Administración de Bases de Datos

Tema 3

Procesamiento y optimización de consultas

Objetivos

- Conocer los mecanismos implementados en Oracle para optimizar las consultas.
- Interpretar un plan de ejecución de consulta.
- Conocer cómo puede cambiarse el comportamiento del optimizador de consultas.

EL OPTIMIZADOR

- El optimizador es una aplicación que determina la forma más eficiente de ejecutar una sentencia SQL.
- La base de datos puede ejecutar una sentencia de SQL de múltiples formas. El optimizador considera muchos factores relacionados con los objetos y las condiciones de la consulta para determinar un plan de ejecución.
- Es un paso importante en el procesamiento de SQL y puede afectar en gran medida al tiempo de ejecución

- Cuando el usuario envía una sentencia SQL para su ejecución, el optimizador realiza los siguientes pasos:

  • Genera un conjunto de posibles planes, para la sentencia SQL, basándose en los posibles caminos de ejecución y sugerencias disponibles.

  • El optimizador estima el coste de cada plan en base a las estadísticas almacenadas en el diccionario de datos. Las estadísticas incluyen información sobre las características de distribución y almacenamiento de los datos en las tablas, índices y particiones accedidas por la sentencia.

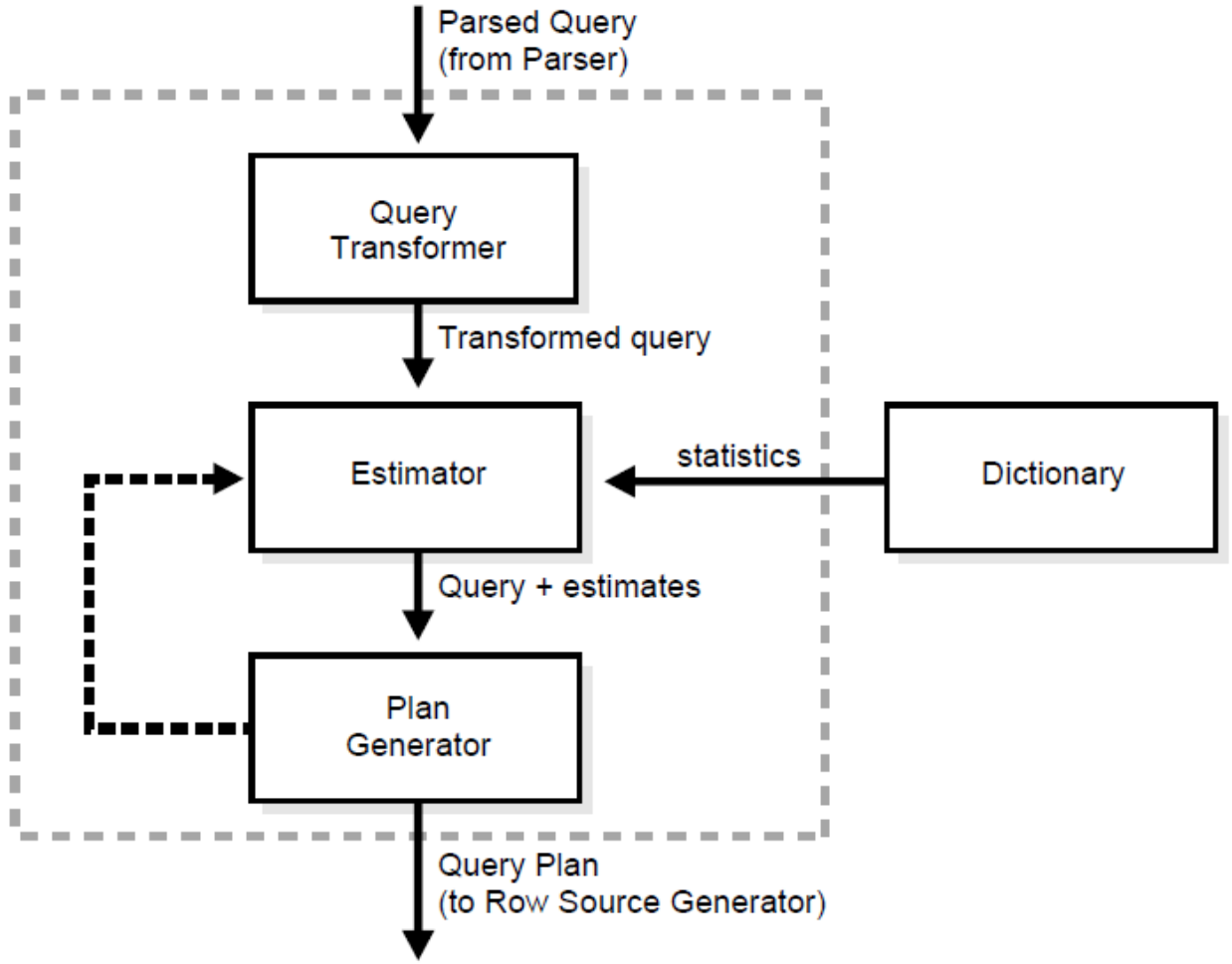
  • El optimizador compara los planes y elige el plan con el coste más bajo.

- La salida del optimizador es un plan de ejecución que describe el mejor método de ejecución. El plan contiene la combinación de pasos que usa Oracle para ejecutar la instrucción SQl.

- El optimizador, para cualquier sentencia SQL, realiza las operaciones siguientes:

  - **Evaluación de expresiones y condiciones** : Se evalúan expresiones y condiciones que contengan constantes de la forma más amplia posible
  - **Transformación de Instrucciones** : Para instrucciones complejas que contengan, por ejemplo, consultas o subconsultas relacionadas, el optimizador podría transformar la sentencia original en una sentencia con un join equivalente.
  - **Elección de caminos de acceso** : Para cada tabla accedida por la instrucción, el optimizador elige uno o más de los caminos de acceso disponibles para obtener los datos de la tabla.
  - **Elección de orden para combinaciones (joins)** : Para cada instrucción de combinación entre más de dos tablas, el optimizador elige qué par de tablas se unen primero, y qué tabla es combinada con el resultado, y así sucesivamente.
  - **Elección de métodos de combinación (join)** : Para cada combinación, el optimizador escoge el método idóneo a usar.

