

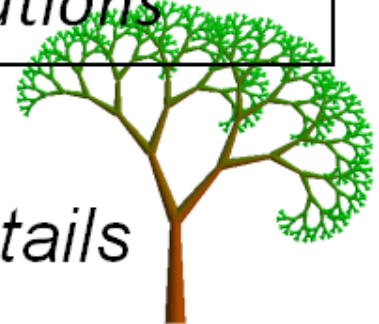


Query Tuning Foundations

Execution Plan

A - Row Source Operations

Branch and Leaf Details



Muriel Fry - Version V1.4 - Sep 2009

- ***Execution Plan***
 - ***A - Row Source Operations***
 - B - Find the Plan
 - C - Interpret the Plan



Objectives

- After completing this lesson, you should be able to do the following:
 - Understand the meaning of usual operations
 - Become familiar with very simple execution plan

ORACLE

Objectives

Objective from this part is to be more familiar with usual operations found in Execution plan.



Row Source Operation Description

- Unary Operations
 - 1 - Access Path
 - 2 – Other Unary Operations
- Binary Operations
 - 3 – Joins
- N-ary Operations
 - 4 – Other N-ary Operations

ORACLE

Row Source Operation Description

A row source is a set of data that can be accessed in a query. It can be a table, an index, a NON-mergeable View, or even the result set of a join tree consisting of many different objects.

Row Source Operations have been classified in 3 parts :

- **Unary Operation** : operations that act on only 1 input (Access Path and some other operations)
- **Binary Operation** : operations that act on 2 inputs (Joins)
- **N-ary Operation** : operations that act several inputs (Relational Operator)

1 Access Path

- Full Table Scan
- Rowid Scan
- Index Scan (Unique)

- Index Scan (Range)
- Index Scan (Full)
- Index Scan (Fast Full)
- Index Skip (Scan)
- Index Scan (Index Join)
- Using Bitmap Indexes
- Combining Bitmap Indexes

2 Other Unary Operations

- Sort – Hash
- Buffer Sort
- InList Iterator
- View
- Count Stop Key
- First Row / (Min/Max)

3 Join Tables

- Nested Loops
- Sort Merge
- Hash Join
- Cartesian Product
- Join Types (Anti - Semi - Star)

4 Other N-ary Operations

- Filter
- Concatenation
- Connect by
- Union all / Intersect / Minus

1 Structures / Access Paths

STRUCTURES

1. Tables (1 - 2)
2. Indexes to access tables
B_tree (3 - 4 - 5 - 6 - 7 - 8)
Index bitmap (9 - 10)
3. Tables organized by :
Cluster
Hash cluster
IOT

ACCESS PATHS

1. Full Table Scan
2. Rowid Scan
3. Index Scan (Unique)
4. Index Scan (Range)
5. Index Scan (Full)
6. Index Scan (Fast Full)
7. Index Scan (Skip)
8. Index Scan (Index Join)
9. Using Bitmap Indexes
10. Combining Bitmap Indexes

ORACLE

Unary Operations

1 - Access Path

2 - Other Unary Operations

Unary Operation : operations that act on only 1 input

Access paths are ways in which data are retrieved from the database. Any row can be located and retrieved with one of the methods mentioned in the slide.

In general, index access paths should be used for statements that retrieve a small subset of table rows, while full scans are more efficient when accessing a large portion of the table. This is costed by optimizer and will be detailed in next Module.

Not mentioned on the slide, there are special types of table access paths including clusters, index organized tables, and partitions.

Clusters are an optional method of storing table data. A cluster is a group of tables that share the same data blocks because they share common columns and are often used together. For example, the EMP

and DEPT table share the DEPTNO column. When you cluster the EMP and DEPT tables, Oracle physically stores all rows for each department from both the EMP and DEPT tables in the same data blocks.

Hash clusters are single-table clusters in which rows with the same hash-key values are stored together. Oracle uses a mathematical hash function to choose the location of a row within the cluster. All rows with the same key value are stored together on disk.

