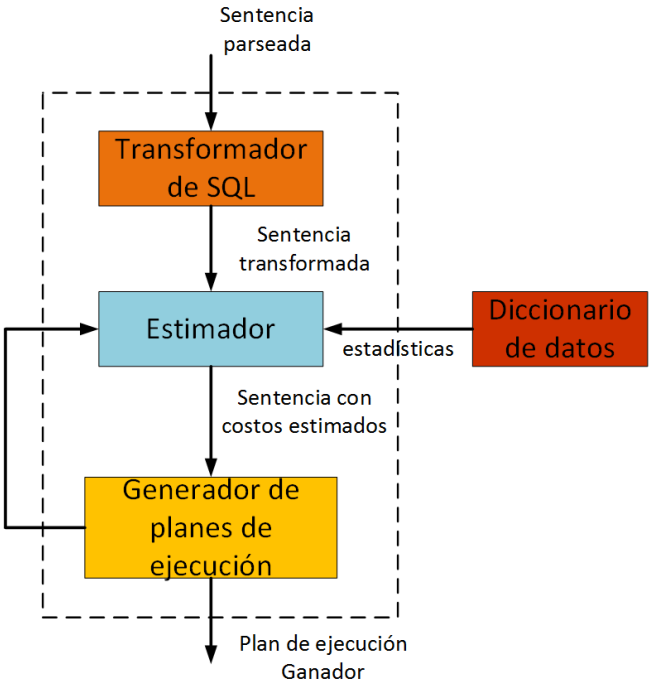
7.3. CARACTERÍSTICAS DE UN OPTIMIZADOR

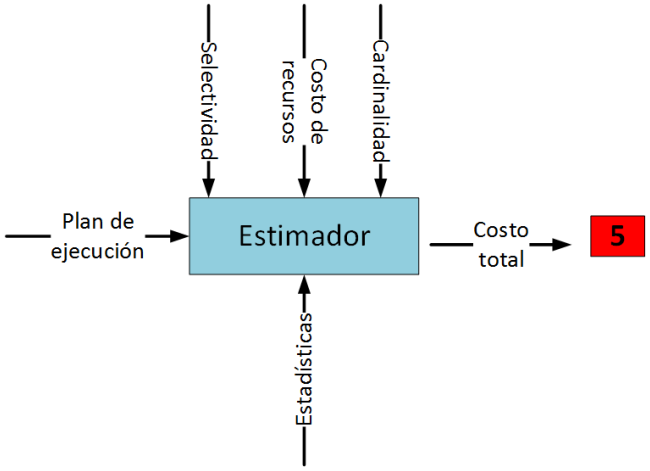
Formado por los siguientes componentes:

7.3.1. Query transformer



- El optimizador determina si es necesario re-escribir la sentencia SQL de tal forma que sea posible obtener mejores planes de ejecución.
- Debido a que SQL no es un lenguaje procedimental, el optimizador es libre de dividir, reorganizar o reescribir una sentencia y procesarla en cualquier orden.
- Transformaciones de sentencias SQL: (Ejemplos de transformaciones en Oracle).
    - OR Expansion
    - View Merging
    - Predicate Pushing
    - Subquery Unnesting
    - Query Rewrite with Materialized Views
    - Star Transformation
    - In-Memory Aggregation
    - Table Expansion
    - Join Factorization

7.3.2. Estimador.



- Su principal función es determinar el costo total (valor numérico) para un determinado plan de ejecución.
- Este cálculo se realiza considerando principalmente variables de:
    - Costo de procesamiento
    - Costo de operaciones I/O
    - Costos de comunicación
- Emplea principalmente 3 medidas para determinar el costo total:
    - *Selectividad*: Porcentaje estimado de registros que se obtendrían de una fuente de datos (una tabla, una vista, o el resultado de un join) al aplicarle un predicado. La selectividad es expresada por un número decimal en el rango [0,1]
    - *Cardinalidad*: Número de registros obtenido en cada operación contenida en el plan de ejecución.

- *Costo de recursos*: Unidades de trabajo de recursos empleados: Uso de disco, procesador y memoria.

- La estimación de los valores de selectividad y cardinalidad se obtienen a través de:
  - Estadísticas
  - Valores por default en caso de no contar con estadísticas.
  - Histogramas, en especial para distribuciones heterogéneas de los datos.

7.3.3. Generador de planes de ejecución.

- Encargado de generar y analizar posibles planes de ejecución por cada ***bloque SQL***. Una de las principales actividades es la ejecución de operaciones JOIN.
- El plan generado es enviado al estimador para determinar su costo total de ejecución.
- Cuando en una sentencia SQL existen múltiples tablas, el generador debe determinar la forma más eficiente de combinar cada pareja de tablas. Para ello, debe realizar una adecuada selección de los siguientes conceptos:
  - Selección del método de acceso a datos (Access Paths)
  - Selección de la estrategia de Ordenamiento de operaciones JOIN
  - El tipo de JOIN (inner, outer, etc.)
  - Selección de la estrategia para ejecutar una operación JOIN

7.3.3.1. Selección del método de acceso a datos (Access Paths).

Ejemplos de métodos de acceso a datos en Oracle.

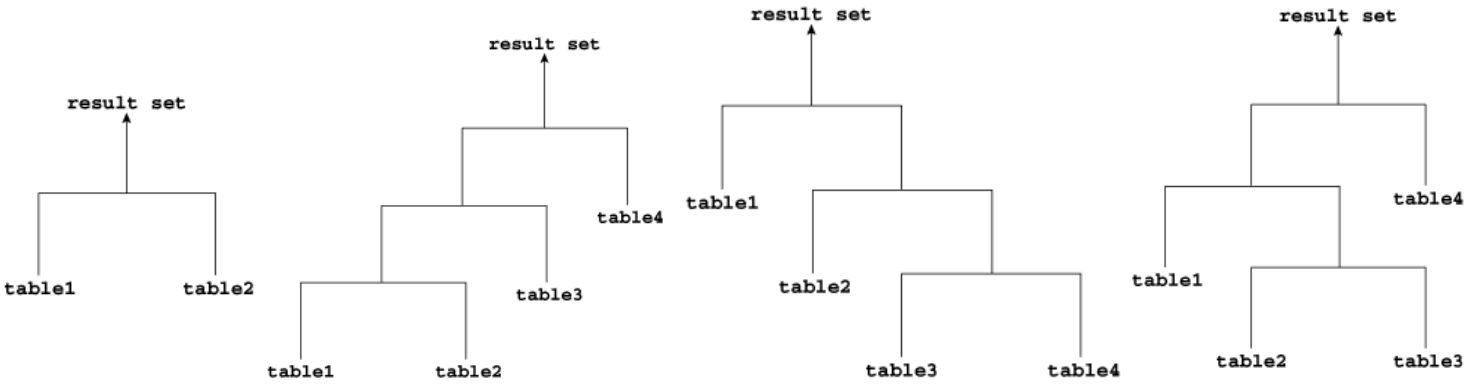
Access Path	Tablas	Índices B-Tree	Índices BitMap	Cluster Tables
Full Table Scans	X			
Table Access by Rowid	X			
Sample Table Scans	X			
Index Unique Scans		X		
Index Range Scans		X		
Index Full Scans		X		
Index Fast full scans		X		
Index Skip Scans		X		
Index Join scans			X	
BitMap index single value			X	
BitMap index Range Scans			X	
BitMap merge				
Cluster scans				X
Hash scans				X

- Un clúster de tablas está formado por un conjunto de tablas que comparten columnas en común. Los datos de dichas columnas se almacenan en el mismo bloque de datos. Esta condición permite que se requiere leer un solo bloque para obtener datos de diferentes tablas.

- Los Access paths BitMap y Hash scan hacen referencia a índices de tipo BitMap y al uso de funciones hash. Para efectos del curso se revisará únicamente Access paths para índices B – Tree.

7.3.3.2. Selección de la estrategia de Ordenamiento de operaciones JOIN.

- Join Tree
- Left Deep Join Tree
- Right Deep Join Tree
- Bushy (tupido) Join Tree



7.3.3.3. Selección del Método para ejecutar operaciones JOIN

- Nested Loops Joins
- Hash Joins
- Sort Merge Joins
- Cartesian Joins

7.3.3.4. Tipos de Joins.

El tipo de join especificado en la sentencia SQL puede influenciar al optimizador en especial para determinar el orden correcto de ejecución:

- Inner Joins
- Outer Joins
- Semijoins
- Antijoins

Una de los principales objetivos en cuanto a la ejecución de operaciones JOIN es la reducción de registros desde las primeras operaciones de tal forma que los pasos siguientes sean menos costosos.

Ejemplo:

- Tratar de ejecutar primero operaciones JOIN que generen como resultado un solo registro a lo más, es decir, detectar predicados asociados con restricciones `unique`, `primary key`:
  `where empleado_id = 10`

7.4. OPTIMIZACIÓN DE CONSULTAS EN ORACLE.

- La entrada del optimizador es una consulta SQL parseada, formada por un conjunto de "bloques". Cada instrucción `select` representa un bloque.

Ejemplo:

```
select first_name, last_name
from hr.employees
where department_id
    in (select department_id
          from hr.departments
          where location_id = 1800
    );
```

- En el ejemplo anterior existen 2 bloques SQL. Por cada bloque el optimizador genera un "Sub-plan" de ejecución. Cada sub-plan es optimizado de forma separada.
- El número posible de "Sub-planes" que pueden existir es proporcional al número de tablas listadas en la cláusula `from`.

La selección del mejor plan de ejecución depende de varios factores:

- La forma en la que se escribe la consulta.
- El tamaño de los conjuntos de datos
- Las estructuras de acceso a datos existentes.
- Los recursos del sistema existentes: memoria, procesador.
- Las estadísticas existentes.

Ejemplo:
Suponer que se desea obtener a todos los empleados que son managers.

- Si las estadísticas indican que el 80% de los empleados son managers, el optimizador seleccionaría un ***full table scan***.
- Si las estadísticas indican que el 5% de los empleados son managers, el optimizador seleccionaría un ***index scan***.