Componentes del optimizador



Transformación de sentencias

- El Parser o intérprete, realiza el Análisis Sintáctico (corrección sintáctica de las instrucciones SQL) y Semántico (verifica que los objetos utilizados y sus atributos existen y son correctos) y traduce las sentencias SQL, a la representación interna utilizada por Oracle.
- A cada parte de una sectencia sql se le llama bloque de consulta (*query block)*. La entrada del query transformer es un query "parseado" o tratado por el "Parser" o intérprete, que contiene un conjunto de query blocks.
- Esta sentencia Sql contiene dos bloques:

```
SELECT first_name, last_name
FROM   employees
WHERE  department_id
   IN  (SELECT department_id FROM departments WHERE location_id = 1800);
```

- El transformador determina cuando es ventajoso reescribir la sentencia SQL por una semanticamente equivalente pero mas eficiente.

- Técnicas de transformación de consultas:

  • Combinación de vistas: El transformador combina el "query block" que contiene la vista dentro del "query block" que lo usa.

```
CREATE VIEW employees_50_vw AS
  SELECT employee_id, last_name, job_id, salary, commission_pct, department_id
  FROM   employees
  WHERE  department_id = 50;
```

```
SELECT employee_id
FROM   employees_50_vw
WHERE  employee_id > 150;
```

```
SELECT employee_id
FROM   employees
WHERE  department_id = 50
AND    employee_id > 150;
```

La optimización de combinaciones de vistas se aplica a vistas que contienen sólo selecciones, proyecciones y joins (no contienen operadores de conjunto (union, intersect, minus..), funciones de agregado (min, max, count), distinct, group by, connect by…).

Para permitir que el optimizador use la combinación de vistas para cualquier consulta creada por el usuario, se debe conceder el permiso de MERGE ANY VIEW al usuario.

• Expansión de predicados: El optimizador empuja o traslada los predicados relevantes del "query block" analizado al "query block" de la vista. Para vistas que no son combinables, esta técnica mejora el subplan porque la base de datos puede usar los predicados "trasladados" para acceder a los índices o para usarlos como filtros.

```
CREATE VIEW all_employees_vw AS
  ( SELECT employee_id, last_name, job_id, commission_pct, department_id
    FROM   employees )
  UNION
  ( SELECT employee_id, last_name, job_id, commission_pct, department_id
    FROM   contract_workers );
```

```
SELECT last_name
FROM   all_employees_vw
WHERE  department_id = 50;
```

```
SELECT last_name
FROM   ( SELECT employee_id, last_name, job_id, commission_pct, department_id
         FROM   employees
         WHERE  department_id=50
         UNION
         SELECT employee_id, last_name, job_id, commission_pct, department_id
         FROM   contract_workers
         WHERE  department_id=50 );
```

- Subconsultas no anidadas: El optimizador transforma una consulta anidada en un sentencia join equivalente y luego optimiza dicho join.

```
SELECT *
FROM   sales
WHERE  cust_id IN ( SELECT cust_id FROM customers );
```

```
SELECT sales.*
FROM   sales, customers
WHERE  sales.cust_id = customers.cust_id;
```

El optimizador puede ejecutar esta transformación sólo si la sentencia join resultante devuelve exactamente las mismas filas que la consulta original y si la subconsulta no contiene funciones de agregado, como puede ser AVG.

Estimación

- El estimador determina el coste global de cada plan de ejecución. El estimador genera para ello tres tipos diferentes de medidas:

• Selectividad: representa una fracción de filas de un conjunto de filas. La selectividad está vinculada a un predicado de la consulta, como puede ser apellido='Fernández' o a una combinación de predicados.

Un predicado filtra un número concreto de filas de un conjunto. Así, la selectividad de un predicado indica cuántas filas pasan el filtro de dicho predicado. La selectividad varía o se mide de 0.0 a 1.0. Una selectividad de 0.0 significa que no hay filas seleccionadas en el resultado, mientras que un valor de 1.0 significa que todas las filas son seleccionadas.

El optimizador estima la selectividad en función de si dispone de estadisticas. Cuando las estadísticas estan disponibles, el estimador las usa para estimar la selectividad.

Si las estadisticas no estan disponibles, dependiendo del valor del parámetro ***OPTIMIZER_DYNAMIC_SAMPLING***, el optimizador o bien usa estadisticas dinámicas o un valor interno predeterminado.

- Cardinalidad: representa el número de filas del resultado.

En el contexto en el que estamos, el resultado puede ser una tabla base, una vista, un join o el resultado de aplicar un operador Group by.

- Coste: representa la medida del trabajo y recursos necesarios para la operación. El optimizador usa, como medida de dicho trabajo, el porcentaje de uso de CPU, operaciones de I/O en disco y el uso de memoria.

Este cálculo de coste se puede realizar explorando una tabla, accediendo a las filas de una tabla a través de un índice, haciendo un join entre tablas u ordenando el conjunto de filas del resultado.

El camino o ruta de acceso determina la cantidad de trabajo requerida para obtener los datos de una tabla base, dependiendo de si se accede mediante un acceso completo a la tabla(full table scan), mediante acceso por índice (index scan) ...

El coste de un join representa la combinación de los costes individuales de acceso de los dos conjuntos que unimos, mas el coste de la operación de unión.

Generación de planes

- La combinación de pasos que Oracle usa para ejecutar una instrucción es lo que se denomina Plan de Ejecución. Un plan de ejecución incluye un camino o ruta de acceso para cada tabla a la que accede la instrucción y el orden de combinación de las tablas, con el apropiado método de combinación.
- Para producir el mismo resultado, existen muchos planes posibles, constituidos por las distintas combinaciones de caminos de acceso, métodos de combinación y órden de combinación, que la base de datos puede usar.
- La base de datos optimiza bloques de consulta por separado, de abajo hacia arriba. La base de datos optimiza el bloque de consulta más interno primero y genera un **subplan** para ella, y luego genera por último el bloque de consulta externa que representa a toda la consulta.
- El número de posibles planes de un query block es proporcional al número de items de la claúsula FROM.

