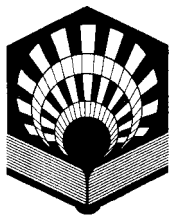

Universidad de Córdoba

DESARROLLO DE APLICACIONES

Manual de Optimización
Revisión 1.0

Servicio de Informática
Area de Sistemas
Julio 1993

1. Introducción.

En este Manual se ha recopilado información relativa a la optimización de aplicaciones desarrolladas con ORACLE, desde el punto de vista del diseñador y del programador.

Aunque la referencia fundamental es *Database Administrator's Guide Version 6.0*, incluimos ideas de otras fuentes (Manuales, Boletines Técnicos, publicaciones independientes,...) detalladas en el Apéndice A. Por otra parte, la experiencia adquirida en el desarrollo y explotación de aplicaciones ORACLE ha sido clave para la confección de este Manual.

Se supone un conocimiento medio de SQL. Referencias complementarias para el lector interesado en la implementación que ORACLE ofrece de dicho lenguaje pueden encontrarse en la documentación técnica correspondiente.

Hacemos notar que este Manual hace referencia a la versión 6.0 del gestor RDBMS de ORACLE. En la actualidad estamos evaluando la versión 7.0 de dicho gestor en el cual se han introducido numerosas modificaciones por lo que es de esperar una proxima revisión de este Manual.

2. Consideraciones generales.

1. Introducción.

Uno de los aspectos fundamentales a tener presente en buena parte de las etapas de diseño y desarrollo de un aplicación es el del rendimiento futuro de la misma. Desde el momento mismo en que se elige una configuración hardware y unas herramientas software para el desarrollo, los impactos sobre el rendimiento deben evaluarse cada vez que se tome una decisión de diseño y, en última instancia, durante toda la fase de codificación.

Una escasa planificación en este terreno nos puede proporcionar una aplicación poco eficiente lo que, con toda probabilidad, conducirá a tomar medidas *ad hoc* muy costosas y de dudosa efectividad.

En nuestro entorno se dispone de una herramienta, ORACLE, altamente parametrizable, lo que permite mejorar el rendimiento global de las aplicaciones. En menor medida, el sistema operativo UNIX permite al administrador adaptar su comportamiento a las necesidades específicas de un entorno de explotación concreto. Un desajuste en cualquiera de estos dos aspectos repercutirá, de inmediato, en el rendimiento de todas las aplicaciones.

La monitorización y ajuste del gestor ORACLE y del sistema operativo es un tema muy complejo y no es nuestra intención abordarlo en estas páginas dado su escaso interés general. Sin embargo, la optimización de aplicaciones, sobre todo en fases tempranas de desarrollo, es un tema de vital importancia para todos aquellos involucrados en la puesta a punto de una aplicación.

Este Manual intenta dar una visión amplia sobre los recursos disponibles para mejorar el rendimiento de las aplicaciones desde el punto de vista del diseñador y, sobre todo, del programador.

Algunas serán afirmaciones obvias fácilmente extrapolables a otros entornos. Sin embargo, la forma, a veces aparentemente caprichosa, de actuar que muestra el gestor ORACLE hace necesario que los desarrolladores dispongan de un conjunto claro de reglas que les permitan tomar decisiones fundadas.

Por último, y para animar al lector a evaluar en su justa medida la importancia del tema aquí tratado, diremos que ORACLE reconoce que en la optimización de un sistema construido con su gestor de bases de datos, aproximadamente el 70% de las mejoras de rendimiento se obtendrán actuando a nivel de aplicación.

Esperamos que estas páginas sean de utilidad a todos aquellos que estén embarcados en el desarrollo de una nueva aplicación o en el mantenimiento de una existente.

1. Diseño.

Las primeras apreciaciones sobre el rendimiento comienzan a discutirse en la fase de diseño. Es imprescindible conocer a fondo los datos antes de decidir la estructura física de la base de datos o de modificar una ya existente.

La flexibilidad del modelo relacional hace que las posibilidades de implementar una idea sean virtualmente ilimitadas. Considerando los previsibles problemas de rendimiento, este número de soluciones posibles se limita de forma apreciable. En general, siguiendo las reglas usuales de normalización asociadas a la teoría relacional, es posible obtener una base de datos con una estructura aceptable. Acogerse a una técnica probada de análisis ayuda a respetar unos criterios estrictos que usualmente conducen a un buen modelo de datos.

En particular, ORACLE recomienda el modelo Entidad/Relación. Una realización práctica de este modelo es el CASE*Method. ORACLE es categórica en sus afirmaciones respecto al cuidado necesario en la fase de análisis: una buena codificación no puede contrarrestar los efectos negativos de una base de datos pobremente diseñada.

En la fase final del análisis debe estudiarse la posibilidad de desnormalizar algunas tablas de modo que se evite el uso excesivo de *joins*. En particular debe prestarse atención a las tablas auxiliares que describen códigos: muchas veces los códigos no significan nada para el usuario final y puede ser más rentable almacenar la descripción completa para evitar el *join*, sobre todo si el rendimiento prima sobre consideraciones de espacio.

En cualquier caso, hay que cuidar el número de campos definidos en cada tabla. Las tablas cargadas de descripciones (los campos numéricos y tipo fecha no ocupan demasiado espacio) pueden

influir negativamente en el rendimiento de la aplicación. Aunque sólo se deseen recuperar algunos de los campos, la lectura de una de estas tablas implica recuperar muchos bloques de disco dado que éstos sólo pueden acomodar unas pocas filas de datos.

Otro aspecto fundamental al que a veces no se presta la suficiente atención es el de la separación de los entornos *online* y *batch*. En *ORACLE: Building High Performance Online Systems* (ver Apéndice A), el autor define el concepto de transacción no diseñada (*undesigned transaction*) como aquella que realiza más de 8 ó 10 operaciones de entrada/salida. Muestra, además, el efecto negativo que sobre el rendimiento de todo un sistema puede tener mezclar transacciones diseñadas y no diseñadas.

Una norma básica en cualquier sistema en explotación es definir un entorno *batch* con un horario de ejecución fuera de horas de oficina en el que se incluirán el mayor número posible de las transacciones no diseñadas que tengan que implementarse. El análisis debe identificar claramente tales transacciones para que el usuario final no pueda desencadenarlas libremente sino por solicitud previa.