More info on these alternative Table Structures :

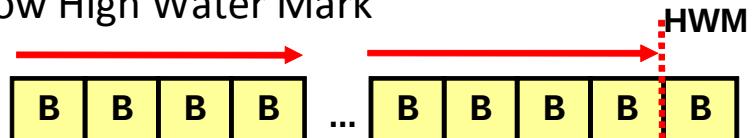
Oracle Open World 2006 – Access Path Optimization – PS_S282983_282983_139-1_FIN_v1.pdf
<http://www28.cplan.com/cc139/catalog.jsp> ([cboracle/oracle6](#))

More information in documentation

Oracle® Database Performance Tuning Guide
10/g/ Release 2 (10.2)
Part Number B14211-01
15 Using Indexes and Clusters

1.1 - Full Table Scan

- Performs Multi-block Reads
(here DB_FILE_MULTIBLOCK_READ_COUNT = 4)
- Reads All Blocks Below High Water Mark
- Filters rows



```
select * from emp where ename = 'KING';
```

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		1	22	2
* 1	TABLE ACCESS FULL	EMP	1	22	2

Predicate Information (identified by operation id):

```
1 - filter("EMP"."ENAME"='KING')
```

ORACLE

FULL TABLE SCAN

This type of scan reads sequentially all rows from a table and filters out those that do not meet the selection criteria. During a full table scan, all blocks in the table that are under the high-water mark are scanned (even if all rows have been deleted from table). Each row is examined to determine whether it satisfies the statement's WHERE clause (using the applicable Filter conditions specified in the query).

Filter Conditions can be seen in the "Predicate Information" Section of the explain plan: The filter to be applied will only return rows where ""EMP"."ENAME"='KING'".

Each row is checked to determine that it matches the condition so that the engine will only output rows for which ("EMP"."ENAME"='KING').

Because a FTS reads all the blocks in a table it will likely read blocks that are physically adjacent to each other. This means that performance benefits can be reaped by utilising I/O calls that read multiple blocks. The size of the read call can range from a single block to any number of blocks up to the init parameter DB_FILE_MULTIBLOCK_READ_COUNT. Using multiblock reads means a full table scan can be performed very efficiently.

Column definition

There will be further explanation in Part2.(Interpret the Plan)

Column	Description
ID	Identification number for this step in the execution plan
OPERATION	Name of the operation performed at this step
COST	Cost of the current operation estimated by the cost-based optimizer (CBO)
CARDINALITY	Number of rows returned by the current operation (estimated by the CBO)
BYTES	Number of bytes returned by the current operation
FILTER_PREDICATES	Predicates used to filter rows before producing them

Read Plan

```
select * from emp where ename = 'KING';
```

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		1	22	2
* 1	TABLE ACCESS FULL	EMP	1	22	2

Predicate Information (identified by operation id):

```
1 - filter("EMP"."ENAME"='KING')
```

- Oracle fetches rows from table EMP
- Then, Oracle applies filter "EMP"."ENAME"='KING' from these fetched rows to output final resultset. (Filter is *FILTER PREDICATES* column)

Why the bad reputation?

In Oracle 6, full-table scans flooded the buffer cache because there was no difference in the way that blocks were handled between full-table scans and reads by index

That was then, this is now...

Since Oracle V7, blocks read by full-table scans don't flood the buffer cache because the space they are allowed to occupy is restricted to a small amount.

Hash joins and parallel query, added in Oracle 7.3, rely heavily on full-table scans for efficiency.

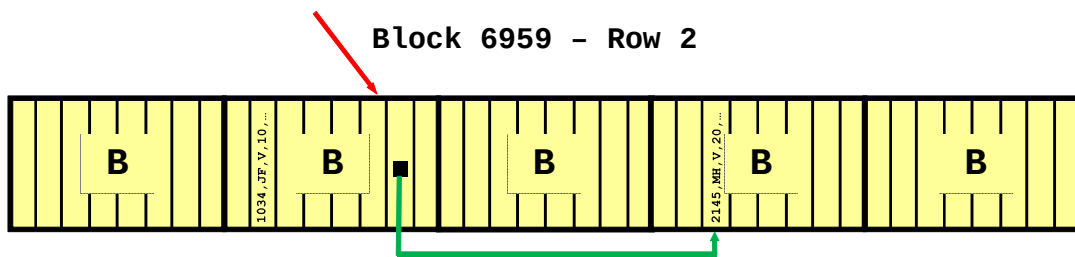
Note.104937.1 Oracle8i New Buffer Replacement Algorithm