- El generador de planes utiliza un “punto de corte interno” para reducir el número de planes a estudiar para encontrar el plan de mas bajo costo.
- El punto de corte se basa en el coste del mejor plan actual. Si el mejor costo actual es grande, entonces el generador de planes explora planes alternativos, para encontrar un plan de menor costo. Si el mejor costo actual es ya pequeño, entonces el generador termina la búsqueda rápidamente porque mejorar aún más el coste no será significativo.
- El punto de corte funciona bien si el generador comienza con un orden de combinación inicial que produzca un plan con un costo cercano al óptimo.
- Encontrar ese buen orden de combinación inicial es el problema

Optimizador de caminos de acceso

- Los caminos de acceso son las formas en que los datos se recuperan de la base de datos.
- En general, los accesos de índice son útiles para los estados que recuperan un pequeño subconjunto de filas de la tabla, mientras que los accesos completos a las tablas son más eficientes cuando se accede a una gran parte de la tabla.
- Las formas más comunes de acceder a un determinado registro de una tabla referenciada por una sentencia SQL son :
- **Mediante un acceso completo a la tabla (Full Table Scans)**

  Se accede a todos los registros de la tabla, descartándose aquellos que no cumplan los criterios de selección.

  Cuando Oracle ejecuta un acceso completo, los bloques se leen secuencialmente. Como los bloques están adyacentes, la base de datos puede hacer lectura simultánea de varios bloques para mejorar el rendimiento.

  El parámetro *DB_FILE_MULTIBLOCK_READ_COUNT sirve para configurar esta lectura simultánea de bloques*, con la que la base de datos puede ejecutar los accesos completos más eficientemente.

El optimizador usa este acceso cuando:

- El optimizador es incapaz de utilizar un camino de acceso mediante índice.
- El optimizador supone que el subconjunto resultante de la consulta es un porcentaje importante del total de registros de la tabla.
- La tabla es muy pequeña y se puede cargar en memoria en una única operación de entrada/salida.
- En general, cuando el optimizador perciba que el número de operaciones de E/S va a ser menor que con los caminos de acceso que utilicen índices.

❑ Mediante acceso por identificador de registro (Rowid Scans)

El rowid es un valor único que especifica la posición física exacta del registro en la base de datos. Este valor puede ser recuperado por una sentencia SELECT y utilizado posteriormente en otras sentencias que recuperen el registro en cuestión (con limitaciones, ya que el ROWID de un registro puede variar a consecuencia de actualizaciones en el valor de sus campos).

Este acceso por ROWID es utilizado casi siempre como segundo paso del acceso por índice (los índices contienen el identificador de los registros a los que apuntan).

❑ Acceso por índice (Index Scans)

Las filas son recuperadas haciendo uso de los valores de las columnas indexadas, utilizadas en la sentencia SQL.

Para ejecutar el acceso por índice, Oracle busca el índice en los valores de las columnas accedidas por la sentencia. Si la sentencia accede sólo a columnas del índice, lee los valores de las columnas indexadas directamente desde el índice, mejor que de la propia tabla.

El índice no sólo contiene el valor indexado, sino también el rowid. De esta forma, si la sentencia accede a otras columnas, ademas de las indexadas, se realiza el acceso mediante ROWID para recuperar los valores del resto de las columnas. Un acceso mediante índice puede ser :

•Acceso por índice único (index unique scan) : El acceso busca una única clave a través de un índice único. Eso significa que siempre la búsqueda devolverá un resultado, en otro caso ya no sería UNIQUE. Un ejemplo claro de este tipo de acceso es el que hacemos cuando buscamos a través de la PK

•Acceso por rango de índice (index range scan) : Como su nombre indica, en este caso no buscamos por un único valor, sino por un rango de ellos, por lo que nos devolverá más de un registro. Esto ocurre cuando usamos los operadores de rango (<, >, <>, <=, >=, BETWEEN), en el que se utilizan valores de algunas o todas las columnas de un índice (UNIQUE ó no) de la tabla. Normalmente este acceso se realiza en orden ascendente de los valores de las columnas del índice, a no ser que se solicite mediante la clausula ORDER BY que se recuperen de forma descendente.

- Index fast full scan : se da cuando se van a utilizar únicamente columnas del índice. Se realiza un full scan sobre los bloques del índice, en vez de sobre los bloques que componen la tabla. En el índice, debe haber alguna columna NOT NULL (para asegurar que hay un registro en el índice por cada registro de la tabla), y es conveniente analizar de vez en cuando el índice, para que el optimizador por costes tenga en cuenta esta posibilidad.
- Acceso mediante combinaciones de índices (index joins) : se da cuando todas las columnas necesarias para la consulta están presentes, en el conjunto de los distintos índices de la tabla, y la optimización por costes está disponible.

❑ ¿Cómo selecciona el optimizador el camino de acceso?

El optimizador primero determina que caminos de acceso estan disponibles a través de una revisión de las condiciones de la claúsula Where y From de la sentencia SQL.

El optimizador genera un conjunto de posibles planes de ejecución, en base a los caminos de acceso disponibles y estima el costo de cada uno, utilizando las estadísticas para los índice, columnas y tablas manejadas por la sentencia.

Finalmente, el optimizador elige el plan de ejecución con el menor costo estimado.

El optimizador de consultas puede estar influenciado por:

- Sugerencias de optimización: Se puede "influir" al optimizador para usar un camino específico de acceso, usando una sugerencia/pista(hint).
- Estadísticas antiguas.

Métodos de combinación de tablas

- Los joins son sentencias que devuelven datos de más de una tabla (o de una única tabla referenciada varias veces). Un join esta caracterizado por múltiples tablas en la cláusula FROM y la relación entre estas tablas es definida a través de la condición WHERE. Los joins pueden ser de tipo inner (internos) o outer (externos).

- El optimizador debe tomar las siguientes decisiones:

  - Caminos de acceso: En cuanto a las sentencias simples, el optimizador debe elegir el camino de acceso para recuperar los datos de cada tabla incluida en el join.
  - Método de combinación (Join): Para unir cada par de orígenes de datos, Oracle debe ejecutar una operación de combinación (join). Los métodos de combinación incluyen bucles anidados, Bucles anidados externos, combinaciones de ordenación y fusión, combinaciones cartesianas y combinaciones hash.
  - Ordenación de join: Para ejecutar una sentencia que combina mas de dos tablas, Oracle combina dos de las tablas y luego combina el resultado con la siguiente tabla. Este proceso continua hasta que todas las tablas están aplicadas en el resultado.