2. Codificación.

Independientemente de la herramienta utilizada, al final, cualquier programa queda reducido a sentencias SQL. Dado que este lenguaje es muy flexible, es posible alcanzar los mismos resultados escribiendo la sentencia de distintas formas.

ORACLE incorpora un optimizador que decide el camino a seguir para recuperar o modificar los datos. Este optimizador está basado en reglas independientes de factores de explotación tales como el número de filas de una tabla, la cardinalidad de un índice o la distribución en disco. Esto hace que la forma de escribir una sentencia sea el único modo de obligar al optimizador a elegir un camino de búsqueda determinado. En consecuencia, gran parte de este Manual estará dedicado a evaluar el efecto de la sintaxis sobre el rendimiento de las aplicaciones.

Una norma básica es utilizar la herramienta adecuada en cada situación. ORACLE dispone de varias herramientas pensadas cada una para un tipo de trabajo. Una elección adecuada de la herramienta puede mejorar el rendimiento de la aplicación.

SQL*Loader es una herramienta diseñada para insertar datos masivamente en una tabla. Esto exige elaborar un fichero de control y disponer de los datos sobre un fichero secuencial. Este trabajo adicional puede llevar a pensar que una operación del tipo `INSERT-SELECT`, mucho más fácil de implementar, puede sustituir con ventaja a una carga con SQL*Loader.

Hemos probado la carga de mil registros utilizando SQL*Loader, por un lado, y el comando `INSERT`, por otro. Los resultados se muestran en la *Tabla I*.

Herramienta	Tiempo de cpu
SQL*Loader	0.43 s
INSERT	20.00 s

Tabla I: Carga masiva de datos.

Como se puede observar, la diferencia es muy apreciable.

La cualificación de los campos con el alias de cada tabla en los *joins*, facilita el trabajo de *parse* al gestor. De lo contrario, éste deberá acceder a la definición de todas las tablas que intervienen para determinar a cual de ellas pertenece cada campo. En este sentido, y para evitar además dependencias del código con respecto al diseño de la base de datos, deben evitarse también sentencias del tipo:

```
SELECT * FROM TAB
```

Por último, decir que es necesario que el programador se familiarice con las funciones que incorpora SQL (`SUBSTR`, `DECODE`,...). En general es preferible emplear dichas funciones a utilizar las propias del lenguaje huésped (`C`, `COBOL`,...).

Hemos comparado el consumo de cpu de dos procesos implementados en Pro*C que calculan la media de un campo numérico. El primero utiliza la función `AVG` proporcionada por SQL. El segundo utiliza directamente el lenguaje C para acumular y dividir por el número de datos. Los resultados se recogen en la *Tabla II*.

Método	Tiempo de cpu
Utilizando SQL	0.1 s
Utilizando C	594.9 s

Tabla II: Uso de funciones incorporadas.

Como vemos, la mejora de rendimiento puede llegar a ser de varios órdenes de magnitud.

En cualquier caso, hay que cuidar el uso de funciones en SQL: cuando una operación no dependa de los valores recuperados no debe incluirse en la sentencia. Por ejemplo, es frecuente implementar la funcionalidad de un `IF` con instrucciones `DECODE` dentro de una sentencia SQL, lo cual perjudica notablemente al rendimiento de la misma. En tales situaciones, PL/SQL permite extraer el `IF` de la sentencia y ejecutarlo una sola vez, en lugar de hacerlo por cada fila recuperada.

3. Indices.

1. Introducción.

Los índices son estructuras opcionales asociadas a las tablas que permiten incrementar notablemente la velocidad de ejecución de las sentencias. ORACLE emplea árboles de tipo B para balancear el tiempo de acceso a cualquier fila. Estos árboles constituyen una estructura independiente con sus propios parámetros de almacenamiento. Si sobre una columna se ha definido un índice, ORACLE accede a éste para obtener la dirección (ROWID) de los datos evitando el gran número de operaciones de E/S a disco que requiere una búsqueda secuencial en una tabla.

2. Impacto sobre las aplicaciones.

En principio, las aplicaciones son independientes de los índices: es posible crearlos y borrarlos sin necesidad de reescribir el código y es el gestor el encargado del mantenimiento de los mismos. No obstante, atendiendo a consideraciones de rendimiento, es imprescindible conocer los índices definidos en cada tabla para escribir el código que haga un uso más eficiente de los mismos.

La creación de un índice mejora sensiblemente la operaciones de recuperación de datos que puedan beneficiarse de la existencia del mismo. Sin embargo, las operaciones de actualización e inserción de datos se verán penalizadas por el hecho de tener que actualizar también el índice. Es por ello que se recomienda valorar cuidadosamente los posibles beneficios frente a los efectos negativos que implica la existencia del índice. En general, una tabla no debería tener más de dos o tres índices.

Sobre una tabla con unas 300.000 filas se ejecuta una sentencia de actualización de una columna que forma parte de sucesivos

índices. La actualización afecta a algo más de 17.000 filas y los resultados se muestran en la *Tabla III*.

Número de índices	Tiempo de cpu
Ninguno	1 m 19 s
1	2 m 15 s
2	5 m 59 s
3	7 m 53 s

Tabla III: Impacto del uso de índices.

Como reflejan los datos, cada índice que se añade a una tabla puede hacer que la actualizaciones de la misma requieran el doble de tiempo o más. Observamos que la actualización sobre la tabla con tres índices consume seis veces más tiempo que sobre la misma tabla en ausencia de índices.

3. Elección de índices.

Los índices pueden crearse sobre un sólo campo o sobre varios, en cuyo caso se habla de índices concatenados. Ambos pueden o no ser únicos. Un índice único hace que el gestor no permita duplicidades de campo (o de la combinación de campos en los concatenados). Sobre una misma tabla pueden crearse más de un índice.

La elección de los índices es una de las decisiones más comprometidas del diseño técnico. Una columna será una buena candidata a ser indexada si:

- No existe duplicidad de valores (el mejor caso). Por ejemplo, el D.N.I.
- El campo toma un amplio rango de valores. Por ejemplo el código de cliente en una tabla de pedidos. Una mala elección sería el campo sexo que sólo toma dos valores.
- El campo es usualmente nulo y con frecuencia se seleccionan aquellas filas que tienen un valor.