http://structureddata.org/2008/08/14/automatic-db_file_multiblock_read_count/

1.2 - Rowid Scan

```
SQL> explain plan for
2 select * from emp where rowid='AAAJ5QAAFAAABsvAAC';
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	1
1	TABLE ACCESS BY USER ROWID	EMP	1	1



ORACLE

ROWID SCAN

The rowID of a row specifies the data file and data block containing the row as well as the location of the row in that block.

- ROWIDs are pointers to a particular row in a block
- ROWIDs can be used to extract a file and block number needed to dump a specific block
- When to use the ROWID access method:
 - When data is indexed, the index block contains the ROWID for each row for faster access
 - ROWID can also be used to extract data from the block when regular access methods fail

Locating a row by specifying its row ID is the fastest way to retrieve a single row, because the exact location of the row in the database is specified.

To access a table by rowID, Oracle first obtains the rowIDs of the selected rows, either from the statement's WHERE clause or through an index scan of one or more. Oracle then locates each selected row in the table based on its rowID.

Most of the time, the optimizer uses rowids after retrieving them from an index. The table access might

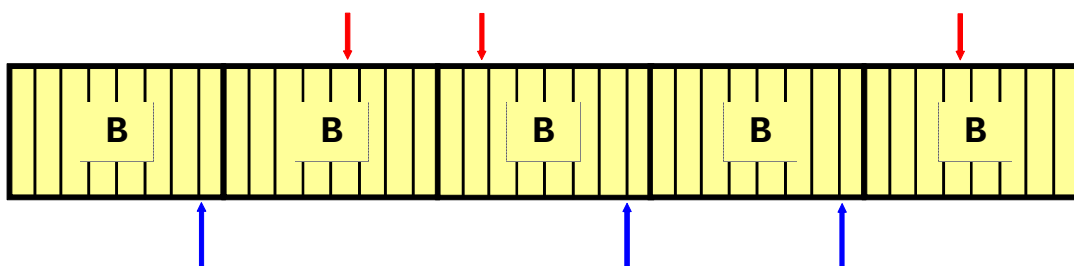
be required for any columns in the statement not present in the index. A table access by rowid does not need to follow every index scan. If the index contains all the columns needed for the statement, then table access by rowid might not occur.

Note: Due to row migration, a rowID can sometimes point to a different address than the actual row location, resulting in more than one block being read to locate a row. By example, an update to a row causes the row to be placed in another block with a pointer in the original block. The rowID, however, will still have only the address of the original block.

Sample Table Scan

```
SELECT * FROM emp SAMPLE BLOCK (10) [SEED (1)];
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)
0	SELECT STATEMENT		4	99	2 (0)
1	TABLE ACCESS SAMPLE	EMP	4	99	2 (0)



ORACLE

SAMPLE TABLE SCAN

A sample table scan retrieves a random sample of data from a simple table or a complex SELECT statement, such as a statement involving joins and views. This access path is used when a statement's FROM clause includes the SAMPLE clause or the SAMPLE BLOCK clause.

- To perform a sample table scan when sampling by rows with the SAMPLE clause, Oracle reads a specified percentage of rows in the table and examines each of these rows to determine whether it satisfies the statement's WHERE clause.
- To perform a sample table scan when sampling by blocks with the SAMPLE BLOCK clause, Oracle reads a specified percentage of table's blocks and examines each row in the sampled blocks to determine whether it satisfies the statement's WHERE clause..

The sample percent is a number specifying the percentage of the total row or block count to be included in the sample. The Sample value must be in the range [0.000001 , 99.999999]. This percentage indicates the probability of each row, or each cluster of rows in the case of block sampling, being selected as part of the sample. It does not mean that the database will retrieve exactly sample_percent of the rows of table.



Types of Indexes

- Index storage techniques:
 - **B*-tree**
 - Normal
 - Descending
 - Reverse key
 - Key compression
 - Function based
 - Index Organized Table
 - **Bitmap** (Stored in a B-Tree)
 - Domain indexes

ORACLE

Indexes

An index is a database object that is logically and physically independent of the table data. Oracle may use an index to access data that is required by a SQL statement, or it may use indexes to enforce integrity constraints. You can create and drop indexes at any time. Oracle automatically maintains indexes when the related data changes.