❑ Bucles anidados (Nested Loop Joins)

El optimizador utiliza este método con conjuntos de datos pequeños, y en el caso de que la condición de join sea un camino de acceso eficiente para la tabla "interior". El optimizador determina cuál es la tabla "conductora", y ejecuta un bucle accediendo a los registros de esta tabla que cumplan las condiciones de la consulta. Para cada registro de la tabla conductora, se ejecuta un bucle accediendo a la tabla interior, mediante los valores actuales de la tabla conductora. Por tanto, es importante que el camino de acceso a la tabla interior sea muy eficiente

Este es el JOIN típico. Se recorren los registros de la tabla conductora A y luego se prueba que cada uno de los registros de A esté en B (tabla interior). Hay que intentar siempre filtrar al máximo la tabla de unión (en el caso exacto la A) y que esta tabla fuente tiene que ser, si es posible, la más pequeña de ambas (incluyendo filtrados).

❑ Bucles anidados externos (Nested Loop Outer Joins)

En caso de combinaciones externas (outer joins), si se cumplen las condiciones de elección de Bucles Anidados, se puede utilizar el método de Bucles Anidados Externos, en el que la tabla conductora es siempre la tabla "exterior" (mientras que, en el método anterior, se puede elegir a una u otra tabla como tabla conductora)

❑ Combinaciones hash (Hash Joins)

Este método es usado para enlazar grandes conjuntos de datos, en los que la condición de join es de igualdad. El optimizador usa la más pequeña de las dos tablas fuentes para construir una tabla hash sobre la clave de join, en memoria. Cuando se recuperan los registros de la tabla más grande, se examina la tabla hash para encontrar las filas enlazadas en la otra tabla. Esta situación es óptima cuando la tabla menor es lo suficientemente pequeña para caber en la memoria disponible, ya que el coste se limita a una simple lectura del dato de la tabla menor. Sin embargo, si la tabla hash es demasiado grande, el optimizador la divide en varias particiones, que se escriben en diferentes segmentos temporales de disco, que son leídos a medida que son necesitados, lo que disminuye la eficiencia

❑ Combinaciones de ordenación y fusión (Sort Merge Joins)

Consisten en una fusión de los dos conjuntos de datos, previamente ordenados por las columnas de la condición de join.

Es una variación de los bucles anidados, pero donde la base de datos ordena los conjuntos de datos, en caso que no estén ya ordenados (SORT JOIN). La base de datos busca, para cada fila del primer conjunto de datos, registros coincidentes en el segundo conjunto de datos (MERGE JOIN), partiendo de la posición de inicio de la iteración anterior. Se utilizan para combinar registros de dos fuentes independientes, especialmente si ya están ordenados mediante las columnas de la condición de join, o si la condición de join incluye algún operador que no sea de igualdad. En otro caso, es mejor utilizar las combinaciones hash. Existe un método similar, para los outer joins, denominado Sort Merge Outer Joins.

- **Combinaciones Cartesianas (Cartesian Joins)**

  Consisten en la combinación de dos fuentes de registros, entre las que no hay definida ninguna condición de join. Su coste es muy elevado, en función de los volúmenes de las dos fuentes de registros.

- **Combinaciones Externas Completas (Full Outer Joins)**

  Consisten en combinaciones externas en las que se deben preservar los registros de ambas fuentes de datos (usando las palabras clave FULL OUTER JOIN).

- **Anti-Combinaciones (Anti-Joins)**

  Son combinaciones en las que se devuelven los registros de la primera fuente de datos que no cumplen las condiciones impuestas a la segunda fuente (operadores NOT IN con subconsultas ó NOT EXISTS).

- **Semi-Combinaciones (Semi-Joins)**

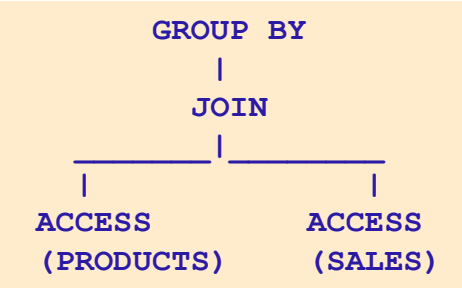
  Son combinaciones en las que se devuelven los registros de la primera fuente de datos que cumplen las condiciones impuestas a la segunda fuente (operadores IN con subconsultas ó EXISTS).

EXPLAIN PLAN

- El comando **EXPLAIN PLAN** muestra la ejecución de los planes elegidos por el optimizador de Oracle para sentencias SELECT, UPDATE, INSERT y DELETE. El plan de ejecución es la secuencia de operaciones que la base de datos sigue para ejecutar la instrucción.
- La salida del plan de ejecución muestra la siguiente información:
  - Una ordenación de las tablas incluidas en la instrucción SQL.
  - Un método de acceso de cada tabla de la instrucción.
  - Un método de combinación de las tablas afectadas por join en la instrucción.
  - Datos de operaciones como filtrar, ordenar o agregación.

```
select prod_category, avg(amount_sold)
from sales s, products p
where p.prod_id = s.prod_id
group by prod_category;
```

```
-----------------------------------------
 Id   Operation               Name
-----------------------------------------
   0  SELECT STATEMENT
   1   HASH GROUP BY
   2    HASH JOIN
   3     TABLE ACCESS FULL   PRODUCTS
   4     PARTITION RANGE ALL
   5      TABLE ACCESS FULL  SALES
-----------------------------------------
```



- Además del árbol de orígenes de datos, la tabla del plan contiene información sobre optimización, costo y cardinalidad de cada operación

```
-----------------------------------------------------
 Id   Operation               Name      Cost (%CPU)
-----------------------------------------------------
   0  SELECT STATEMENT                    17  (18)
   1   HASH GROUP BY                      17  (18)
*  2    HASH JOIN                         15   (7)
   3     TABLE ACCESS FULL   PRODUCTS      9   (0)
   4     PARTITION RANGE ALL               5   (0)
   5      TABLE ACCESS FULL  SALES         5   (0)
-----------------------------------------------------
```

Como pueden cambiar los planes de ejecución

- Conforme las entradas en las que se basa el optimizador cambian, los planes de ejecución pueden y deben cambiar.
- La salida del EXPLAIN PLAN muestra como Oracle debería ejecutar la sentencia cuando ésta fué analizada. Este plan puede diferir del plan de ejecución actual de la sentencia debido a las diferencias entre el entorno de ejecución y el usado por explain plan.