Si el índice es concatenado, independientemente del orden de los campos en la tabla, la primera columna del índice será el campo más usado o el más selectivo.

Las siguientes secciones explican en detalle las reglas dadas aquí anteriormente.

4. Uso eficiente de índices.

ORACLE puede acceder a las tablas de dos modos:

- Secuencialmente (FULL SCAN).
- Usando índices.

El diseñador crea los índices en función del uso más frecuente de los datos, pero es el programador el que decide, en último término, si en la sentencia en que está trabajando hará que el gestor RDBMS emplee los índices o haga un acceso secuencial.

En general, el acceso por índice es preferible. Sin embargo, si se preve que un alto porcentaje de los datos será consultado (digamos más de un 25%) un acceso secuencial puede ser más rápido.

Sobre una tabla de 4.253 filas hemos ejecutado una sentencia que recupera 3.265 (~75%), otra que recupera 988 (~25%) y, por último, otra que recupera 218 (~5%). Los resultados, empleando

índices y accediendo secuencialmente, se muestran en la *Tabla IV*.

Número de filas	Modo	Tiempo de cpu	Número de bloques
75 %	Secuencial	1.20 s	881
	Indexado	3.10 s	6.532
25 %	Secuencial	1.02 s	765
	Indexado	1.12 s	1.977
5 %	Secuencial	1.16 s	726
	Indexado	0.12 s	437

Tabla IV: Comparación de accesos sencuencial e indexado.

Los resultados confirman que el 25 % es una buena estimación del número de filas por encima del cual resulta más rentable hacer una búsqueda secuencial.

ORACLE puede leer varios bloques de datos en una sola operación. Si el acceso es secuencial, el gestor aprovecha esta situación para leer todas las filas recuperadas en una lectura sin accesos adicionales al disco.

Por el contrario, cuando el acceso es indexado, el gestor acude al índice y, con la dirección dada por éste, lee el correspondiente bloque de datos del disco. Aunque el siguiente dato a recuperar estuviese en el mismo bloque, dado que el acceso no es secuencial, ORACLE vuelve a repetir la lectura del mismo. La situacuón se agrava si se tiene en cuenta que cada vez que el gestor accede al disco recupera varios bloques de datos.

No obstante, esta descripción simplifica mucho el proceso. En realidad ORACLE utiliza una compleja gestión de memoria (siguiendo algoritmos del tipo LRU) que minimizan el acceso a disco.

Este comportamiento del gestor, lleva a ORACLE a recomendar que, para tablas pequeñas, no se utilicen índices, aunque ésto no parece corroborarlo la experiencia.

Hemos realizado pruebas sobre dos tablas pequeñas (menos de 100K) tomando tiempos de ejecución para la recuperación de una única fila utilizando el índice y buscándola secuencialmente. Los resultados se muestran en la *Tabla V*.

Tamaño de tabla	Modo	Tiempo de cpu	Número de bloques
244 filas (70K)	Secuencial	0.04 s	14
	Indexado	0.02 s	3
17 filas (10K)	Secuencial	0.00 s	1
	Indexado	0.01 s	2

Tabla V: Indices sobre tablas pequeñas.

Los datos de tiempo son poco significativos para poder apreciar la diferencia, por lo que se ha incluido el número de bloques (de 2K) que ha necesitado recuperar el gestor para resolver la consulta.

Se observa que no utilizar índices puede ser mas rápido para tablas realmente pequeñas, pero para dichas tablas la mejora de rendimiento puede ser insignificante. Creemos que esta mejora no compensa el esfuerzo extra de programación necesario para anular un índice cuando este deba ser creado (por ejemplo, para garantizar que no haya duplicidades).

5. Condiciones para el uso de índices.

En las discusiones que siguen haremos uso frecuente del concepto de predicado. Definimos un predicado como el criterio

de selección de una sentencia DML, esto es, las condiciones definidas por la cláusula WHERE. Por ejemplo, en la sentencia

```
SELECT  X,Y,Z
  FROM  TAB
  WHERE X > 1
  AND   Z LIKE 'C%'
```

el predicado sería:

```
  WHERE X> 1
  AND   Z LIKE 'C%'
```

El gestor RDBMS podrá usar un índice sólo si se cumplen estas dos condiciones:

C1: La columna es referenciada en un predicado.

C2: La columna indexada no está modificada por ninguna función u operador aritmético.

Que se cumplan ambas condiciones no implica necesariamente que el índice sea usado. El optimizador de ORACLE decidirá si es apropiado o no emplearlo.

Las siguientes secciones están dedicadas a profundizar en las condiciones antes indicadas y a dar algunas de las reglas que emplea el optimizador (no todas están bajo el control del programador).

5.1 Columnas referenciadas explícitamente en predicados .

A veces puede interesar suprimir el uso de algún índice, por ejemplo, para asegurarnos que el optimizador elige otro que consideremos más adecuado. Sin embargo, esto no es frecuente y, lamentablemente, resulta habitual escribir sentencias que suprimen el uso de índices simplemente por error. Conviene, por tanto, matizar el significado de las condiciones dadas en la sección anterior.

La condición C1 implica que una sentencia del tipo:

```
SELECT X, Y FROM TAB
```

no empleará índices por carecer de predicado y hará un acceso secuencial (lo cual es generalmente preferible dado que se accede a toda la tabla). El que aparezca la columna X como una de las seleccionadas, no hace que se emplee un hipotético índice sobre dicha columna dado que no aparece en ningún predicado.

Supuesto un índice sobre la columna X, las sentencias:

```
SELECT  X,Y
  FROM  TAB
  WHERE X = 1
```

y

```
SELECT  X,Y
  FROM  TAB
  WHERE X > 0
```

sí emplearían el índice sobre la columna X.

En el caso de índices concatenados, la condición C1 se aplica a la primera columna del índice. Así, suponiendo un índice concatenado sobre la columnas (X,Y,Z), en este orden, el predicado:

```
WHERE   X = 1
  AND   Y = 'A'
  AND   Z = 'C'
```

empleará el índice eficientemente. Sin embargo, los predicados:

```
WHERE   X = 1
  AND   Y = 'A'
```

y

```
WHERE   X = 1
  AND   Z = 'C'
```

sólo emplearán el índice parcialmente ya que el predicado no es suficientemente selectivo y habrá que hacer lecturas secuenciales (RANGE SCAN).