7.4.1. Planes de ejecución

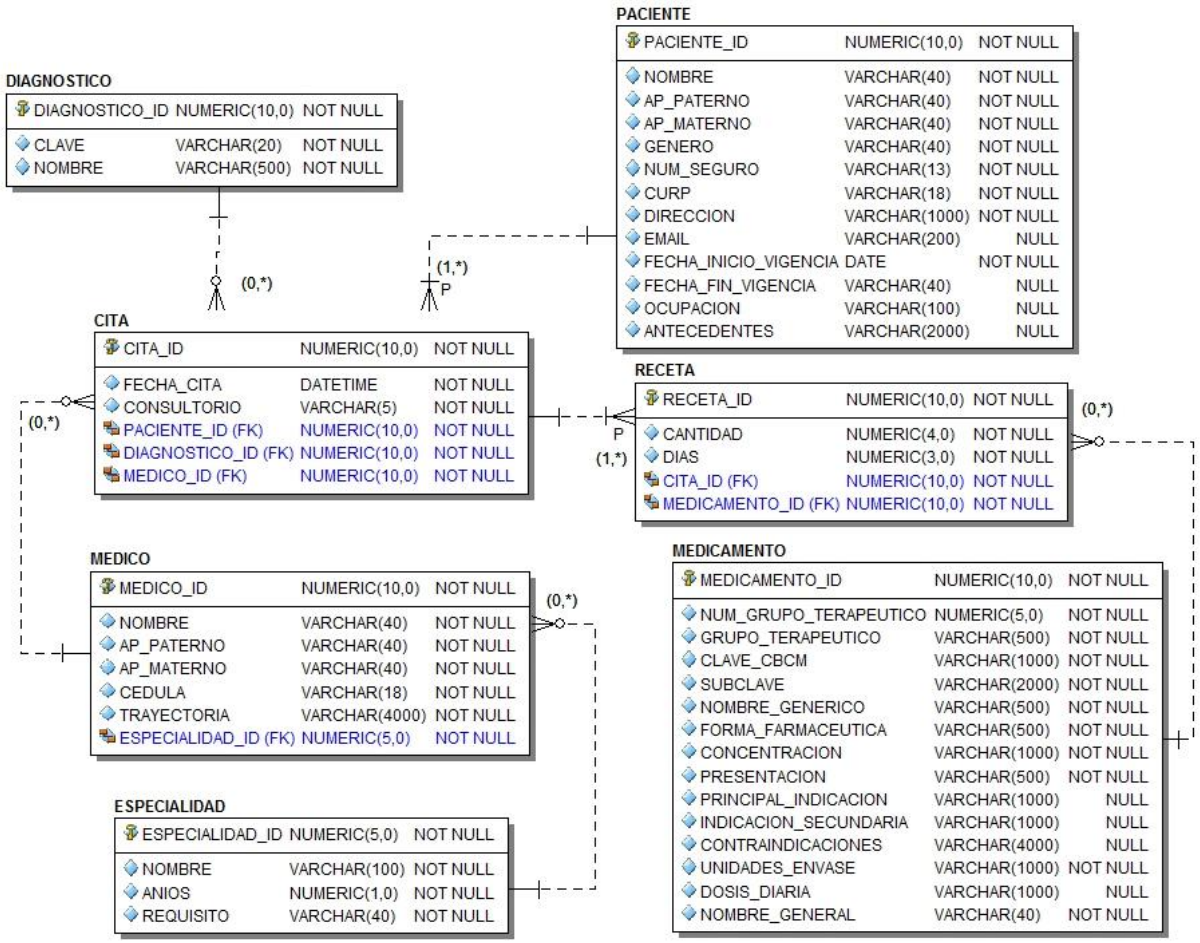
- Un plan de ejecución está formado por la combinación de pasos que la BD realiza para ejecutar una sentencia SQL.
- En cada paso se obtienen datos físicamente.
- El plan de ejecución incluye:
    - "Access Paths" para cada tabla incluida en la sentencia
    - El orden en el que se ejecutan las operaciones JOIN
    - El método seleccionado para ejecutar un JOIN.
    - Operaciones sobre los datos como son: `filter, sort, aggregation`.

- Costo y cardinalidad de cada operación.
- Información de particionamiento en caso de tablas particionadas.
- Procesamiento en paralelo.

- Para mostrar un plan de ejecución se emplea la sentencia `explain plan`.
- Al ejecutar la sentencia anterior, el optimizador elige el plan de ejecución e inserta los datos que los describen en una tabla llamada `plan_table`
- En caso de no existir, la tabla se puede crear ejecutando `@$ORACLE_HOME/rdbms/admin/catplan.sql`
- Posterior a la ejecución de la sentencia `explain plain` es posible emplear algún script o paquete proporcionado por Oracle para mostrar el plan de ejecución más reciente almacenado en `plan_table`, todos ellos ubicados en $`ORACLE_HOME/rdbms/admin`
  - `utlxpls.sql`
  - `utlxplp.sql`
  - También existe esta función para personalizar la salida del plan: `dbms_xplan.display`, por ejemplo, para definir el detalle de la salida se especifica: `basic`, `serial`, `typical`, y `all`

Ejemplo:

Considerar el siguiente modelo relacional para ilustrar los ejemplos que corresponde con el control de citas y médicos de un hospital. Descomprimir el archivo `ejemplo-control-medico.zip` y ejecutar el script `s-01-control-medico-main.sql`



Importante: Ejecutar la recolección de estadísticas sobre el esquema que contiene a estas tablas para obtener buenos resultados. Para ello, ejecutar con el usuario `sys` el siguiente código en la PDB donde se encuentra el modelo. Cambiar el nombre del usuario (`ownname`) en caso de ser necesario

```
begin
  dbms_stats.gather_schema_stats (
      ownname => 'CONTROL_MEDICO',
      degree  => 2
  );
end;
/
```

- Obtener el plan de ejecución para la siguiente consulta:

Mostrar el nombre de los médicos y las fechas de sus citas programadas en el consultorio C-593

```
SQL>
explain plan
set statement_id ='s1' for
select m.nombre,c.fecha_cita
from medico m, cita c
where m.medico_id = c.medico_id
and consultorio='C-593'
```

Mostrar el plan de ejecución:

```
SQL> set linesize 100
SQL> select plan_table_output
     from table(dbms_xplan.display('PLAN_TABLE','s1','TYPICAL'));
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
Plan hash value: 2487834737

-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |           |    17 |  1105 |    51   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                 |           |    17 |  1105 |    51   (0)| 00:00:01 |
|   2 |   NESTED LOOPS                |           |    17 |  1105 |    51   (0)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL          | CITA      |    17 |   510 |    34   (0)| 00:00:01 |
|*  4 |    INDEX UNIQUE SCAN          | MEDICO_PK |     1 |       |     0   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| MEDICO    |     1 |    35 |     1   (0)| 00:00:01 |

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("CONSULTORIO"='C-593')
   4 - access("M"."MEDICO_ID"="C"."MEDICO_ID")

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
   - this is an adaptive plan
```

A nivel básico un plan de ejecución típicamente está formado por los siguientes elementos (mostrados en la figura anterior).

Columna Id:

- Representa el identificador de la operación. Este valor NO indica el orden de ejecución. Se emplea únicamente para hacer referencia a cada operación.
- Observar los "*" que se muestran del lado izquierdo. Este carácter indica la existencia de información adicional para la operación. Observar en la imagen anterior, en la sección "Predicate Information" aparece el detalle de cada operación empleando su identificador.

Operation:

- Indica el tipo de operación que se va a realizar. Existen diversos tipos de operación, pero a nivel básico, esta columna incluye el "Access Path" o método de acceso a datos que se empleará para obtener datos.
- Observar que existe una indentación entre operaciones. La operación con la mayor indentación se ejecuta primero. En este ejemplo, las operaciones 3 y 4 se ejecutarán primero de forma independiente. El resultado de ambas sirve como entrada para ejecutar la siguiente operación, que en este caso es la operación 2. Observar que el resultado de la operación 2 junto con el resultado de la operación 5 se encuentran en el mismo nivel, por lo tanto, ambas operaciones serán empleadas para ejecutar la operación 1. Finalmente, el resultado de la operación 1 será empleado por la operación 0 para mostrar el resultado final.

Rows:

- Indica el número de registros estimados que obtendría cada una de las operaciones ejecutadas.

Bytes:

- Memoria estimada en bytes que se requiere para ejecutar cada una de las operaciones del plan

Cost:

- Porcentaje estimado de procesamiento requerido para ejecutar cada una de las operaciones del plan.

Time:

- Tiempo estimado de procesamiento de cada una de las operaciones del plan.

7.4.2. Métodos de acceso a datos (Access Paths)

Este concepto es fundamental para entender y generar planes de ejecución eficientes. Como se mencionó anteriormente, a nivel básico existen 2 tipos de métodos de acceso:

- Acceso a índices.
- Acceso a tablas.

7.4.2.1. Acceso a Índices

Para el caso de los índices, es importante recordar la información que contienen. Para efectos de un plan de ejecución, el índice puede ser visualizado como una tabla de datos con 2 columnas: `ROW_ID` y `etiqueta`. Por ejemplo, si la columna `email` de la tabla `cliente` está indexada, los datos que contiene el índice se verán así:

ROW_ID	etiqueta
01	juan@mail.com
02	paco@mail.com
03	hugo@mail.com
…	….

- Recordando, un ROW_ID representa un puntero a disco donde se encuentra almacenado cada registro de la tabla y representa el método de acceso más eficiente para recuperar un dato. Por simplicidad, se emplean los valores 01,02,03, pero en realidad los valores de un ROW_ID en el caso de Oracle son 18 caracteres. Para mayores detalles en cuanto al funcionamiento de índices B tree y ROW_IDs, revisar tema 3 del curso de BD, o en su defecto https://docs.oracle.com/cloud/latest/db112/CNCPT/indexiot.htm#CNCPT721
- El objetivo de leer un índice es la obtención de una lista de ROW_IDs que serán empleados para acceder a los datos de una tabla de forma eficiente.