- Los planes de ejecución pueden diferir debido a:

Diferentes esquemas

- La ejecución y el explain plan tienen lugar en diferentes bases de datos.
- El usuario que estudia/analiza la sentencia es diferente del que la ejecuta. Dos usuarios podrían estar apuntando a diferentes objetos de la misma base de datos, lo que produce diferentes planes de ejecución.
- Los cambios en el esquema (por lo general cambios en los índices) entre las dos operaciones (Explain plan y ejecución real).

Diferentes costes

Algunos factores que afectan a los costes son:

- Volumen de datos y estadísticas
- Los tipos de variables de vinculación y valores
- Cambios en los parámetros de inicialización a nivel global o a nivel de sesion

Mirando más allá de los planes de ejecución

- El plan de ejecución de la operación, por sí sola, no puede diferenciar entre declaraciones bien afinadas y las que funcionan mal.
- Por ejemplo, una salida de EXPLAIN PLAN que muestra que una sentencia Sql utiliza un índice, no significa necesariamente que la sentencia se ejecute de forma eficiente. A veces los índices son extremadamente ineficientes. En este caso, se debe examinar lo siguiente:
  - Las columnas del índice que están siendo usadas
  - Su selectividad (fraccion de la tabla que es accedida)
- Lo mejor es usar EXPLAIN PLAN para determinar un plan de acceso, y luego probar que es el plan óptimo a través de pruebas. Al evaluar un plan, se debe examinar el consumo real de recursos de la sentencia.

La tabla de salida PLAN_TABLE

- Esta tabla (PLAN_TABLE) es creada automáticamente como un sinónimo público de una tabla temporal global.
- PLAN_TABLE es la tabla por defecto de salida en la que la sentencia EXPLAIN PLAN inserta las filas describiendo el plan de ejecución.
- La tabla PLAN_TABLE es creada automáticamente para cada usuario. Podemos usar el script SQL ***catplan.sql*** para crear manualmente la tabla temporal global y el sinónimo PLAN_TABLE. El nombre y localización de este script depende del sistema operativo.
- Oracle recomienda que se borre y reconstruya la tabla local PLAN_TABLE después de actualizar la versión de la base de datos, porque las columnas pueden haber cambiado.

Ejecutando EXPLAIN PLAN

- Para aplicar EXPLAIN PLAN a una sentencia SQL, debemos usar el comando *EXPLAIN PLAN y a continuación la claúsula FOR* inmediatamente antes de la sentencia Sql a analizar. Por ejemplo:

```
EXPLAIN PLAN FOR
  SELECT last_name FROM employees;
```

- Con esto, el plan de ejecución se inserta en la tabla PLAN_TABLE, pudiendo ahora consultarlo desde dicha tabla.
- Se puede especificar un identificador para la sentencia Sql que se desee analizar, que sirva como referencia del plan de ejecución incorporado a PLAN_TABLE:

```
EXPLAIN PLAN
  SET STATEMENT_ID = 'st1' FOR
  SELECT last_name FROM employees;
```

Mostrando la salida de PLAN_TABLE

- Tras analizar una sentencia sql y que su resultado se haya incorporado a PLAN_TABLE, podemos usar posteriormente los siguientes scripts o paquetes PL/SQL proporcinados por oracle, para mostrar la tabla plan table :
  - *UTLXPLS.SQL* : Este script muestra la tabla de salida PLAN TABLE en orden secuencial de procesamiento.
  - *UTLXPLP.SQL* : Incluye columnas de ejecución en paralelo.
  - *DBMS_XPLAN.DISPLAY*: Esta función acepta opciones, que permiten seleccionar el tipo de salida que muestra el plan. Estas opciones son:
    - El nombre de la tabla de plan, si hemos usado una tabla diferente de PLAN_TABLE (default '*PLAN_TABLE*')
    - Un identificador de sentencia (ID), si hemos asignado dicho identificador a una sentencia analizada con EXPLAIN PLAN (default *null*)
    - Una opción de formato que determina el nivel de detalle a mostrar: *BASIC*, *TYPICAL* y *ALL* (default '*TYPICAL*')

- Veamos algunos ejemplos de uso de DBMS_XPLAN, que muestra la salida de PLAN_TABLE:

```
SELECT PLAN_TABLE_OUTPUT
   FROM TABLE(DBMS_XPLAN.DISPLAY());
```

```
SELECT PLAN_TABLE_OUTPUT
   FROM TABLE(DBMS_XPLAN.DISPLAY('MY_PLAN_TABLE', 'st1', 'TYPICAL'));
```

Leyendo la salida de EXPLAIN PLAN

```
EXPLAIN PLAN
   SET statement_id = 'ex_plan1' FOR
SELECT phone_number
 FROM employees
 WHERE phone_number LIKE '650%';
```

```
-------------------------------------
| Id  | Operation         | Name      |
-------------------------------------
|   0 | SELECT STATEMENT  |           |
|   1 |  TABLE ACCESS FULL| EMPLOYEES |
-------------------------------------
```

Este plan muestra la ejecución de un sentencia SQL "SELECT". La tabla *employees* es accedida usando un "full table scan".

- Cada fila de la tabla *employees* es accedida y el criterio de la claúsula WHERE se evalúa para cada fila.
- La sentencia SELECT devuelve las filas que cumplen el criterio expresado en el WHERE.

```
EXPLAIN PLAN
  SET statement_id = 'ex_plan2' FOR
SELECT last_name FROM employees
 WHERE last_name LIKE 'Pe%';
```

```
SELECT PLAN_TABLE_OUTPUT
   FROM TABLE(DBMS_XPLAN.DISPLAY(NULL,'ex_plan2','BASIC'));

-----------------------------------------
| Id  | Operation         | Name        |
-----------------------------------------
|   0 | SELECT STATEMENT  |             |
|   1 |  INDEX RANGE SCAN| EMP_NAME_IX |
-----------------------------------------
```

Este plan muestra la ejecución de la sentencia SELECT, reflejando que:

- La base de datos se apoya en el exploración del índice *EMP_NAME_IX* para evaluar el criterio de la claúsula WHERE.
- La sentencia SELECT devuelve las filas que cumplen la condición.