Lo más importante a destacar es que un predicado como:

```
 WHERE  Y = 'A'
  AND   Z = 'C'
```

no podrá usar el índice dado que la primera columna del mismo (X) no aparece en la cláusula WHERE. En esta última situación es posible obligar al optimizador a emplear el índice. Suponiendo que el campo X toma siempre valores positivos, sería suficiente con reescribir la cláusula del siguiente modo:

```
WHERE  Y = 'A'
 AND   Z = 'C'
 AND   X > 0
```

Es importante resaltar que el orden de los campos en un predicado no es significativo para la utilización o no de un índice.

Conociendo esta forma de actuar del optimizador de ORACLE, se deducen dos reglas para la creación de índices concatenados:

- Ordenar las columnas por su selectividad, eligiendo como primera columna la más selectiva para limitar el impacto de los usos parciales del índice (RANGE SCAN)
- Elegir como primera columna la más frecuente en los predicados.

Un compromiso entre ambas reglas determinará la creación del índice. La primera suele ser prioritaria dado que ya hemos visto que reescribiendo la sentencias adecuadamente es posible, en general, suprimir los efectos de la segunda regla.

Mencionar, por último, que una característica importante de los índices concatenados es que permiten satisfacer consultas con sólo acceder al índice. En efecto, a veces, todos los campos requeridos pertenecen al índice concatenado y la consulta es resuelta sin acceder a la tabla.

5.2 Columnas modificadas por funciones u operadores.

La segunda condición hace que, en el supuesto de que un índice existiese sobre la columna X y otro sobre la Y, los siguientes predicados:

```
WHERE   X + 1 = 3

WHERE   SUBSTR (Y,1,1) = 'A'

WHERE   Y||Z = 'ACDR'
```

no permitirían el uso de los correspondientes índices, dado que las columnas han sido modificadas por funciones u operadores.

Así, una misma sentencia puede escribirse para que use o no el índice según convenga. Siguiendo el ejemplo anterior, en las consultas:

```
WHERE   X = 2

WHERE   Z = 'A'
```

es posible anular el empleode índices reescribiendo las sentencias del siguiente modo:

```
WHERE   X+0 = 2

WHERE   Z||'' = 'A'
```

Una situación frecuente se produce en la conversión de fechas. Supuesto que FECHA es un campo tipo DATE, el predicado:

```
WHERE   FECHA = TO_DATE ('14-ENE-80')
```

empleará un índice que se defina sobre el campo FECHA, mientras que el predicado:

```
WHERE   TO_CHAR (FECHA) = '14-ENE-80'
```

no podrá usar dicho índice. El siguiente predicado también emplearía índices:

```
WHERE   FECHA = '14-ENE-80'
```

5.3 Excepciones.

Una excepción importante a estas reglas es la optimización especial empleada con las funciones MAX y MIN, que devuelven un sólo valor y, por ello, siempre emplean índices.Así, una sentencia select que contenga la expresión:

```
{ MAX | MIN } ( col {+|-} constante )
```

y ninguna otra columna, como en

```
SELECT 2 * MAX (X+1) FROM TAB
```

empleará el índice sobre la columna X aunque no cumpla ninguna de las condiciones necesarias.

Recordemos que esta consulta puede satisfacerse accediendo sólo al índice, sin necesidad de recuperar los datos de la tabla.

6. Indices y valores nulos.

Siempre que sea posible, es preferible definir las columnas como NOT NULL (incluso buscando un valor como cero o espacio para substituir el significado del nulo). Esto permite el uso de índices en un mayor número de sentencias.

En efecto, ORACLE no empleará índices si el predicado contiene las cláusulas:

```
IS NULL
IS NOT NULL
```

Esto se debe a que el gestor no almacena los valores nulos en el índice. En general, como hemos comentado con anterioridad, un índice no contendrá columnas que admitan valores nulos. Cuando la búsqueda se realiza sobre uno de estos índices, aunque el predicado sea de igualdad e identifique completamente la fila deseada, el gestor accede parcialmente en modo secuencial (RANGE SCAN).

Aún cuando se haya definido un índice sobre una columna que admite nulos, es posible, en algunos casos, obligar al optimizador a usar parcialmente los índices. Por ejemplo, suponiendo X una columna indexada y con valores nulos, si estamos interesados en aquellas filas que tienen un valor no nulo de X, la sentencia:

```
SELECT  X,Y, Z
  FROM  TAB
  WHERE X IS NOT NULL
```

no empleará el índice, mientras que lasentencia:

```
SELECT  X,Y,Z
  FROM  TAB
  WHERE X > 0
```

sí empleará el índice y el resultado será el mismo, asumiendo que X tome siempre valores positivos. Si, por el contrario, estamos interesados en las filas cuyo valor sea nulo, no hay forma de evitar la lectura secuencial.

7. Tablas con múltiples índices.

Si se ha definido más de un índice en una tabla, el gestor podrá usarlos simultáneamente si:

- Los índices noson únicos.
- Los predicados son de igualdad.

La primera condición es obvia: si uno de los índices es único (devuelve una única fila) no es necesario recurrir a índices adicionales.

Por ejemplo, suponiendo un índice único sobre X y uno no único sobre Y, en la sentencia:

```
SELECT  X,Y,Z
  FROM  TAB
  WHERE X = 1
  AND   Y = 'A'
```

ORACLE usará el índice sobre X para recuperar la fila correspondiente a X=1 y, a continuación, comprobará si Y='A' en función de lo cual entregará o no la fila al programa que hizo la consulta.

La segunda condición es una limitación del optimizadorque debe tenerse en cuenta cuando se trate de evaluar cuales serán los índices empleados.

Si el optimizador decide usar más de un índice, selecciona las filas según cada criterio y luego combina (*merge*) los resultados en un sólo conjunto de datos eliminando duplicidades.

En cualquier caso, ORACLE no usará más de cinco índices. Si en una sentencia existiesen predicados que involucrasen a más de cinco índices, sería interesante suprimir el uso de los menos efectivos con las técnicas descritas anteriormente.

