir de la información que se va a guardar en el. Es ese índice que se genera el que se llama HASH. En el caso que el índice sea único se dice de indexación directa. Es en sí es una base de nodos llave y listas ligadas colgando de cada llave.
Hint: Sugerencia que indica al optimizador de Oracle de que manera debe acceder la información.
IEC. International Electrotechnical Commission
IEEE. Institute of Electrical and Electronics Engineers
Input/Output (I/O). Operaciones de Entrada/Salida, accesos al disco.
ISO. International Organization for Standarization
Join. Es aplicar la operación reunión entre dos relaciones, en términos de bases de datos es combinar dos tablas de información mediante un campo común.
Match. Cuando dos valores son iguales
ORACLE. Es uno de los sistemas manejadores de bases de datos relacionales comerciales más importantes del mercado.
Query. Sentencia SQL de selección o modificación de información.
Online. En línea, activo.
Rollback. Rechazar la transacción.
RowID. Identificador interno de los registros en una tabla.
SQL. Lenguaje estructurado de consulta, lenguaje estándar de las bases de datos relaciónales.
Trigger. Es un procedimiento almacenado en la base de datos que se activa al ocurrir sobre la tabla asociada algún evento de actualización, del tipo: insertar, actualizar o borrar datos de algún registro.
Tunning. Optimización, de la base de datos y/o de los querys de la aplicación.