Operaciones y Opciones de la tabla PLAN_TABLE

Cada registro de PLAN_TABLE se corresponde con una determinada operación de base de datos. Cada operación posible, dentro de un plan de ejecución, puede ser modificada mediante una serie de opciones. Las principales operaciones, con sus respectivas opciones, son :

- TABLE ACCESS : operación de acceso a datos de una tabla. Sus principales opciones son:
  - FULL : se accede a todos los registros de la tabla.
  - SAMPLE : se realiza un muestreo de registros de la tabla.
  - BY ROWID RANGE : se recuperan los registros de un rango de ROWIDs.
  - SAMPLE BY ROWID RANGE : se recupera una muestra de los registros de un rango de ROWIDs.
  - BY USER ROWID : se recuperan registros mediante ROWIDs proporcionados explícitamente en la sentencia SQL.

- BY INDEX ROWID : se recuperan registros mediante ROWIDs proporcionados por una operación de acceso con índice.
- BY GLOBAL INDEX ROWID : en el caso de tablas particionadas, se recuperan registros mediante ROWIDs de índices globales de la tabla.
- BY LOCAL INDEX ROWID : para tablas particionadas, se recuperan registros mediante ROWID de índices locales de particiones.

❑ INDEX : operación de acceso a datos mediante índice. Sus principales opciones :

UNIQUE SCAN : recuperación de un único identificador de registro.

RANGE SCAN : recuperación de una serie de identificadores de registro, en orden ascendente del índice.

RANGE SCAN DESCENDING : recuperación de una serie de ROWIDs, en orden descendente del índice.

❑ VIEW : recuperación de datos de una vista.

❑ SEQUENCE : recuperación de datos de una secuencia

- REMOTE : recuperación de datos de una base de datos remota.
- PARTITION : recuperación de datos de particiones de una tabla. Algunas de sus opciones son :
  - SINGLE : acceso a una única partición
  - ITERATOR : acceso a varias particiones
  - ALL : acceso a todas las particiones de la tabla
- SORT : operaciones de ordenación y agrupamiento de datos. Algunas opciones :
  - AGGREGATE : resultado de agrupaciones sin clausula GROUP BY.
  - UNIQUE : ordenación para eliminar duplicados.
  - GROUP BY : resultado de agrupaciones con clausula GROUP BY.
  - JOIN : ordenación previa a una combinación MERGE JOIN.
  - ORDER BY : ordenación de acuerdo a una clausual ORDER BY.
- COUNT : devuelve el número de registros de una tabla.

- FILTER : toma un conjunto de registros y devuelve el subconjunto que cumpla con una serie de restricciones.
- FIRST ROW : devuelve únicamente el primer registro de una consulta
- FOR UPDATE : recupera y bloquea un conjunto de registros, por si es necesario hacerles alguna modificación.
- AND-EQUAL : combina varios conjuntos de ROWIDs, y devuelve la intersección entre ellos, eliminando duplicados.
- HASH JOIN : operación de combinación de dos conjuntos de registros.
- MERGE JOIN : operación de combinación de dos conjuntos de registros, previamente ordenados. Puede tener la opción OUTER, entre otras.
- NESTED LOOPS : operación de combinación de dos conjuntos de registros. Puede tener la opción OUTER, entre otras.
- UNION : esta operación toma dos conjuntos de registros y devuelve su unión, eliminando registros duplicados.

- CONCATENATION : esta operación toma dos conjuntos de registros y devuelve su unión, sin eliminar duplicados.
- INTERSECTION : toma dos conjuntos de registros y devuelve su intersección.
- MINUS : toma dos conjuntos de registros y devuelve los registros del primero que no estén en el segundo, eliminando duplicados.
- CONNECT BY : recupera un conjunto de registros en orden jerárquico

Usando vistas V$SQL_PLAN

- También se pueden utilizar las vistas V$SQL_PLAN para mostrar el plan de ejecución de una sentencia SQL. Su definición es similar a la PLAN_TABLE.
- La vista V$SQL_PLAN_STATISTICS ofrece las estadísticas de ejecución actuales para cada operación en el plan, como el número de filas de salida y el tiempo transcurrido. Todas las estadísticas, excepto el número de filas de salida, son acumulativos. Por ejemplo, las estadísticas de una operación join también incluye las estadísticas de sus dos entradas
- La vista *V$SQL_PLAN_STATISTICS_ALL* permite comparaciones de las estimaciones que proporcina el optimizador para el número de filas y tiempo transcurrido. Esta vista combina la información de *V$SQL_PLAN* y de *V$SQL_PLAN_STATISTICS*.

Habilitar características del optimizador de consultas

- El parámetro de inicialización OPTIMIZER_FEATURES_ENABLE permite una serie de funciones relacionadas con el optimizador, dependiendo de la versión.
- Oracle no recomienda configurar explícitamente el parámetro OPTIMIZER_FEATURES_ENABLE a una versión anterior a la 10

```
SHOW PARAMETER optimizer_features_enable

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------
optimizer_features_enable            string      11.2.0.2
```

```
ALTER SYSTEM SET optimizer_features_enable='10.2.0.5';
```

Controlando el comportamiento del optimizador

- Algunos de los parámetros de inicialización que se pueden utilizar para controlar el comportamiento del optimizador de consultas son los siguientes:

Initialization Parameter	Description
DB_FILE_MULTIBLOCK_READ_COUNT	Especifica el número de bloques que se leen en una sola E / S durante un full table scan o index fast full scan. Los valores más altos resultan en un costo más favorable para los recorridos de tablas completos (full table scan) y pueden dar lugar a que el optimizador elija el mismo frente a un recorrido de índice (index scan)
OPTIMIZER_MODE	Establece el modo del optimizador al iniciar la instancia. Los posible valores son ALL_ROWS, FIRST_ROWS_n y FIRST_ROWS.
PGA_AGGREGATE_TARGET	Establece la cantidad de memoria asignada a las ordenaciones y los hash joins. Mayores cantidades de memoria asignadas para las ordenaciones y hash joins reducen el costo de optimización de estas operaciones.