8. Optimizando joins

El empleo de índices en los *joins* resulta crítico para el rendimiento global de toda la aplicación. Siempre que se necesite utilizar un *join*, el programadordebe cuidar la forma de codificarlo de modo que se aprovechen eficientemente los índices.

La regla fundamental consiste sencillamente en asegurar que las columnas claves del *join*, aquellas que relacionan los datos de cada par de tablas, estén indexadas. Además, debe escribirse la sentencia de forma que las funciones y operadores no anulen el uso de índices.

Para realizar la consulta generada por un *join*, el gestor elige una de las tablas como conductora (*driver*) del *join*. ORACLE lee cada fila de esta tabla que cumpla las condiciones dadas y, para cada una de estas filas busca en las demás tablas las filas que cumplan las condiciones adicionales especificadas en el *join*.

El acceso a la tabla conductora es siempre secuencial. Por el contrario, el acceso al resto de tablas que participan en el *join* puede realizarse por medio de índices, siempre que esto sea posible.

Así pues, el objetivo del programador será conseguir que la tabla conductora sea aquella que presente, a priori, un camino de acceso más lento (sin índices, índices no únicos o con menor número de filas).

El optimizador de ORACLE selecciona la tabla conductora en función de los índices definidos y de sus propias reglas de optimización. En las siguientes secciones detallaremos estas reglas.

8.1 Joins no indexados.

Los *joins* implican siempre una condición de igualdad (*equijoins*) sobre uno o varios campos de las tablas que participan en el mismo. A dichos campos, que relacionan las filas recuperadas de las diferentes tablas, le llamaremos campos clave del *join*.

Siempre en un *join* el optimizador intentará utilizar los índices definidos sobre los campos clave del mismo. Si no es posible usar índices (por ejemplo, porque no existan), ORACLE debe realizar una ordenación de todas las tablas y una combinación posterior de las mismas (*sort-merge join*).

Concretamente, el gestor seleccionará los datos de cada tabla aplicando las condiciones dadas para los campos que no son claves del *join*. Los subconjuntos obtenidos a partir de cada tabla, son ordenados por separado. Los resultados se combinan en una nueva relación basándose en las condiciones clave del *join*.

Las condiciones adicionalesmás selectivas (aquellas que no son claves) deben situarse al final del predicado. Esto disminuirá el número de comprobaciones durante la primera fase de la operación anteriormente descrita.

En un *join* no indexado,por tanto, ninguna de las tablas es la conductora.

8.2 Joins indexados.

Este tipo de consulta obliga al optimizador a realizar un complejo trabajo para decidir la tabla conductora.

En el *join* de dos tablas, si sólo existen índices usables para una de ellas, el optimizador elegirá laotra tabla para conducirlo. De esta forma, realizará un barrido secuencial sobre la tabla no indexada y, por cada fila, accederá por índice a la otra. La situación inversa obligaría a una lectura secuencial de toda la tabla no indexada por cada fila leida en la tabla indexada.

Por ejemplo, si la columna `X` está indexada en la tabla `TAB_1` pero no lo está en la tabla `TAB_2`, en ambas consultas:

```
SELECT  T1.X,                  SELECT   T1.X,
        T2.Y                            T2.Y,
  FROM  TAB_1 T1,                 FROM  TAB_1 T1,
        TAB_2 T2                        TAB_2 T2
  WHERE T1.X = T2.X               WHERE T2.X = T1.X
```

la tabla conductora será `TAB_2`. Una excepción a esta regla se da cuando uno de los predicados sobre la tabla no indexada garantiza que se va a recuperar una sola fila (por ejemplo, buscando por `ROWID`).

Si hay más de un índice usable, tanto en las columnas claves del *join* como en el resto de predicados, `ORACLE` ordena todos los caminos de acceso que encuentre según los criterios mostrados en la *Tabla VI*, eligiendo como tabla conductora la que tenga mayor puntuación (es decir, el camino de acceso más lento).

Puntuación	Camino
1	`ROWID` = constante
2	Columna con índice único = constante
3	Indice único concatenado entero = constante
4	Indice *cluster* entero = índice correspondiente a otra tabla dentro del mismo cluster.
5	Indice *cluster* entero = constante
6	Indice no único concatenado entero = constante
7	Indice no único = constante
8	Indice concatenado entero = rango inferior cerrado
9	Indice concatenado especificado parcialmente
10	Columna con índice único en una cláusula `BETWEEN` o con la condición `LIKE 'A%'` (rangos cerrados)
11	Columna con índice no único en una cláusula `BETWEEN` o con la condición `LIKE 'A%'` (rangos cerrados)
12	Columna con índice único o constante (rangos abiertos)
13	Columna con índice no único o constante (rangos abiertos)
14	*Joins* no indexados (*sort-merge*)
15	`MAX` o `MIN` sobre una sóla columna indexada
16	`ORDER BY` sobre columnas que pertenecen a un índice
17	Acceso secuencial a una tabla (`FULL SCAN`)
18	Columna no indexada = constante o columna `IS NULL` o columna `LIKE '%A%'`

Tabla VI: Reglas del optimizador.

Si `ORACLE` no encuentra una clara opción para seleccionar la tabla conductora, elegirá la última especificada en la cláusula `FROM`. Esto es importante tenerlo en cuenta si se preve un posible empate en cuanto a la velocidad de acceso. En esta situación será

rentable situar la tabla con mayor número de filas al final de la cláusula FROM.

Hacemos notar, por último, que la *Tabla VI* sirve también como referencia al programador sobre las reglas internas que emplea el optimizador ORACLE. Conociendo estas reglas, es posible codificar las sentencias de modo que el acceso sea lo más rápido posible no solamente en los *joins* sino en cualquier sentencia SQL.

No obstante, la *Tabla VI* debe tomarse como una ayuda a la hora de proceder a codificar una sentencia SQL. No es necesario comprender a fondo todos los mecanismos de optimización de ORACLE y, por tanto, se concederá a dicha tabla una importancia relativa. Además, debe tenerse en cuenta, que las reglas de optimización tienen un caracter heurístico y, por ello, podrán variar en futuras versiones del gestor.