7.4.2.2. Acceso a tablas.

En esta estrategia se realiza un acceso directo a los registros de una tabla. Generalmente un acceso a un índice requiere menos recursos ya que su estructura es de un tamaño mucho menor que el de una tabla. Sin embargo, en algunos casos un acceso a una tabla puede ser mejor opción que un acceso a disco.
La manera en la que la tabla organiza el almacenamiento de sus datos también influye en la selección del tipo de acceso:

- ***Heap Organized table (Default):*** Los registros no son almacenados en algún orden en particular. Cada registro se almacena en el primer espacio disponible dentro del segmento de datos.
- ***Index Organized table***: Los registros se ordena de acuerdo al valor de la PK.
- ***External table:*** Tabla de solo lectura, los metadatos de la tabla se almacenan en la BD mientras que los datos son almacenados de forma externa.

En las siguientes secciones se revisan estos 2 tipos de accesos a detalle.

7.4.3. Table Access Paths.

7.4.3.1. Table Access Full.

En general el optimizador selecciona esta estrategia en las siguientes situaciones:

- No existe un índice
- El predicado del query invoca funciones y no existe un índice basado en dicha función.
- Se lanza un `count(*)` , el índice existe, pero existen valores nulos en la columna indexada.
- El número de registros a obtener es grande (selectividad baja).
  - Es más eficiente hacer Table Access Full (leer cantidades grandes de datos pocas veces) que leer pocos datos con alta frecuencia.
- Las estadísticas están desactualizadas.

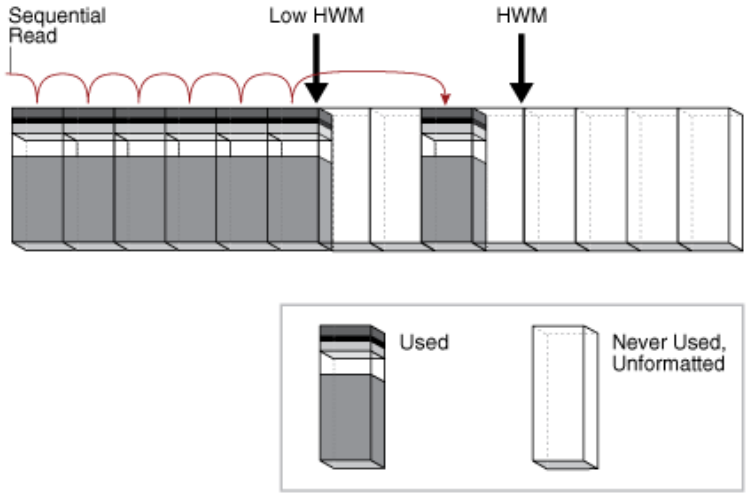
- Al inicio la tabla es pequeña, después crece. Dicho crecimiento permitiría el uso de un índice para mejorar el desempeño. Sin embargo, al no actualizar dicho crecimiento en las estadísticas, el optimizador no podrá determinar que el índice es más adecuado.

- Tablas pequeñas.
  - Si la tabla contiene menos de N bloques de datos donde N es un umbral definido por el parámetro
    `db_file_multiblock_read_count`

```
SQL> show parameter db_file_multiblock_read_count

NAME                                 TYPE        VALUE
------------------------------------ ----------- ---------
db_file_multiblock_read_count        integer     128
```

- La consulta usa un Hint para forzar un full table scan : `full(table alias)`

Para realizar un full table scan, el manejador leerá todos los bloques de datos que contiene datos (bloques con formato) hasta encontrar a la llamada "Marca de Agua". Bloques sin formato son saltados. Cada bloque se lee una sola vez.



Ejemplo:

```
explain plan for
select * from paciente;

select plan_table_output
from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 588422155

--------------------------------------------------------------------------------
| Id  | Operation         | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |          | 15302 |    26M|   477   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| PACIENTE | 15302 |    26M|   477   (0)| 00:00:01 |
--------------------------------------------------------------------------------
```

7.4.3.2. Table Access by rowid

- La forma más rápida de acceder a un registro es a través de su ROW_ID.
- Generalmente se emplea este método de acceso posterior al escaneo de un índice.
- Si el índice contiene todas las columnas requeridas se omite realizar esta operación.

Ejemplo:

```
explain plan for
select nombre
from paciente
where paciente_id = 3;

select plan_table_output
from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------
Plan hash value: 3341724136

--------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |     1 |    35 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| PACIENTE    |     1 |    35 |     2   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN          | PACIENTE_PK |     1 |       |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------
---------------------------------------------------

   2 - access("PACIENTE_ID"=3)
```

Ejemplo:

```
explain plan for
select nombre
from paciente
where paciente_id >18500;

select plan_table_output
from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------------
Plan hash value: 2503382986

--------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |             |     1 |    35 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| PACIENTE    |     1 |    35 |     1   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | PACIENTE_PK |     1 |       |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------------
--------------------------------------------------

   2 - access("PACIENTE_ID">18500)
```

- En este ejemplo, la lectura del índice produce pocos ROW_IDs, pero el manejador lee por bloque para minimizar el número de accesos a un mismo bloque. Esto se indica con la palabra BATCHED.

7.4.3.3. Sample table scan

- En esta estrategia se obtiene una muestra de registros de una tabla o de una sentencia `SELECT` compleja que contiene diversos joins y "Vistas".
- La muestra deseada se expresa en porcentaje [0.000001,100)
- Obtiene un porcentaje aproximado del contenido de una tabla. El porcentaje se aplica a los bloques, no a los registros.

Ejemplo:

```
explain plan for
select *
from cita sample block(10);

select plan_table_output
from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 1426444112

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |   208 | 16848 |     5   (0)| 00:00:01 |
|   1 |  TABLE ACCESS SAMPLE| CITA |   208 | 16848 |     5   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

7.4.3.4. In Memory table scan

- Obtiene registros de una tabla con atributos almacenados en memoria de forma Columnar: *IM Column Store*.
- IM Column Store es un área de memoria ubicada en la SGA que almacena copias de tablas en un formato columnar especial que permite agilizar el escaneo de datos.

Ejemplo:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

SQL_ID  2mb4h57x8pabw, child number 0
-------------------------------------
select * from oe.product_information where list_price > 10 order byproduct_id

Plan hash value: 2256295385
-----------------------------------------------------------------------------------------------
|Id| Operation                   | Name                |Rows|Bytes |TempSpc|Cost(%CPU)|Time|
-----------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT            |                     |    |      |       |21 (100)|        |
| 1|  SORT ORDER BY              |                     | 285| 62415|82000|21   (5)|00:00:01|
|*2|   TABLE ACCESS INMEMORY FULL| PRODUCT_INFORMATION | 285| 62415|       | 5   (0)|00:00:01|
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - inmemory("LIST_PRICE">10)
       filter("LIST_PRICE">10)
```

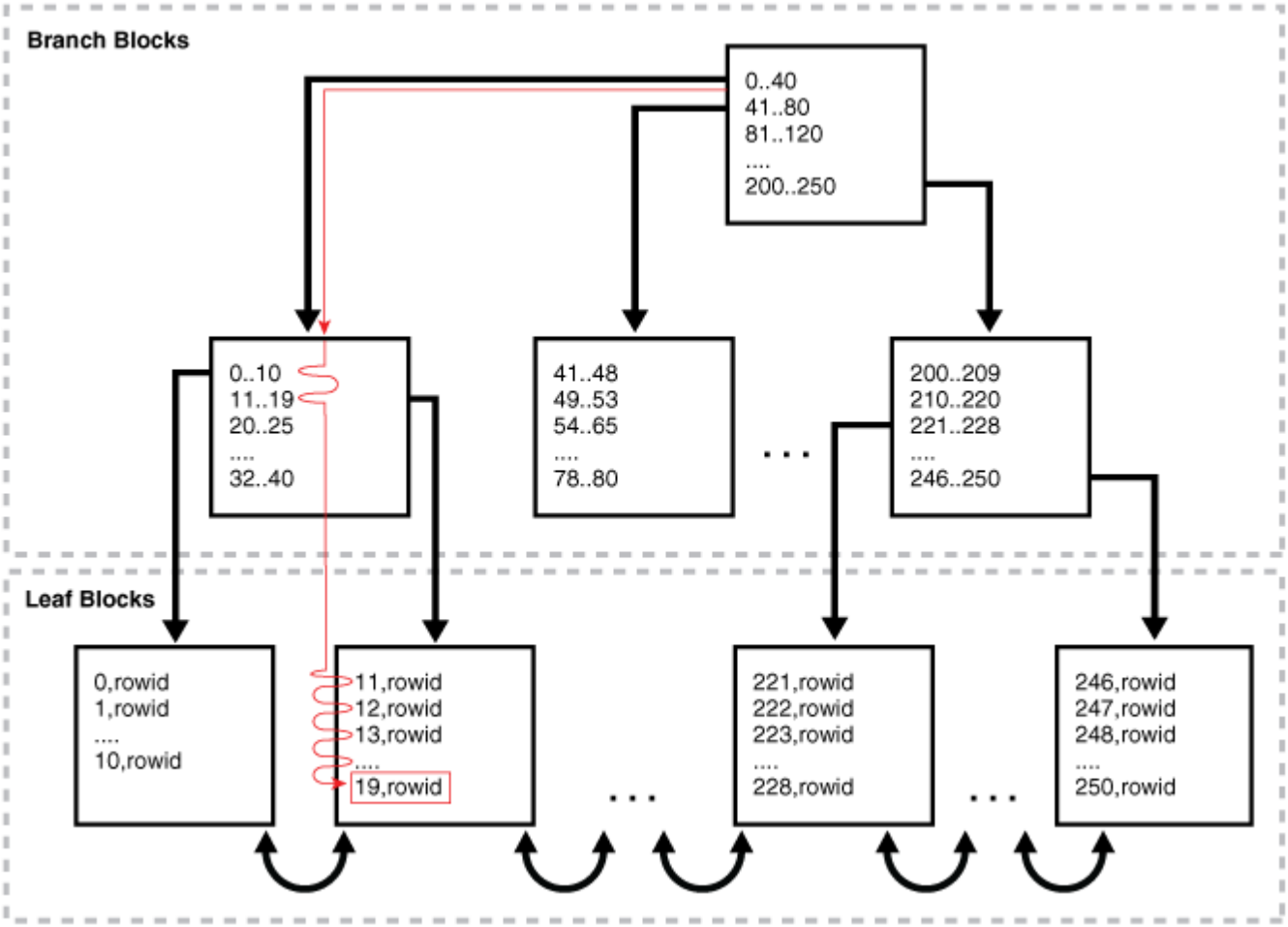
7.4.4. B-Tree Index Access Paths.