Elegir una meta para el optimizador

- Podemos establecer los siguientes objetivos para el optimizador:

  • Mejor rendimiento(Best throughput)(default)

  La base de datos usa la menor cantidad de recursos necesarios para procesar todas las filas accedidas por la sentencia.

  Por lo general, el rendimiento es más importante en aplicaciones por lotes, ya que el usuario que inicia la aplicación sólo está preocupado por el tiempo necesario para que la aplicación se complete. El tiempo de respuesta es menos importante porque el usuario no examina los resultados de las sentencias individuales, mientras se ejecuta la aplicación.

  • Mejor tiempo de respuesta (Best response time)

  La base de datos utiliza la menor cantidad de recursos necesarios para procesar la primera fila accedida por el instrucción SQL.

  Por lo general, el tiempo de respuesta es importante en aplicaciones interactivas, pues el usuario interactivo está esperando a ver la primera o primeras filas accedidas por la instrucción.

Configuración del parámetro de inicialización OPTIMIZER_MODE

Value	Description
ALL_ROWS	El optimizador utiliza un enfoque basado en los costos para todas las sentencias SQL en la sesión, independientemente de la presencia de las estadísticas y optimiza con el objetivo del mejor rendimiento (mínimo consumo de recursos para completar la totalidad de la instrucción). Este es el valor por defecto.
FIRST_ROWS_n	El optimizador utiliza un enfoque basado en los costos, independientemente de la presencia de las estadísticas, y optimiza con el objetivo del mejor tiempo de respuesta para devolver las n primeras filas, donde n es igual a 1, 10, 100 o 1000.
FIRST_ROWS	El optimizador utiliza una mezcla de costos y heurística para encontrar el mejor plan para la entrega rápida de las primeras filas. Tengamos en cuenta que el uso de heurísticas a veces lleva al optimizador a generar un plan con un costo que es significativamente mayor que el costo de un plan sin aplicar heurística. FIRST_ROWS está disponible por compatibilidad con versiones anteriores. Use FIRST_ROWS_n en su lugar.

- Se puede cambiar el objetivo del optimizador de querys, para las instrucciones SQL de una sesión, estableciendo el valor del parámetro en el fichero de inicialización o a través de la instrucción *ALTER SESSION SET OPTIMIZER_MODE*.
- Por ejemplo, la siguiente instrucción SQL cambia el objetivo del optimizador para la sesión actual al mejor tiempo de respuesta:

```
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS_1;
```

- Si el optimizador usa el enfoque basado en los costes para una instrucción SQL y algunas de las tablas accedidas por la instrucción no tiene estadísticas, el optimizador usa información interna, como el número de bloques de datos asignado a estas tablas, para estimar otras estadísticas con las que trabajar para dichas tablas.

Estadísticas del Optimizador en el Diccionario de Datos

- Las estadísticas proporcionan al optimizador de consultas información sobre la unicidad y la distribución de datos. Con esta información, el optimizador es capaz de calcular los costos del plan con un alto grado de precisión y elegir el mejor plan de ejecución basado en el menor costo.
- El optimizador decide automáticamente si se debe utilizar estadísticas dinámicas o no, dependiendo de si las estadísticas disponibles son suficientes para generar un plan de ejecución óptimo. Si las estadísticas disponibles no son suficientes, entonces el optimizador utiliza estadísticas dinámicas para aumentar y completar las estadísticas existentes
- A partir de Oracle Database 11g Release 2 (11.2.0.4), el parámetro de inicialización OPTIMIZER_DYNAMIC_SAMPLING dispone de un valor configurable (Level 11) que permite al optimizador recopilar estadísticas dinámicas cuando lo considere necesario.

Level	Cuando el optimizador utiliza estadísticas dinámicas	tamaño de la muestra (Blocks)
0	No se usan estadísticas dinámicas	n/a
1	Utilizar las estadísticas dinámicas para todas las tablas que no tienen estadísticas, pero sólo si se cumplen los siguientes criterios: - En la consulta hay al menos 1 tabla sin particiones, que no tiene estadísticas. - Esta tabla no tiene índices. - Esta tabla tiene más bloques que el número de bloques que se utilizaría para las estadísticas dinámicas de esta tabla.	32
2	Utilizar las estadísticas dinámicas si al menos una tabla en la instrucción no tiene estadísticas. Esta es la configuración por defecto.	64
3	Utilizar las estadísticas dinámicas si alguna de las siguientes condiciones: - La declaración cumple criterios del nivel 2. - La declaración tiene una o más expresiones utilizadas en los predicados de la cláusula WHERE, por ejemplo, Where SUBSTR (cust_last_name, 1,3).	64
...	...	...
11	Utilizar las estadísticas dinámicas automáticamente siempre que lo considere necesario el optimizador.	Automatico

Utilización de las sugerencias (hint) del Optimizer

- Un **hint** es una instrucción para el optimizador. El diseñador o programador de una aplicación puede conocer información sobre los datos que no esté disponible para el optimizador de Oracle (Sobre su distribución presente o futura, sobre selectividad de índices, etc). Basado en esta información, puede elegir un plan de ejecución más eficiente que el optimizador. Para este caso, Oracle proporciona los **hints** (sugerencias), un mecanismo para que el programador pueda transmitir esta información al Optimizador Basado en Costes, para forzarlo a utilizar un plan de ejecución alternativo.
- Por ejemplo, podemos saber que un determinado índice es más selectivo para determinadas consultas. En este caso, podemos utilizar sugerencias para instruir al optimizador con vistas a utilizar un mejor plan de ejecución.