4. Optimización de sentencias.

1. Introducción.

En este capítulo recogemos información sobre la optimización de distintas sentencias SQL así como de otros aspectos no contemplandos en capítulos anteriores.

2. Optimizando el uso del operador NOT.

Siempre que aparece la negación de una igualdad (`!=` o `NOT=`), `ORACLE` no emplea índices. Así, aunque un índice se haya definido para la columna `X`, la sentencia:

```
SELECT  X,Y,Z
  FROM  TAB
  WHERE X != 0
```

no empleará dicho índice. Este comportamiento se justifica porque es de suponer que la consulta anterior recupere la mayor parte de las filas de la tabla; en estos casos, suele ser más rápida una búsqueda secuencial.

En la situación anterior, si además suponemos que se ha definido un índice sobre la columna `Y`, la sentencia:

```
SELECT  X,Y,Z
  FROM  TAB
  WHERE X != 0
  AND   Y = 'A'
```

podría emplear el índice sobre la columna `Y` a pesar de la negación sobre la columna `X`.

3. Optimizando el uso del operador OR.

Las sentencias que incluyen el operador `OR` pueden usar índices si estos se han definido sobre todas lascolumnas que intervienen.

Si alguna de las columnas no está indexada, es posible que el optimizador decida no emplear índices. Esta última circunstancia queda fuera del control del programador (suele darse en sentencias que incluyen un `CONECT BY` y en *joins* externos).

Cuando todas las columnas están indexadas, `ORACLE` puede usar los índices eficientemente. Por ejemplo, si se han definido índices sobre las columnas `X` e `Y`, la sentencia:

```
SELECT  X,Y,Z
  FROM  TAB
  WHERE X = 1
  OR    Y = 'A'
```

es transformada en algo similar a la unión de las dos consultas siguientes:

```
SELECT  X,Y,Z
  FROM  TAB
  WHERE X = 1
```

```
SELECT  X,Y,Z
  FROM  TAB
  WHERE Y = 'A'
  AND   X != 1
```

Conociendo este comportamiento, es mejor situar al principio la condición mas selectiva.

Sobre una tabla de una 300.000 filas hemos realizado una consulta secuencial cuyo predicado incluye dos condiciones enlazadas por un `OR`. Una condición la cumple solamente el 0.26% de las filas (es muy selectiva). La otra aproximadamente el 38%. La diferencias de tiempos que resultan de cambiar el orden de las condiciones se muestran en la *Tabla VII*.

Condición más selectiva	Tiempo de cpu
al principio	7 s
al final	52 s

Tabla VII: Optimización de `OR`s.

Se observa una mejora sustancial en el rendimiento. La ganancia puede ser mas notable aún si las diferencias de selectividad de las condiciones son más pronunciadas.

La optimización sobre los OR's se aplica, en igual medida, a la cláusula IN, dado que el predicado:

```
WHERE Y IN (A,B,C)
```

es equivalente a:

```
WHERE   Y = 'A'
  OR    Y = 'B'
  OR    Y = 'C'
```

*4. Optimizando el uso del operador **AND**.*

Cuando en un predicado intervienen varias condiciones unidas por el operador AND, puede obtenerse un ligeroaumento de rendimiento situando la condición más selectiva al principio.

Esto se debe al orden en que el *parser* de ORACLE analiza las sentencias: las condiciones unidas por el operador AND dejan de evaluarse cuando cualquiera de ellas no es cierta.

Bajo las mismas condiciones de las pruebas comentadas para el operador OR, hemos realizado dos consultas situando la condición más selectiva al principio y al final. Las diferencias de tiempo de

cpu no son apreciables por lo que se incluyen los bloques leidos en la *Tabla VIII*.

Condición más selectiva	Tiempo de cpu	Número de bloques
al principio	0.01 s	7
al final	0.03 s	9

Tabla VIII: Optimización de ANDs.

Como se pueder ver, las diferencias son mínimas aunque tampoco supone un esfuerzo excesivo cuidar la situación de las condiciones cuando estas van unidas por el operador `AND`.

5. Ordenación.

El gestor de `ORACLE` dispone de unas rutinas de ordenación y combinación de datos (`SORT/MERGE`) altamente especializadas que optimizan el rendimiento de aquellas operaciones donde se requiere una ordenación previa: `CREATE INDEX`, `SELECT DISTINCT`, `ORDER BY`, `GROUP BY` y ciertos tipos de *joins*.

De forma simplificada podemos decir que el gestor ordena los datos en pequeños grupos (*runs*) los cuales combina (*merge*) para obtener un resultado final. Existen varios parámetros que permiten al DBA influir sobre las rutinas deordenación de `ORACLE` de forma que un alto porcentaje (más del 98%) de las ordenaciones se realicen en memoria.

Es importante conocer que la única forma de garantizar el orden de recuperación de un conjunto de datos es con la cláusula `ORDER BY`. Según la definición de índices asociada a las tablas consultadas puede obtenerse un orden adecuado sin especificar la cláusula `ORDER BY`; sin embargo, `ORACLE` no garantiza que en futuras versiones se mantenga este orden.

Cuando se realiza una consulta que contiene un GROUP BY, el gestor debe ordenar los datos antes de agruparlos, por lo que es mucho más eficiente imponer todas los criterios de selección como un predicado WHERE (antes de la ordenación) que con la cláusula HAVING. Esta última debe reservarse para cuando las condiciones se desean imponer sobre criterios de grupo. Por ejemplo, la consulta:

```
SELECT  Y, AVG(X)
  FROM  TAB
  WHERE Y LIKE 'A%'
  GROUP BY Y
```

es preferible a:

```
SELECT  Y, AVG(X)
  FROM  TAB
  GROUP BY Y
  HAVING Y LIKE 'A%'
```

6. *Creación y llenado de tablas.*

En la ejecución de tareas *batch*, es frecuente utilizar tablas temporales de trabajo. Realizar un borrado completo (DELETE) de las mismas es una operación muy costosa que, además, tiene efectos colaterales muy perjudiciales (fragmentación). Por tanto, y hasta que la sentencia TRUNCATE no esté disponible, lo normal suele ser borrar la tabla (DROP) y crearla de nuevo.