Para efectos del curso, solo se consideran los índices tipo B-Tree.

- Index Unique Scans
- Index Range Scans
- Index Full Scans
- Index Fast Full Scans
- Index Skip Scans
- Index Join Scans

7.4.4.1. Index Unique scan.

- Regresa a lo más un ROW_ID al realizar el escaneo del índice.
- Empleado cuando se incluye un operador de igualdad en una columna indexada
- El índice se recorre hasta encontrar la primera ocurrencia debido a que se trata de un Índice de tipo UNIQUE y por lo tanto, no es necesario continuar con el escaneo.
- En la siguiente figura se muestra la forma en la que se realiza el escaneo para encontrar el ROW_ID cuyo valor de la columna es 19.



Ejemplo:

```
create unique index paciente_email_idx
on paciente(email);

explain plan for
select email
from paciente
where email =' jonh@mail.com';

select plan_table_output
from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
--------------------------------------------------
Plan hash value: 839293805

--------------------------------------------------------------------------------
| Id  | Operation         | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                    |     1 |   102 |     1   (0)| 00:00:01 |
|*  1 |  INDEX UNIQUE SCAN| PACIENTE_EMAIL_IDX |     1 |   102 |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
--------------------------------------------------

   1 - access("EMAIL"=' jonh@mail.com')
```

Ejemplo:

Observar que el siguiente ejemplo no se hace uso del índice. ¿Por qué razón?

```
explain plan for
select email
from paciente;


select plan_table_output
from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
--------------------------------------------------
Plan hash value: 588422155

--------------------------------------------------------------------------------
| Id  | Operation         | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |          | 16584 |  1651K|   477   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| PACIENTE | 16584 |  1651K|   477   (0)| 00:00:01 |
--------------------------------------------------------------------------------
```

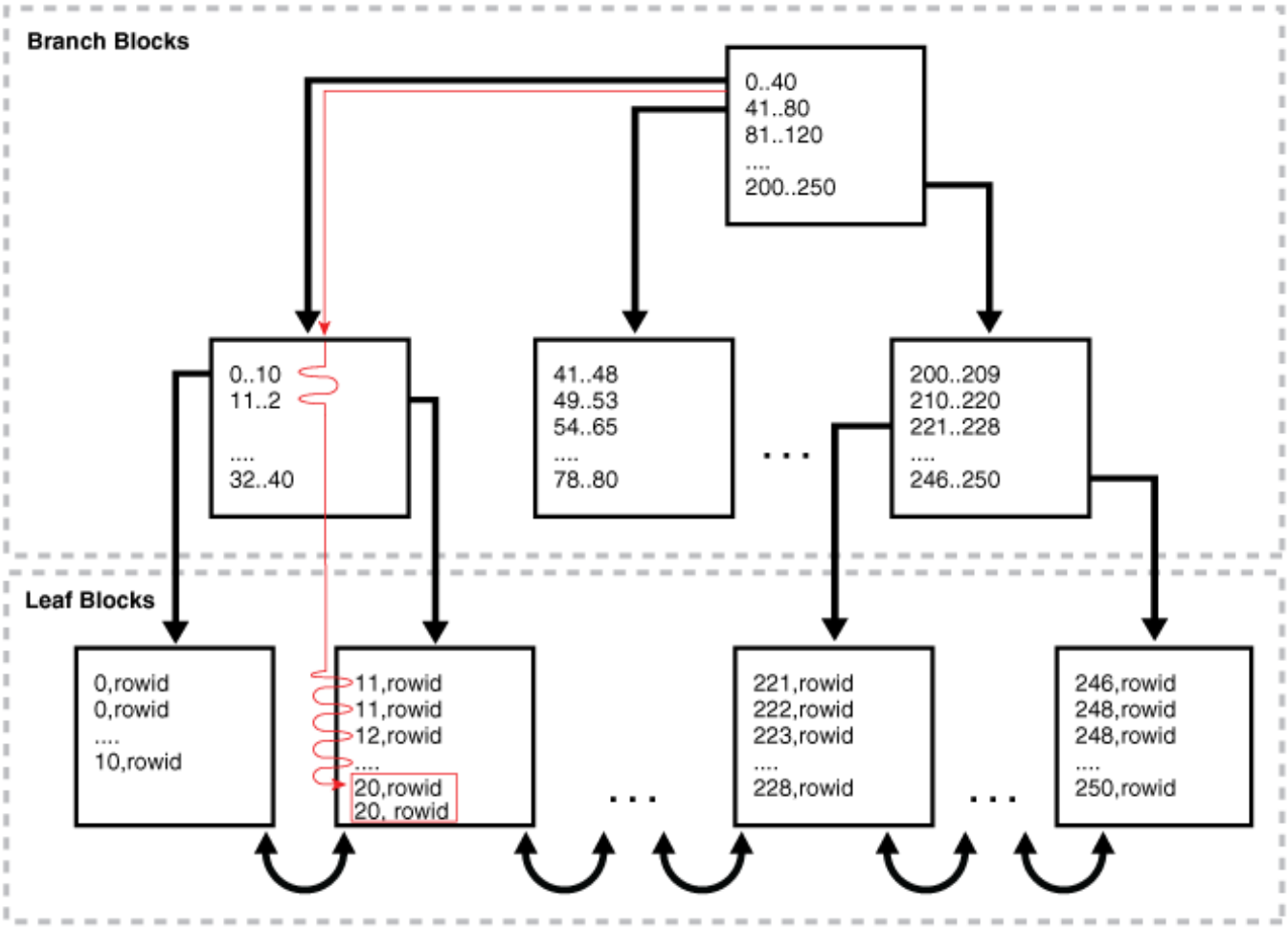
Respuesta:

La tabla contiene valores nulos.

7.4.4.2. Index Range scans.

- Realiza un escaneo de valores ordenados a un índice B-tree
- Por default, los valores en un índice se almacenan de forma ascendente y se escanean en el mismo orden.
- Se emplea en casos como:
  - Predicados en la columna indexada con operadores >=, <=, etc.
  - El índice es `non unique`. Significa que pueden obtenerse más de un valor.

En la siguiente imagen se muestra el funcionamiento de un escaneo del índice por rango. Observar que el escaneo puede realizarse de forma horizontal empleando los punteros horizontales entre los nodos hojas. Por ejemplo, se escanean todos los ROW_IDs entre 20 y 40.



Ejemplo:

```
explain plan for
select email
from paciente
where email  like 'bob@%'
order by email desc;

select plan_table_output
from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------------------
--------------------------------------------------
Plan hash value: 3471813995

-----------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                     |     1 |   102 |     1   (0)| 00:00:01 |
|*  1 |  INDEX RANGE SCAN DESCENDING| PACIENTE_EMAIL_IDX |     1 |   102 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------------------
---------------------------------------------------

   1 - access("EMAIL" LIKE 'bob@%')
       filter("EMAIL" LIKE 'bob@%')
```

Ejemplo:

- ¿Por qué razón no se emplea índice?

```
create index paciente_nombre_idx
on paciente(nombre);

explain plan for
select nombre
from paciente
order by paciente_id desc;

select plan_table_output
from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------------------
--------------------------------------------------
Plan hash value: 375339519

-----------------------------------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          | 16584 |   566K|       |   635   (1)| 00:00:01 |
|   1 |  SORT ORDER BY     |          | 16584 |   566K|   728K|   635   (1)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| PACIENTE | 16584 |   566K|       |   477   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

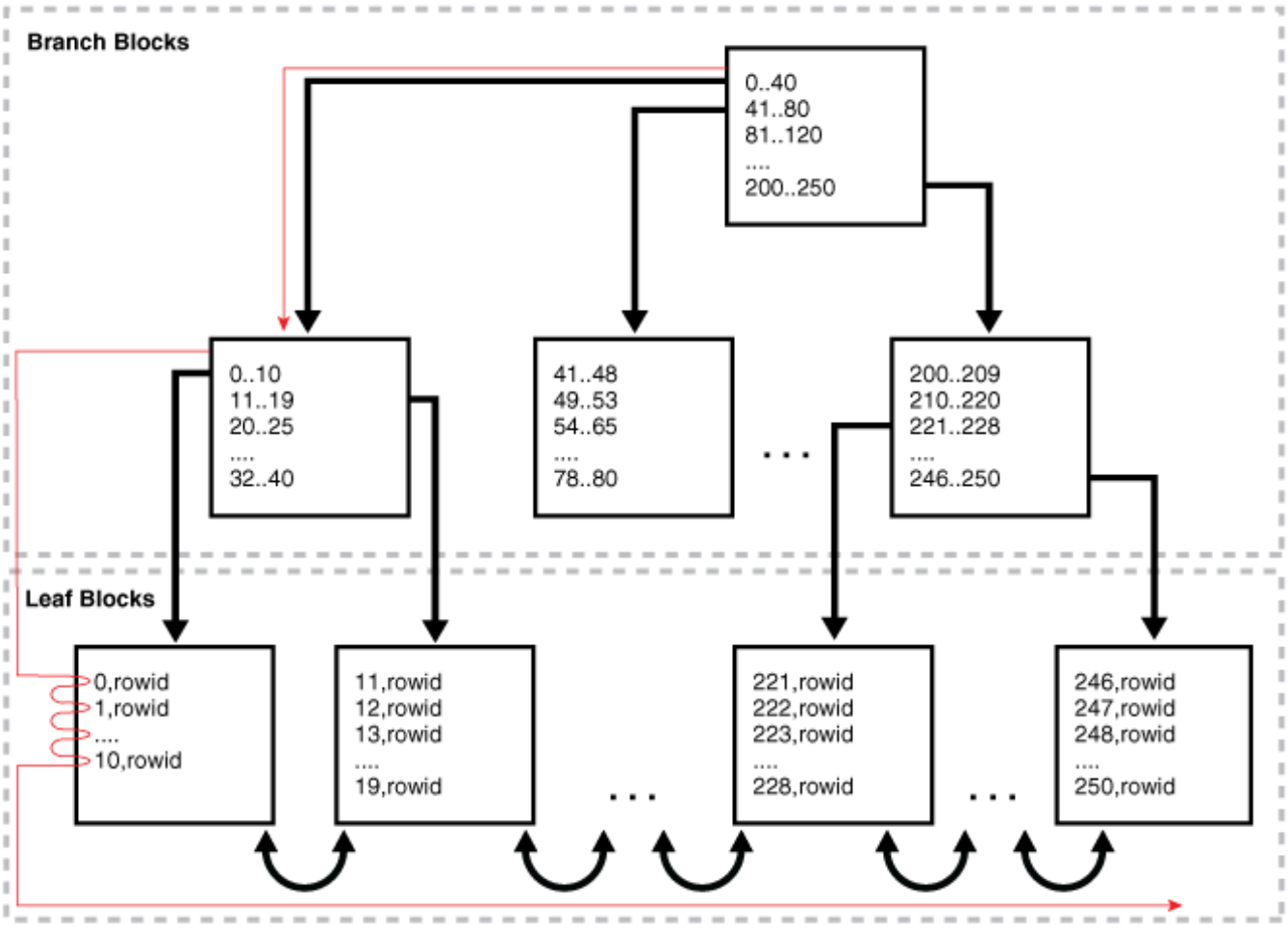
Respuesta:
La condición de ordenamiento involucra a un campo diferente al de la condición.

7.4.4.3. Index Full scan

- Se realiza la lectura completa del índice de forma ordenada.
- Empleado en casos donde el predicado hace referencia a alguna de las columnas de un índice compuesto, dicha columna no debe ser la primera columna especificada en el índice.
- No existe predicado, pero existen las siguientes condiciones:
    - Todas las columnas solicitadas por la consulta están en el índice

- Al menos una de las columnas indexadas está declarada como NOT NULL

- La sentencia incluye ORDER BY sobre una columna indexada.



Ejemplo:

```
explain plan for
select nombre
from paciente
order by nombre;

select plan_table_output
from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
--------------------------------------------------
Plan hash value: 898347700

--------------------------------------------------------------------------------
| Id  | Operation        | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |                     | 16584 |   356K|    45   (0)| 00:00:01 |
|   1 |  INDEX FULL SCAN | PACIENTE_NOMBRE_IDX | 16584 |   356K|    45   (0)| 00:00:01 |
--------------------------------------------------------------------------------
```

7.4.4.4. Index fast full scan

- En este caso se leen todos los bloques del índice de forma desordenada, tal cual se almacenaron en disco.
- Este tipo de índice se emplea en consultas donde se seleccionan únicamente columnas contenidas en el índice.

Físicamente, los bloques que integran al índice no necesariamente están ordenados dentro del segmento asociado al índice

Ejemplo:

```
explain plan for
select email
from paciente
where email is not null;

select plan_table_output
from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------------
----------------------------------------------------
Plan hash value: 3687434030

-------------------------------------------------------------------------------------------------
| Id  | Operation            | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                    | 14438 |  1438K|    20   (0)| 00:00:01 |
|*  1 |  INDEX FAST FULL SCAN| PACIENTE_EMAIL_IDX | 14438 |  1438K|    20   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Ejemplo:

```
explain plan for
select count(*)
from paciente;

select plan_table_output
from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
----------------------------------------------
Plan hash value: 2895057998

-----------------------------------------------------------------------------
| Id  | Operation             | Name        | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |             |     1 |     9   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE       |             |     1 |            |          |
|   2 |   INDEX FAST FULL SCAN| PACIENTE_PK | 15000 |     9   (0)| 00:00:01 |
-----------------------------------------------------------------------------
```

7.4.4.5. Index skip Scan

- Empleado en casos con índices compuestos.
- La primera columna tiene muy pocos valores diferentes.

- La segunda columna tiene una gran cantidad de valores distintos.
- La consulta no incluye en el predicado a la primera columna.

Ejemplo:

Suponer un índice compuesto con las columnas de la tabla paciente genero e email:

```
. . .
F,Wolf@company.example.com,rowid
F,Wolsey@company.example.com,rowid
F,Wood@company.example.com,rowid
F,Woodman@company.example.com,rowid
F,Yang@company.example.com,rowid
F,Zimmerman@company.example.com,rowid
M,Abbassi@company.example.com,rowid
M,Abbey@company.example.com,rowid
. . .
```

El manejador procesa la consulta creando 2 sub-índices de manera lógica. Un índice para procesar los valores 'F' y otro para los valores 'M'

```
( select *
  from   paciente
  where  genero = 'F'
  and    email = 'abbey@company.com' )
union all
( select *
  from   paciente
  where  genero = 'M'
  and    email = 'abbey@company.com' )
```

Plan de ejecución:

```
create index paciente_email_genero_idx
on paciente(genero,email);

explain plan for
select *
from paciente
where email='smith@mail.com';

select plan_table_output
from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------
Plan hash value: 3560863078

------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name                       | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                            |     1 |   773 |     4   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| PACIENTE                   |     1 |   773 |     4   (0)| 00:00:01 |
|*  2 |   INDEX SKIP SCAN                   | PACIENTE_EMAIL_GENERO_IDX  |     1 |       |     3   (0)| 00:00:01 |
------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------
------------------------------------------------

   2 - access("EMAIL"='smith@mail.com')
       filter("EMAIL"='smith@mail.com')
```

7.4.4.6. Index Join Scan

- Significa realizar un Hash Join entre los resultados obtenidos al leer N índices.
- Empleado generalmente cuando las columnas solicitadas por las columnas están contenidas en los índices, no se requiere hacer acceso a los registros de las tablas. Es decir, se hace join entre índices.

Ejemplo:

```
create index paciente_nombre_idx
on paciente(nombre);

create index paciente_ap_paterno_idx
on paciente(ap_paterno);

explain plan for
select nombre,ap_paterno
from paciente
where nombre like 'A%';

select plan_table_output
from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------
------------------------------------------------
Plan hash value: 3337046678

------------------------------------------------------------------------------------------------------------
| Id  | Operation              | Name                    | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                         |   422 |  6330 |    42   (0)| 00:00:01 |
|*  1 |  VIEW                  | index$_join$_001        |   422 |  6330 |    42   (0)| 00:00:01 |
|*  2 |   HASH JOIN            |                         |       |       |            |          |
|*  3 |    INDEX RANGE SCAN    | PACIENTE_NOMBRE_IDX     |   422 |  6330 |     3   (0)| 00:00:01 |
|   4 |    INDEX FAST FULL SCAN| PACIENTE_AP_PATERNO_IDX |   422 |  6330 |    49   (0)| 00:00:01 |
------------------------------------------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------
------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("NOMBRE" LIKE 'A%')
   2 - access(ROWID=ROWID)
   3 - access("NOMBRE" LIKE 'A%')
```

7.4.5. Joins

- Un Join combina la salida de 2 fuentes de datos llamadas “Row Sources”.
- Obtiene como salida otro “Row Source” llamado “Data Set”.
- La condición del Join permite realizar la comparación de las 2 fuentes de datos. De no especificar dicha condición se genera producto cartesiano estableciendo una correspondencia de cada uno de los registros de un Row Source con cada uno de los registros del otro.
- Cuando existen múltiples tablas en una consulta, el optimizador debe tomar en consideración diversos aspectos para determinar las operaciones más eficientes para cada pareja de “Row sources”:
    - Access Paths (métodos de acceso).
    - Join Methods: Para realizar la comparación o ejecución del join el optimizador aplica alguno de los métodos disponibles. Es decir, en esta etapa se define el método a emplear para obtener el resultado de un join entre 2 “Row sources”:
        - Nested loop
        - Sort Merge
        - Hash Join
        - Cartesian Join.
    - Join Types
        - Outer
        - Inner
    - Join order: Empleado especialmente con sentencias que contienen más de 2 tablas. El orden en el que se realizan las operaciones de Joins es importante.

Con todas estas variantes el optimizador genera diversos planes de ejecución seleccionando aquel que tenga el menor costo.
A nivel general, el optimizador considera los siguientes aspectos:

- Revisa si el resultado de un Join contiene a lo más 1 registro.
    - Para ello revisa la existencia de constraints `unique` Y `primary key`.
    - Si esto ocurre, el join se ejecuta al principio.

Antes de iniciar con la revisión de los distintos métodos que existen para implementar una operación `join` es importante considerar los siguientes 2 conceptos:

- Cada método define 2 conceptos principales:
    - Una de las fuentes de datos se selecciona como “Driving table” o “Outer table”.
    - La otra fuente de datos se le conoce como “Driven table” o “Inner table”.