- Las sugerencias se pasan como comentarios en sentencias SQL, de manera que no afectan al código que deba ser ejecutado sobre bases de datos sin estas características. Se incluye el comentario-sugerencia inmediatamente después de la palabra clave INSERT, UPDATE, DELETE y SELECT. Si no se introducen correctamente, o no tienen significado, se ignoran, sin que la base de datos devuelva un error.
- En un mismo comentario se pueden introducir varias sugerencias, empezando con un signo más, y separando sucesivas sugerencias con espacios.
- La desventaja de los hits es el código adicional que debemos manejar, verificar y controlar. Los cambios en la base de datos y en su entorno pueden hacer que los hints estén obsoletos o incluso tener consecuencias negativas

- Oracle soporta mas de 60 hints, cada uno de los cuales puede tener de 0 o varios parámetros.
- Un bloque de intrucción puede tener sólo un comentario conteniendo hints, debiendo ir el mismo a continuación del término SELECT, UPDATE, INSERT, MERGE o DELETE.
- Por ejemplo, el siguiente hint dirige al optimizador para escoger el plan de consulta que devuelva las primeras 10 filas de la tabla de empleados con el menor coste:

```
SELECT /*+ FIRST_ROWS(10) */
  FROM employees;
```

- Hints para decidir el método de Acceso:

Para determinar el método de acceso a una tabla, se utilizan las siguientes sugerencias

- /*+ FULL(tabla) */ : FULL SCAN sobre la tabla indicada.
- /*+ CLUSTER(tabla) */ : acceso por cluster a la tabla.
- /*+ HASH(tabla) */ : acceso por hash a la tabla.
- /*+ INDEX(tabla índice) */ : acceso por un determinado índice a la tabla.
- /*+ NO_INDEX(tabla índice) */ : desactiva el uso de los índices indicados.
- /*+ INDEX_ASC(tabla índice) */ : acceso ascendente por un determinado índice a la tabla.
- /*+ INDEX_DESC(tabla índice) */ : acceso descendente por un determinado índice a la tabla.

- /*+ INDEX_FFS(tabla índice) */ : acceso FAST FULL INDEX para la tabla.
- /*+ NO_INDEX_FFS(tabla índice) **/**: Hace que el optimizador excluya FAST FULL INDEX de los índices específicos indicados de la tabla
- /* + INDEX_JOIN (tabla índices) **/**: Instruye al optimizador para usar una combinación de índices como método de acceso. Para que el hint tenga un efecto positivo, debe existir un reducido número de índices para poder responder a la petición del query
- …

- Hints para decidir el orden de combinación:

  • **LEADING**: Indicamos al optimizador que use el conjunto de tablas especificadas como prefijo del plan de ejecución.

  • **ORDERED**: Indicamos que haga el join de las tablas en el orden en el que aparecen en la claúsula FROM. Oracle recomienda usar LEADING, que es mas versátil, que el hint ORDERED .

  Es posible utilizar la sugerencia ORDERED para especificar un orden de combinación específico, si se sabe algo que el optimizador no sabe sobre el número de filas seleccionadas de cada tabla.

  Con esta información, le puede permitir elegir una tabla interna y externa, mejor de lo que podría hacer el optimizador.

- Hints para decidir el método de combinación:

  • **USE_NL**: dirige al optimizador para realizar un join de cada tabla especificada con las filas de otro origen de datos, con un nested loops join, usando la tabla especificada como tabla interna (inner table).

  /*+ USE_NL(tabla1 tabla2) */ : indica un Nested Loop Join entre las tablas tabla1 y tabla2, con la tabla interior tabla1.

  El uso de USE_NL and USE_MERGE hints es recomendable conjuntamente con los hints LEADING y ORDERED. El optimizador usa estos hints cuando la tabla referenciada es forzada como tabla interna del join. Los hints son ignorados si la tabla referenciada es tabla externa.

  • **NO_USE_NL**: Lo contrario de lo anterior. Dictamos al optimizador que no aplique nested loops joins cuando haga el join entre la tabla especificada y otra fuente de datos y la tabla indicada en primer lugar sea la tabla interna.

```
SELECT /*+ NO_USE_NL(l h) */ *
  FROM orders h, order_items l
  WHERE l.order_id = h.order_id
    AND l.order_id > 3500;
```

• **USE_NL_WITH_INDEX**: Instruimos al optimizador para aplicar nested loops join con la tabla especificada y otra posible fuente de datos, siempre que la tabla especificada sea la tabla interior y bajo la siguiente condición:

Si no se especifica ningún índice, el optimizador debe ser capaz de encontrar y utilizar algún índice presente como predicado del join. Si se especifica un índice, el optimizador debe ser capaz de utilizar este índice, presente como predicado del join, como clave de índice.

```
SELECT /*+ USE_NL_WITH_INDEX(l item_product_ix) */ *
  FROM orders h, order_items l
  WHERE l.order_id = h.order_id
    AND l.order_id > 3500;
```

• **USE_MERGE**: indicamos al optimizador que haga sort-merge join entre las tablas especificadas.

• **NO_USE_MERGE**: Lo contrario de lo anterior. No se desea usar sort-merge joins entre las tablas especificadas, cuando la primera tabla sea la tabla "interior".

• **USE_HASH**: Indicamos que use hash join entre las tablas indicadas.

• **NO_USE_HASH**: Indicamos que no se use hash join entre las tablas, cuando la primera de ellas sea la tabla "interior".

- Hints para transformación de querys:

  • **NO_QUERY_TRANSFORMATION**: indicamos al optimizador que no haga transformación en el query.

  • **MERGE**: Permitimos mezclar vistas en una consulta. Cuando se usa MERGE sin argumento, debe estar emplazado en la vista. Cuando se usa con el nombre de la vista como argumento, debe ser emplazado en el query principal.

```
SELECT /*+ MERGE(v) */ e1.last_name, e1.salary, v.avg_salary
  FROM employees e1,
    (SELECT department_id, avg(salary) avg_salary
       FROM employees e2 GROUP BY department_id) v
  WHERE e1.department_id = v.department_id AND e1.salary > v.avg_salary;
```

  • **NO_MERGE**: Instruimos al optimizador para que la consulta 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

- **UNNEST**: Instruye al optimizador a desanidar y fusionar la subconsulta, en el cuerpo del bloque de consulta que la contiene, lo que permite que el optimizador las considere conjuntamente en la evaluación de las rutas de acceso y joins
- **NO_UNNEST**: Desactiva unnest.

- En este ejemplo, El hint LEADING hint especifica el orden exacto del join, indicando también el método:

```
SELECT /*+ LEADING(e2 e1) USE_NL(e1) INDEX(e1 emp_emp_id_pk)
           USE_MERGE(j) FULL(j) */
       e1.first_name, e1.last_name, j.job_id, sum(e2.salary) total_sal
  FROM employees e1, employees e2, job_history j
 WHERE e1.employee_id = e2.manager_id
   AND e1.employee_id = j.employee_id
   AND e1.hire_date = j.start_date
 GROUP BY e1.first_name, e1.last_name, j.job_id
 ORDER BY total_sal;
```