En este proceso se plantean dos alternativas:

- Crear la tabla, llenarla y crear los índices.
- Crear la tabla, crear los índices y llenarla.

En principio, ORACLE recomienda llenar las tablas sin índices y crear estos posteriormente. Una vez creada la tabla, parece menos costoso ordenar todas sus filas para crear el índice correspondiente. Además se evita el procesamiento que conlleva la búsqueda del lugar adecuado a cada nueva entrada del índice (recordemos que son estructuras en arbol de tipo B y que el gestor debe balancear las entradas para igualar la profundidad de todas ellas).

No obstante, un buen dimensionamiento del índice antes de crearlo puede ahorrar mucho trabajo al gestor si se opta por crearlo antes de llenar la tabla.

Hemos probado a llenar una tabla con mil filas creando el índice previamente y dejando la creación del mismo para después de insertar las filas. Los resultados se muestran en la *Tabla IX*.

Creación del índice	Tiempo de cpu	Número de bloques
Después de insertar	21.24 s	7.069
Antes de insertar	20.56 s	11.145

Tabla IX: Creación y llenado de tablas.

Las diferencias en tiempo de cpu (utilizado para llenar la tabla y crear el índice) no son apreciables. Sin embargo, se observa que creando el índice despúes de llenar la tabla el gestor necesita procesar muchos menos bloques. Es de esperar que para tablas mayores, las diferecias de cpu adquieran importancia y se decanten a favor de insertar las filas antes de crear el índice.

7. Subqueries vs. joins.

ORACLE afirma que su gestor de bases de datos está especialmente diseñado para resolver eficientemente los *joins*. Por ello recomienda usarlos en lugar de otras posibilidades que ofrece SQL.

En concreto, es posible utilizar *queries* anidadas o *subqueries* como alternativa al *join*. Una *subquery* crea una tabla temporal para almacenar las filas recuperadas de la tabla *hija* y, posteriormente, realiza un *join* con la tabla *padre*. Por tanto, además de no eliminar la operación de *join*, una *subquery* se apoya en tablas temporales que carecen de índices y sobre cuya parametrización (dimensionamiento) no se puede influir directamente.

Esta discusión, hace pensar que un *join* es mucho más eficiente que una *subquery*, pero la experiencia no parece corroborarlo.

Como prueba hemos recuperado 1000 filas a partir de dos tablas, ambas indexadas, una con 300.000 filas y otra con sólo 100. En la prueba se ha buscado que la tabla conductora del *join* sea la mayor, al igual que en la *subquery* el `SELECT` *padre* se realiza sobre dicha tabla. Los resultados se muestran en la *Tabla X*.

Operación	Tiempo de cpu
Join	1.12 s
Subquery	1.23 s

Tabla X: Comparación entre *join* y *subquery*.

Aunque las diferencias de tiempo no son significativas, no hemos incluido datos sobre el número de bloques utilizados por que dicho número coincide en ambos métodos. Por tanto, podemos concluir que no existe una clara ventaja de un método de acceso sobre el otro. No obstante, y dado que los argumentos teóricos en favor del *join* están justificados, es preferible emplear éste siempre que sea posible. Quizás, en otro tipo de prueba, las diferencias sean más relevantes.

5. Herramientas.

8. Introducción.

Existen diversas herramientas que ayudan a localizar problemas de rendimiento en las aplicaciones. Están pensadas fundamentalmente para ser utilizadas por el DBA y no vamos a describirlas en este Manual. No obstante, es importante que los equipos de desarrollo sepan de su existencia para que así puedan pedir la colaboración del Area de Sistemas a la hora de diagnosticar un problema de rendimiento.

Aparte de la herramientas propias del DBA (`MONITOR`, `TKPROF`,...) existen una utilidad que puede ayudar a los programadores a decidir entre diferentes alternativas. Se trata de la sentencia `EXPLAIN PLAN` que nos muestra las decisiones que tomará el optimizador cuando ejecute una sentencia determinada.

Este capítulo explica en detalle la utilización de esta utilidad.

9. La sentencia ***EXPLAIN PLAN***.

La sentencia `EXPLAIN PLAN` muestra el plan de ejecución elegido por el gestor `ORACLE` para optimización de sentencias DML. Un plan de ejecución es un conjunto de operaciones elementales y un orden en el que el gestor ejecuta dichas operaciones.

Antes de poder utilizar la sentencia `EXPLAIN PLAN` es necesario crear una tabla de trabajo llamada `PLAN_TABLE`. Para ello se ejecuta el procedimiento:

```
SQL> @?/rdbms/admin/xplainpl
```

Una vez creada la tabla de trabajo es posible utilizar la sentencia `EXPLAIN PLAN`, la cual almacena el plan de ejecución resultante en esta tabla de trabajo. La sintaxis de dicha sentencia es:

```
EXPLAIN PLAN [SET STAMENT_ID = 'descripción']
  [INTO tabla]
  FOR sentencia_sql
```

en donde:

`descripción` es una identificación opcional de hasta 30 caracteres que permite diferenciar los distintos planes de ejecución almacenados en la tabla de trabajo. Usualmente, la tabla de trabajo no contendrá más de uno o dos planes de ejecución.

`tabla` es el nombre de la tabla de trabajo. Si no se especifica, se utiliza la tabla `PLAN_TABLE`.

10. Descripción de la tabla de trabajo.

La tabla de trabajo, `PLAN_TABLE`, en la cual la sentencia `EXPLAIN PLAN` almacen el plan de ejecución, tiene los siguientes campos:

STATEMENT_ID	Identificador opcional del plan de ejecución.
TIMESTAMP	Fecha del análisis
REMARKS	Comentarios adicionales (hasta 80 caracteres).
OPERATION	Nombre de la operación elemental.
OPTIONS	Descripción adicional sobre la operación a realizar.
OBJECT_NODE	*Database link*. No suele interesar.
OBJECT_OWNER	Propietario de la tabla o índice utilizado en la operación.
OBJECT_NAME	Nombre de la tabla o índice utilizado en la operación.
OBJECT_INSTANCE	Un número indicando la posición del objeto en la sentencia: numerados de izquierda a derecha. No suele interesar.
OBJECT_TYPE	Un modificador que da más información sobre el objeto. Por ejemplo `NON-UNIQUE` para índices.
SEARCH_COLUMNS	No usado actualmente.