Lo anterior es análogo a un ciclo for anidado:

```
for (int i =0 :i<n; i++) {     => driving table
     for(int j =0;j<k;j++) {   => diven table
          . . . .
     }
}
```

7.4.5.1. Nested Loops Joins

- En este tipo de Join, para cada registro de la tabla Outer se buscan correspondencias con N registros de la tabla Inner.
- Para cada registro de la tabla Outer que cumpla con el predicado de la operación `join`, la BD obtiene todos los registros que satisfacen al predicado.
- Si existe un índice que permita recuperar los datos de la tabla Inner, el manejador puede emplearlo a través de `row_ids` (Aplica solo para los datos de la tabla Inner).
- En un plan de ejecución un NESTED LOOP aparece de la siguiente manera:

```
NESTED LOOP
    Outer_table
    Inner_table
```

¿En qué situaciones es conveniente el uso de un Nested Join?

- Las fuentes de datos tienen pocos registros.
- La condición del Join permite acceder a los datos en la Inner table de forma eficiente. Por ejemplo: Suponer una operación `join` entre `diagnostio` y `cita`, su predicado es `d.diagnostico_id = c.diagnostico_id`.
- `diagnostico` será seleccionada como outer table al detectar una menor cantidad de registros con respecto a `cita`.
- `cita` podría ser seleccionada como la tabla inner. Sin embargo, ¿Qué sucedería si la llave foránea en cita `c.diagnostico_id` está indexada?
- Si esto ocurre, el optimizador puede seleccionar al índice como tabla inner en lugar de usar a cita.
- Esto es factible ya que para verificar correspondencias entre las tablas basta con tener los valores de `diagnostico_id` de cada conjunto de datos con sus respectivos row_ids.

- De lo anterior, es importante mencionar que la inner table no siempre corresponde con una tabla existente en la BD. Tanto la outer como la inner table pueden ser índices.
- En general este método es adecuado para Joins con pocos registros en las tablas fuente en el que las columnas que participan están indexadas.
- Existe un umbral interno que puede influenciar al optimizador el uso de un Nested Loop. Si el número de registros en la tabla Outer no excede el umbral, el optimizador puede considerar Nested Loop. De lo contrario, Hash Join podría ser la alternativa.

Ejemplo 1:

Obtener la fecha de las citas cuya clave de diagnóstico inicie con `A85`. Suponer las siguientes condiciones:

- Existencia del siguiente índice:

```
create index cita_diagnostico_id_ix on cita(diagnostico_id);
```

- El DBMS decide usar la técnica de Nested Loop haciendo uso de los índices a medida de lo posible.
- Para efectos del ejemplo, considerar las siguientes muestras de datos:

diagnostico			
Row_id	**d.diagnostico_id**	**d.clave**	**Nombre**
01	409	A90	Diarrea
02	410	A91	Migraña
03	411	A851	Obesidad
04	412	A857	Leucemia
05	413	A854	Depresión
06	414	A852	Estrés
07	415	A853	Estreñimiento

cita				
c.row_id	**c.fecha_cita**	**cita_id**	**medico_id**	**diagnostico_id**
0001	10/05/2013	1	45	411
0004	23/04/2015	2	32	412
0002	14/08/2003	3	23	412
0003	20/02/1988	4	125	411
0005	30/09/2001	5	4	413
0008	22/07/2006	6	9	304
0009	10/11/2004	7	13	396
0006	14/14/2001	8	23	987

A. Generar la sentencia SQL que genere el plan de ejecución
B. Determine el método de acceso y la fuente de datos que producirá a la Outer table
C. Considerando la muestra de datos anterior, generar la tabla de datos que representa a la tabla Outer.
D. Determinar la fuente de datos que producirá a la tabla inner.
E. Considerando la muestra de datos anterior, generar la tabla de datos que representa a la tabla inner.
F. Determinar la expresión booleana que se aplicará a cada registro de la tabla inner, así como el método de acceso que se empleará sobre la tabla inner para recuperar los datos requeridos.
G. Determinar el resultado del Nested loop.
H. Determinar la siguiente operación del plan de ejecución para mostrar el resultado final de la consulta.

Solución:

A. Sentencia SQL que genera el plan de ejecución

```
explain plan for
select d.diagnostico_id, c.fecha_cita
from cita c
```

```
join diagnostico d on c.diagnostico_id = d.diagnostico_id
where d.clave like 'A85%';

select plan_table_output from table(dbms_xplan.display);
```

B. Tabla outer.

- El optimizador trata de aplicar posibles operaciones de selección (σ_p) para reducir las cardinalidades de las tablas:
  - Para la tabla `diagnostico` existe el predicado `clave like 'A85%'`. Debido a que el campo clave no está indexado, la única opción es realizar un `table access full`. De la muestra de datos se obtendrán 5 registros que serán considerados para realizar el `join`.
  - Para la tabla `cita` no existe algún predicado por lo que los 8 registros participarán para realizar el `join`.
- Para ejecutar el nested loop se requieren los siguientes datos:
  - `d.diagnostico_id` el cual ya se tiene disponible ya que se hizo un table Access full en el punto anterior.
  - `c.diagnostico_id` La forma más rápida de acceder a este campo es haciendo uso del índice `cita_diagnostico_id_ix`
- De lo anterior, el optimizador elige a la tabla `diagnostico` como tabla outer ya que tiene solo 5 registros.

C. Tabla outer con datos.

De la tabla `diagnostico` solo se selecciona la columna `diagnostico_id` con los 5 identificadores que se requieren para ejecutar el join. Notar que no se necesita ninguna otra columna.

d.diagnostico_id
411
412
413
414
415

D. Tabla inner.

- La tabla inner estará representada por el índice `cita_diagnostico_id_ix`, ya que representa la forma más eficiente de acceder a los valores de `c.diagnostico_id`
- Notar que tanto la tabla inner como la outer puede estar representada por índices, no solo por tablas.

E. Tabla inner con datos:

cita_diagnostico_ix	
c.diagnostico_id	**row_id**
411	0001
412	0004
412	0002

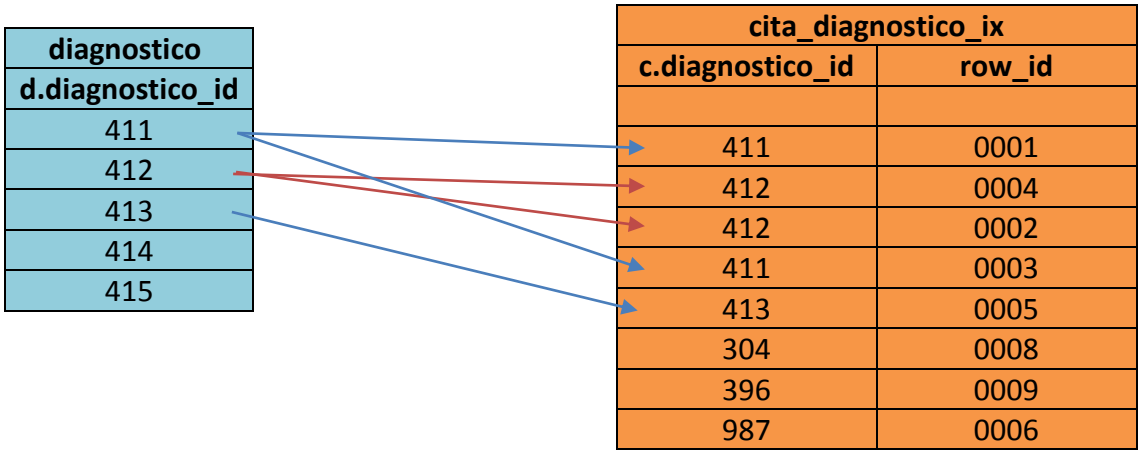
411	0003
413	0005
304	0008
396	0009
987	0006

F. Expresión booleana:
`d.diagnostico_id = c.diagnostico_id`

Por cada registro de la tabla outer (`d.diagnostico_id`) se hará un *index range scan* al índice `cita_diagnostico_ix` para obtener una lista de row_ids, es decir:

- Iteración 1: index range scan para obtener todos los row_ids donde `d.diagnostico_id` = 411
  - L ={0001,0003}
- Iteración 2: index range scan para obtener todos los row_ids donde `d.diagnostico_id` = 412
  - L ={0002,0004}
- Iteración 3: index range scan para obtener todos los row_ids donde `d.diagnostico_id` = 413
  - L ={0005}
- En las iteraciones restantes no se obtienen correspondencias.

Se requiere un index range scan ya que por cada valor de `d.diagnostico_id` se pueden obtener varias correspondencias en el índice:



G. Resultado del nested loop.

diagnostico	cita_diagnostico_ix
d.diagnostico_id	row_id
411	0001
411	0003
412	0002
412	0004
413	0005

H. Finalmente, en esta operación, empleando los ROW_IDs de la tabla anterior, se realizará un table Access by index row id para recuperar el campo `fecha_cita` en la tabla `cita`.

diagnostico	cita_diagnostico_ix
d.diagnostico_id	row_id
411	0001
411	0004
412	0002
412	0003
413	0005

cita	
c.row_id	c.fecha_cita
0001	10/05/2013
0004	23/04/2015
0002	14/08/2003
0003	20/02/1988
0005	30/09/2001

El resultado final será:

diagnostico	cita
d.diagnostico_id	c.fecha_cita
411	10/05/2013
411	23/04/2015
412	14/08/2003
412	20/02/1988
413	30/09/2001

En Oracle, este último paso se representa como un segundo nested loop:

```
Plan hash value: 3773197897

-------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                         |     3 |   282 |    37   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |                         |     3 |   282 |    37   (0)| 00:00:01 |
|   2 |   NESTED LOOPS               |                         |     3 |   282 |    37   (0)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL         | DIAGNOSTICO             |     1 |    25 |    34   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | CITA_DIAGNOSTICO_ID_IX  |     2 |       |     1   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| CITA                    |     2 |   138 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("D"."CLAVE" LIKE 'A85%')
   4 - access("C"."DIAGNOSTICO_ID"="D"."DIAGNOSTICO_ID")
```

- Observar que el segundo nested loop con id = 1, la tabla outer corresponde con el resultado del primer nested loop.
- Por cada ROW_ID contenido en la tabla outer, se realizará la operación `table_access_by_index_row_id` y se recupera el valor de la columna `fecha_cita`, empleando a la tabla `cita` como tabla inner (paso 5).

Ejemplo 2:

- Considerar nuevamente la consulta anterior.
- ¿Qué efectos tendrá el plan de ejecución si se elimina el índice `cita_diagnostico_id_ix` y se agrega la condición `c.medico_id = 2286`?

```
drop index cita_diagnostico_id_ix;

explain plan for
select d.diagnostico_id, c.fecha_cita
from cita c
join diagnostico d
on c.diagnostico_id = d.diagnostico_id
where d.clave like 'A85%'
and c.medico_id = 2286;

select plan_table_output from table(dbms_xplan.display);
```

```
Plan hash value: 4020497077

-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |     1 |    64 |    37   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |                |     1 |    64 |    37   (0)| 00:00:01 |
|   2 |   NESTED LOOPS               |                |     7 |    64 |    37   (0)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL         | CITA           |     7 |   273 |    30   (0)| 00:00:01 |
|*  4 |    INDEX UNIQUE SCAN         | DIAGNOSTICO_PK |     1 |       |     0   (0)| 00:00:01 |
|*  5 |   TABLE ACCESS BY INDEX ROWID| DIAGNOSTICO    |     1 |    25 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("C"."MEDICO_ID"=2286)
   4 - access("C"."DIAGNOSTICO_ID"="D"."DIAGNOSTICO_ID")
   5 - filter("D"."CLAVE" LIKE 'A85%')
```

- En este ejemplo solo se cuenta con los índices de las PKs de ambas tablas. El manejador hará lo posible por emplearlos.
- Para ejecutar el predicado `c.medico_id = 2286` la única opción es Table Access Full. Se estiman 7 registros.
- Debido a que se realizó un table access full, se cuenta con todos los valores de las columnas de la tabla `cita`. Uno de estos atributos es `c.diagnostico_id` cuyos valores corresponden con la PK de diagnóstico y ahí si existe un índice.
- Para cada uno de estos 7 valores, el manejador podría acceder al índice de la PK en la tabla `diagnostico` evitando así otro table access full.
- Para poder hacer esto, el manejador deberá elegir como tabla outer a `cita` ya que, para cada uno de los 7 valores, se hará un *unique index scan* al índice `diagnostico_pk`.
- Observar que en este caso no se respetó la regla de seleccionar a la tabla con el menor número de registros como tabla outer.

Este proceso se muestra a detalle con la siguiente muestra de datos:

A. **id 3**: El optimizador elige a `cita` como la tabla outer. La tabla outer contendrá `fecha_cita` y `diagnostico_id` para poder ejecutar el Nested Loop. Solo se muestran 4 registros por simplicidad. Se agrega el campo `fecha_cita` ya que fue solicitado en la consulta y se encuentra disponible al haber realizado el table access full.

cita	
c.fecha_cita	**c.diagnostico_id**
10/05/2013	3
23/04/2015	4
14/08/2003	5
20/02/1988	6

B. **id 4:** La tabla Inner está representada por el índice `diagnostico_pk`.

- La tabla `cita` fue seleccionada como tabla outer.
- Al ejecutar el Nested Loop, para cada valor de `cita.diagnostico_id` se buscarán correspondencias con la tabla `diagnostico`. El campo `diagnostico_id` representa la PK de `diagnostico` y tiene un índice unique asociado. El manejador detecta esta condición y decide emplear el índice `diagnostico_pk` empleando un `unique_index_scan`.

cita	
c.Fecha_cita	**c.diagnostico_id**
10/05/2013	3
23/04/2015	4
14/08/2003	5
20/02/1988	6

diagnostico_pk	
diagnostico_id	**Row_id**
3	0003
4	0004
5	0005
6	0006

C. **Id 2**: El resultado del Nested Loop es:

cita		diagnostico_pk
c.Fecha_cita	**c.diagnostico_id**	**d.Row_id**
10/05/2013	3	0003
23/04/2015	4	0004
14/08/2003	5	0005
20/02/1988	6	0006

D. **Id 1**: En el siguiente Nested Loop, la tabla outer es representada por la tabla anterior.

E. **Id 5**: La tabla inner es representada por `diagnostico`. Por cada registro de la tabla outer, se emplea el `row_id` para localizar a los registros de `diagnostico` empleando la operación table access by index row id. En este mismo paso, al recuperar a cada registro se le aplica el filtro `d.clave like 'A85%'`

cita		diagnostico_pk
c.Fecha_cita	**c.diagnostico_id**	**d.Row_id**
10/05/2013	3	0003
23/04/2015	4	0004
14/08/2003	5	0005
20/02/1988	6	0006

diagnostico	
d.Row_id	**d.Clave**
0003	A857
0004	A868
0005	A859
0006	A867

F. Finalmente, el resultado de la consulta considerando esta muestra ficticia será:

cita	diagnostico
c.Fecha_cita	d.diagnostico_id
10/05/2013	3
20/02/1988	6

Haber quitado el índice `cita_diagnostico_id_ix` provocó el cambio de elección de la tabla outer.

7.4.5.2. Hash Joins

- Se emplean para realizar Joins con conjuntos grandes de datos.
- La condición del JOIN es la igualdad (equi-join).
- El optimizador selecciona al Source Row de menor tamaño para construir un ***hash table***.
- La BD hace uso del HashTable para localizar las correspondencias con la otra tabla.
- Esta técnica resulta eficiente siempre y cuando el Row Source menor pueda ser cargado completamente en memoria.
- Si los datos no caben en memoria, se aplica un "particionamiento" del row source, incrementan las operaciones I/O especialmente en el tablespace Temporal. Parte del hashTable se almacena en memoria y la otra parte en disco.

Algoritmo:

1. Típicamente la BD hace un full scan del row source menor llamada "*Build Table*", aplica una función hash a cada uno de los valores de la columna empleada como condición del join. Se construye el Hash Table y se guarda en la PGA (se guardan todas las columnas de la tabla en el hashTable).

```
for small_table_row in (select * from small_table)
loop
  slot_number := hash(small_table_row.join_key);
  insert_hash_table(slot_number,small_table_row);
end loop;
```

2. La tabla que no fue seleccionada para crear el hash table se le conoce como "*probe table*". Básicamente, para cada valor de la columna que participa en la condición del Join se le aplica la función Hash. El resultado es empleado para localizar las correspondencias en el HashTable.

Ejemplo:

Obtener el nombre del médico y el nombre de su especialidad para todos los registros existentes en la base de datos.

A. Generar la sentencia SQL que genera el plan de ejecución
B. Describir el algoritmo considerando los siguientes datos asumiendo que el manejador selecciona la técnica de la tabla hash.

ESPECIALIDAD

Especialidad_id	nombre	anios	requsito
1	Anatomía Patológica	4	Medicina general
2	Anestesiología	3	Medicina general

3	Cardiología	1	Medicina general

MEDICO

medico_id	nombre	especialidad_id
1	Ale	3
2	Amalia	3
3	Aurora	2
4	Aldo	2
5	Felipe	3
6	Laura	2

A. Sentencia SQL que genera el plan de ejecución.

```
explain plan for
select m.nombre,e.nombre
from especialidad e
join medico m
on m.especialidad_id = e.especialidad_id;

select plan_table_output
from table(dbms_xplan.display);
```

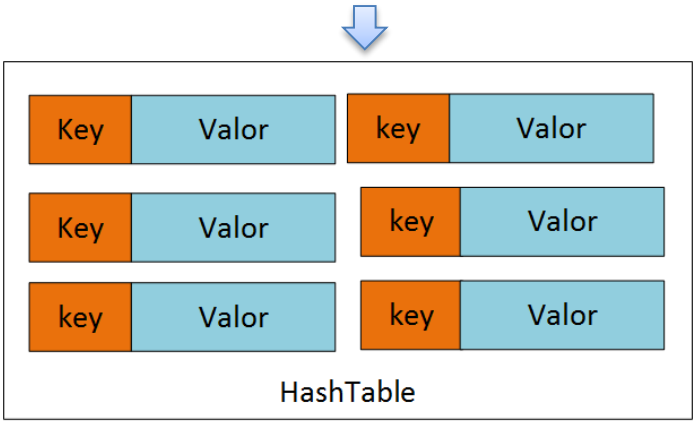
Predicado del join

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
--------------------------------------------------
Plan hash value: 2505893939

-------------------------------------------------------------------------------
| Id  | Operation          | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |              |  5000 |   166K|   241   (0)| 00:00:01 |
|*  1 |  HASH JOIN         |              |  5000 |   166K|   241   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| ESPECIALIDAD |    53 |  1272 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| MEDICO       |  5000 | 50000 |   238   (0)| 00:00:01 |
-------------------------------------------------------------------------------


PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
--------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("M"."ESPECIALIDAD_ID"="E"."ESPECIALIDAD_ID")
```

B. Algoritmo:

- De la imagen se puede observar que efectivamente se selecciona a la tabla `especialidad` como ***build table***. Con ella se construirá la HashTable. Su construcción se puede representar de la siguiente manera. Notar que el "valor" de cada pareja contiene al registro completo.

key	value			
ora_hash(especialidad_id)	Especialidad_id	nombre	anios	requsito
2342552567	1	Anatomía Patológica	4	Medicina general
2064090006	2	Anestesiología	3	Medicina general
2706503459	3	Cardiología	1	Medicina general



Para generar hash codes se puede emplear la función `ora_hash`. Notar que el parámetro que se le pasa a la función `ora_hash` corresponde con los valores de la columna que se usa en el predicado de la operación join, `especialidad_id`.