ID	Un número asignado a cada operación elemental del plan de ejecución.
PARENT_ID	El ID de la siguiente operacion elemental. Más adelante se verá la relación entre ambos campos.
POSITION	Orden de procesamiento para aquella operaciones que tienen el mismo PARENT_ID.
OTHER	Información adicional (por ejemplo, proceso distribuido) que que no suele ser de interés general.

*11. Utilización de la sentencia **EXPLAIN PLAN**.*

Para mostrar el uso de la sentencia EXPLAIN PLAN, empleamos las tablas DEPT y EMP que tradicionalmente usa ORACLE en todos sus ejemplos. Dada la sentencia:

```
SELECT  E.ENAME, E.JOB, E.SAL, D.DNAME
  FROM  EMP E,
        DEPT D
WHERE   E.DEPNO = D.DEPNO
  AND   NOT EXISTS
           (SELECT  'TRUE'
           FROM     SALGRADE
           WHERE    E.SAL BETWEEN LOSAL AND HISAL)
```

podemos obtener el plan de ejecución con la sentencia:

```
EXPLAIN PLAN
SET STATEMENT_ID = 'Prueba 1'
FOR
SELECT  E.ENAME, E.JOB, E.SAL, D.DNAME
  FROM  EMP E,
        DEPT D
WHERE   E.DEPNO = D.DEPNO
  AND   NOT EXISTS
           (SELECT  'TRUE'
              FROM   SALGRADE
              WHERE  E.SAL BETWEEN LOSAL AND HISAL)
```

El plan de ejecución ha sido almacenado en la tabla de trabajo. Una forma de obtener un listado identado correctamente es con la sentencia:

```
SELECT  LPAD(‘ ’,2*LEVEL)||OPERATION||‘ ’||
        OPTIONS||‘ ’||OBJECT_NAME PLAN_DE_EJECUCION
  FROM  PLAN_TABLE
  WHERE STATEMENT_ID = ‘Prueba 1’
  CONNECT BY PRIOR ID = PARENT_ID
        AND STATEMENT_ID = ‘Prueba 1’
        START WHIT ID = 1
```

que muestra el siguiente resultado:

```
PLAN DE EJECUCION

  FILTER
    MERGE JOIN
      SORT JOIN
        TABLE ACCESS FULL DEPT
      SORT JOIN
       TABLE ACCESS FULL EMP
    TABLE ACCESS FULL SALGRADE
```

Como se ve, el optimizador ha decidido realizar un *merge-join* de las tablas EMP y DEPT, dado que no hay índices definidos sobre ninguna de las tablas empleadas en la sentencia. El filtro del nivel superior, provoca una búsqueda secuencial en la table SALGRADE por cada fila entregada por la operación previa (*merge-join*).

La *Tabla XI* describe todas las operaciones elementales que puede mostrar un plan de ejecución así como sus diferentes opciones. La interpretación completa de un plan es compleja y no creemos necesario profundizar en ello. El Area de Sistemas podrá ayudar en la interpretación de dichos planes cuando éstos sean excesivamente complicados.

Operación	Opciones	Descripción
AND-EQUAL		Recuperación utilizando la intersección de ROWIDs obtenidos por búsqueda en índices. Esta operación se utiliza en predicados de igualdad que unen condiciones con el operador AND.
CONNECT BY		Se utiliza para implementar un CONECT BY.
CONCATENATION		Recuperación a partir de un grupo de tablas. Es una operación de UNION ALL de las tablas originales.
COUNTING		Cuenta el número de filas recuperadas de una tabla.
FILTER		Aplica una restricción sobre las filas a recuperar.
FIRST ROW		Recupera sólo la primera fila del resultado de una consulta.
FOR UPDATE		Genera un *row lock* sobre las filas seleccionadas.
INDEX	UNIQUE SCAN	Búsqueda por índice para localizar un único valor.
	RANGE SCAN	Búsqueda por índice para localizar un rango de valores.
INTERSECTION		Recupera las filas comunes de dos tablas. Previamente éstas son ordenadas.
MERGE JOIN		Un *join* obtenido por la combinación (*merge*) de dos conjuntos ordenados de tablas.
	OUTER	El *join* es externo.
MINUS		Recuperación de filas que están en una tabla pero no en otra.
NESTED LOOPS		Un *join* realizado sobre dos operaciones hijas. Para cada fila recuperada por la primera operación, se realiza la segunda operación.
	OUTER	El *join* es externo.

Tabla XI: Operaciones presentes en un Plan de Ejecución.

Operación	Opciones	Descripción
PROJECTION		Recuperación de un subconjunto de columnas de una tabla.
REMOTE		Recuperación sobre una Base de Datos diferente a la actual.
SEQUENCE		Una operación que involucra a un generador de secuencias.
SORT	UNIQUE	Recuperación ordenada de filas para producir valores únicos.
	GROUP BY	Recuperación ordenada de filas destinadas a realizar agrupaciones.
	JOIN	Recuperación ordenada de filas para la realización de un *join*.
	ORDER BY	Recuperación ordenada de filas provocada por la cláusula ORDER BY.
TABLE ACCESS	BY ROWID	Acceso a una tabla por ROWID, normalmente obtenido de la lectura previa de un índice.
	FULL	Acceso secuencial a una tabla (FULL SCAN).
	CLUSTER	Acceso a una tabla en *cluster*.
UNION		Recuperación de filas de dos tablas eliminando duplicidades.
VIEW		Recuperación de filas utilizando una vista.

Tabla XI: Operaciones presentes en un Plan de Ejecución.

A. Referencias.

W.H. Inmon. ORACLE*: Building High Performance Online Systems*. QED Information Sciencies, Inc. 1989.

U. Rodgers. ORACLE*: A Database developer's guide*. Yourdon Press. 1991.

S. Dimmick et al ORACLE RDBMS. *Database Administratior's Guide. Version 6.0*. 1990.

B. Linden et al. ORACLE RDBMS. *Performance Tuning Guide. Version 6.0.* 1990.

VV.AA. ORACLE UK. *Technical Bulletin*. March 1991.

E. Armstring et al. ORACLE7 *Server. Application Developer's Guide.* 1992