```
col nombre format A30
col requisito format A30

select  especialidad_id,nombre,anios,requisito,ora_hash(especialidad_id)
from especialidad
where rownum <5;
```

Al ejecutar la sentencia se obtiene lo siguiente:

```
ESPECIALIDAD_ID NOMBRE                         ANIOS REQUISITO                      ORA_HASH(ESPECIALIDAD_ID)
--------------- ------------------------------ ----- ------------------------------ -------------------------
              1 Anatomia Patologica                4 Medicina General                           2342552567
              2 Anestesiologia y Recuperacion      3 Medicina General                           2064090006
              3 Anestesiologia Pediatrica          1 Especialista en Anestesiologia            2706503459
              4 Cardiologia                        4 Medicina General                           3217185531
```

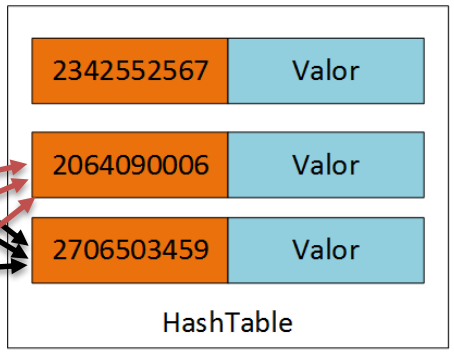
- Una vez que el hash table se ha construido, se comienza a iterar a la ***probe table*** que en este caso es representada por la tabla `medico`.
- Por cada registro de la tabla `medico`, se realizará una búsqueda en el HashTable empleando el valor del atributo `m.especialidad_id`

En SQL sería algo similar a la siguiente instrucción:

```
select hash_table.search(ora_hash(m.especialidad_id))
from medico;
```

Observar que al campo `m.especialidad_id` se le aplica la función `ora_hash`. El código hash generado se emplea como llave para localizar correspondencias en el hash table.

Medico		key
m.especialidad_id	m.nombre	ora_hash(m.especialidad_id)
3	Ale	ora_hash(3)= **2706503459**
3	Amalia	ora_hash(3)= **2706503459**
2	Aurora	ora_hash(2)= **2064090006**
2	Aldo	ora_hash(2)= **2064090006**
3	Felipe	ora_hash(3)= **2706503459**
2	Laura	ora_hash(2)= **2064090006**

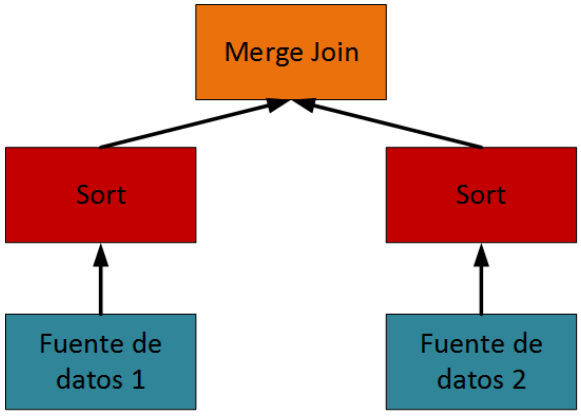


- Finalmente, se unen las columnas con base a las correspondencias anteriores y se obtiene el resultado.

Medico	Especialidad
m.nombre	e.nombre
Ale	Cardiología
Amalia	Cardiología
Aurora	Anestesiología
Aldo	Anestesiología
Felipe	Cardiología
Laura	Anestesiología

7.4.5.3. Sort Merge Joins

- Representa una variación de un Nested loop Join.
- La BD realiza un ordenamiento de las 2 fuentes de datos en caso de que estas no estén ya ordenadas (SORT JOIN).
- Por cada registro en la primera fuente de datos, se verifica si existe correspondencia con la segunda fuente de datos (MERGE JOIN).
- Esta operación se realiza de manera secuencial debido a que ambas fuentes de datos están ordenadas.



Esta estrategia puede emplearse bajo las siguientes condiciones:

- Se emplea en lugar de un Hash Join para procesar grandes cantidades de datos.
- La condición del join no es una igualdad, por ejemplo: >= , >, <=, <

- En un Hash Join siempre se requiere la igualdad "=". Por lo tanto, está técnica no se podría emplear con un "non-equijoin".
- SI existe un índice, el manejador puede omitir el ordenamiento del primer conjunto de datos. El segundo conjunto de datos siempre se ordena.
- Está técnica puede ser mejor que un Hash Join una vez que se cuenta con los datos ordenados, de no existir, el ordenamiento es más costoso que la construcción de una tabla hash.
- Si los datos no caben en memoria para construir un hash table, Sort Merge Join es mejor opción.
- En caso de que la memoria no sea suficiente para almacenar a los 2 conjuntos de datos, también se emplea disco, sin embargo, el número de lecturas es menor a las requeridas por un Hash Join.

Algoritmo de un Sort Merge Join.

```
read data_set_1 sort by join key to temp_ds1
read data_set_2 sort by join key to temp_ds2

read ds1_row from temp_ds1
read ds2_row from temp_ds2

while not eof on temp_ds1,temp_ds2
loop
    if ( temp_ds1.key = temp_ds2.key ) output join ds1_row,ds2_row
    elsif ( temp_ds1.key <= temp_ds2.key ) read ds1_row from temp_ds1
    elsif ( temp_ds1.key => temp_ds2.key ) read ds2_row from temp_ds2
end loop
```

Ejemplo:

Suponer los siguientes conjuntos de datos ordenados:

Data Set 1

```
10,20,30,40,50,60,70
```

Data set 2

```
20,20,40,40,40,40,40,60,70,70
```

Comparación 1
Ds1 Ds2
10 < 20 => **no match,** continua con el siguiente valor de Ds1, no hay match con 10 **next(Ds1)**
20 = 20 => **match**, continua revisando con Ds2 **next(Ds2)**
20 = 20 => **match** , (segundo valor de Set 2), continua con Ds2 **next(Ds2)**
20 < 40 => **no match** continua con el siguiente valor de Ds1, no hay match con 20 **next(Ds1)**
30 > 20 => **no match** compara con el último match de Ds2, en este caso con el segundo 20 **next(Ds2)** ya que puede haber match al ser mayor
30 < 40 => **no match, next(Ds1)** ya no puede existir match
40 > 20 => **no match,** next(Ds2)
40 = 40 => match , next(Ds2)
40 = 40 => match , next(Ds2)
40 = 40 => match , next(Ds2)
40 = 40 => match , next(Ds2)

40 = 40 => match , next(Ds2)
40 < 60 => next(Ds1) ya no puede existir match
50 > 40 => next(Ds2)
50 < 60 => next(Ds1)
70 > 40 => next(Ds2)
70 > 60 => next(Ds2)
70 = 70 => match, next(Ds2)
70 = 70 => match , next(Ds2)

- Observar que no se necesita leer todos los valores se Ds2 , esto representa una ventaja sobre Nested Loop.

Ejemplo:
Obtener los nombres de los médicos y los nombres de sus especialidades. Considerar únicamente nombres de médicos que inician con A.

```
explain plan for
select m.nombre,e.nombre
from especialidad e, medico m
where e.especialidad_id = m.especialidad_id
and m.nombre like 'A%' ;

select plan_table_output from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
------------------------------------------------
Plan hash value: 3154104233

-------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                 |   141 |  4794 |   241   (1)| 00:00:01 |
|   1 |  MERGE JOIN                   |                 |   141 |  4794 |   241   (1)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| ESPECIALIDAD    |    53 |  1272 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN            | ESPECIALIDAD_PK |    53 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                   |                 |   141 |  1410 |   239   (1)| 00:00:01 |
|*  5 |    TABLE ACCESS FULL          | MEDICO          |   141 |  1410 |   238   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("E"."ESPECIALIDAD_ID"="M"."ESPECIALIDAD_ID")
       filter("E"."ESPECIALIDAD_ID"="M"."ESPECIALIDAD_ID")
   5 - filter("M"."NOMBRE" LIKE 'A%')
```

A. **Id 3.** Se realiza un *index full scan* del índice `especialidad_pk`.
B. **Id 2**. Se obtienen todos datos de `especialidad` empleando un *table acces by index row id* (53 registros). Esto se realiza de esta manera para tener los datos ordenados.
C. **Id 5.** Se realiza un table access full a `medico` aplicando el predicado `nombre like 'A%'`, se estiman 141 registros.
D. **Id 4.** Al no contar con un índice aquí, se tiene que lanzar una operación de ordenamiento sobre los 141 registros: SORT JOIN en el paso 4.

E. **Id 1.** Una vez que se tienen los 2 conjuntos de datos ordenados, se aplica el algoritmo MERGE JOIN en el paso 1.

7.4.5.4. Cartesian Joins

- Se produce al omitir alguna condición de JOIN.
- En algunos casos el optimizador puede seleccionar este método, por ejemplo, hacer un producto cartesiano entre 2 tablas muy pequeñas que en conjunto hacen JOIN con una tabla grande.

Algoritmo:

```
for ds1_row in ds1 loop
  for ds2_row in ds2 loop
    output ds1_row and ds2_row
  end loop
end loop
```

- ds1 corresponde con el conjunto de datos de menor tamaño.

Ejemplo:

```
SQL>
explain plan for
select  m.nombre, e.nombre
from medico m, especialidad e;

SQL> select plan_table_output from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
--------------------------------------------------
Plan hash value: 631757493

-------------------------------------------------------------------------------
| Id  | Operation           | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |              |   265K|  7246K|  7016   (1)| 00:00:01 |
|   1 |  MERGE JOIN CARTESIAN|              |   265K|  7246K|  7016   (1)| 00:00:01 |
|   2 |   TABLE ACCESS FULL  | MEDICO       |  5000 | 35000 |   238   (0)| 00:00:01 |
|   3 |   BUFFER SORT        |              |    53 |  1113 |  6778   (1)| 00:00:01 |
|   4 |    TABLE ACCESS FULL | ESPECIALIDAD |    53 |  1113 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

A. El primer paso a ejecutar es el número 4. Se obtiene el contenido completo de `especialidad`.
B. El siguiente paso es el 3. BUFFER SORT significa, copiar los datos de la SGA a la PGA. Esto se debe a que el producto requiere múltiples lecturas hacia los datos de `especialidad`. Para evitar la contención con otras consultas, se aíslan estas lecturas, pasando los datos a la PGA.
C. En el paso 1, se aplica el algoritmo para obtener el producto.