Indexes can be unique or nonunique. A composite index (also called a concatenated index) is an index that you create on multiple columns (up to 32) in a table. Columns in a composite index can appear in any order and need not be adjacent in the table.

Index Storage Techniques

For standard indexes, Oracle uses **B*-tree indexes** that are balanced to equalize access times. B-tree indexes can be normal, reverse key, descending, or function based.

B*Tree indexes: They are by far the most common indexes. Similar in construct to a binary tree, B*Tree indexes provide fast access, by key, to an individual row or range of rows, normally requiring few

reads to find the correct row. However, that the “B” in “B*Tree” does not stand for *binary* but rather for *balanced*.

The term B-tree (or a variant, the B*-tree) was first described by Bayer and McCreight in 1972. A B*-tree is a data structure that maintains an ordered set of data and allows efficient operations to find, delete, insert, and browse the data. A B*-tree consists of node records containing the keys, and pointers that link the nodes of the B*-tree together.

Keys are kept in sorted order within each node. A corresponding list of pointers are effectively interspersed between keys to indicate where to search for a key if it is not in the current node.

The B*-tree structure has the following advantages:

- All leaf blocks of the tree are at the same depth, so retrieval of any record from anywhere in the index takes approximately the same amount of time.
 - B*-tree indexes automatically stay balanced.
 - All blocks of the B*-tree are three-quarters full on the average.
 - B*-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.
 - Inserts, updates, and deletes are efficient, maintaining key order for fast retrieval.
-
- B*-trees are automatically balanced by definition
 - B*-tree level always grows bottom up
 - B*-trees start balanced (one node); all operations changing its height leave it balanced

Insert Operation

Step 1: The search algorithm locates the node at the lowest level of the index set in which a key logically belongs. If for example node N contains free space, key K is inserted into the node N and the process terminates. Otherwise:

Step 2: Node N is split into two nodes and then the keys are ordered. In the worst case, splitting occurs all the way to the top of the tree; a root node (parent node) must be split and the height of the tree will increase by one level. Entries are made into the leaf block in ROWID order. So all the blocks are scanned in turn until an entry is found with a greater ROWID than the new row, and the row is inserted into that block.

Delete Operation

The delete operation is the opposite of an insert. When an index entry must be deleted, the row is deleted from the index leaf block and the space within the index leaf block released to the block for further inserts with the appropriate key range. If a leaf block has even one entry, it is still part of the tree; hence only entries that belong in it positionally can be accommodated. After a leaf block is completely empty, it is put on the free list.

Update Operation

Changing values is handled by deleting the old values and inserting the new value

Descending indexes: Descending indexes allow for data to be sorted from “big to small” (descending) instead of “small to big” (ascending) in the index structure.

Reverse key indexes: These are B*Tree indexes whereby the bytes in the key are “reversed.” Reverse key indexes can be used to obtain a more even distribution of index entries throughout an index that is populated with increasing values. For example, if I am using a sequence to generate a primary key, the sequence will generate values like 987500, 987501, 987502, and so on. With a reverse key index, Oracle will logically index 005789, 105789, 205789, and so on instead of 987500, 987501, 987502. Because these reverse keys are now likely to be placed in different locations this can reduce contention for particular blocks that may otherwise be targets for contention.

Index Key Compression: The basic concept behind a compressed key index is that every entry is broken into two pieces: a “prefix” and “suffix” component. The prefix is built on the leading columns of the concatenated index and will have many repeating values. The suffix is built on the trailing columns in the index key and is the unique component of the index entry within the prefix. This is not compression in the same manner that ZIP files are compressed; rather, this is an optional compression that removes redundancies from concatenated (multicolumn) indexes.

Function-based indexes: These are B*Tree or bitmap indexes that store the computed result of a function on a row’s column(s), not the column data itself. You can consider them an index on a virtual (or derived / hidden) column—in other words, a column that is not physically stored in the table. You can gather statistics on this virtual column created.

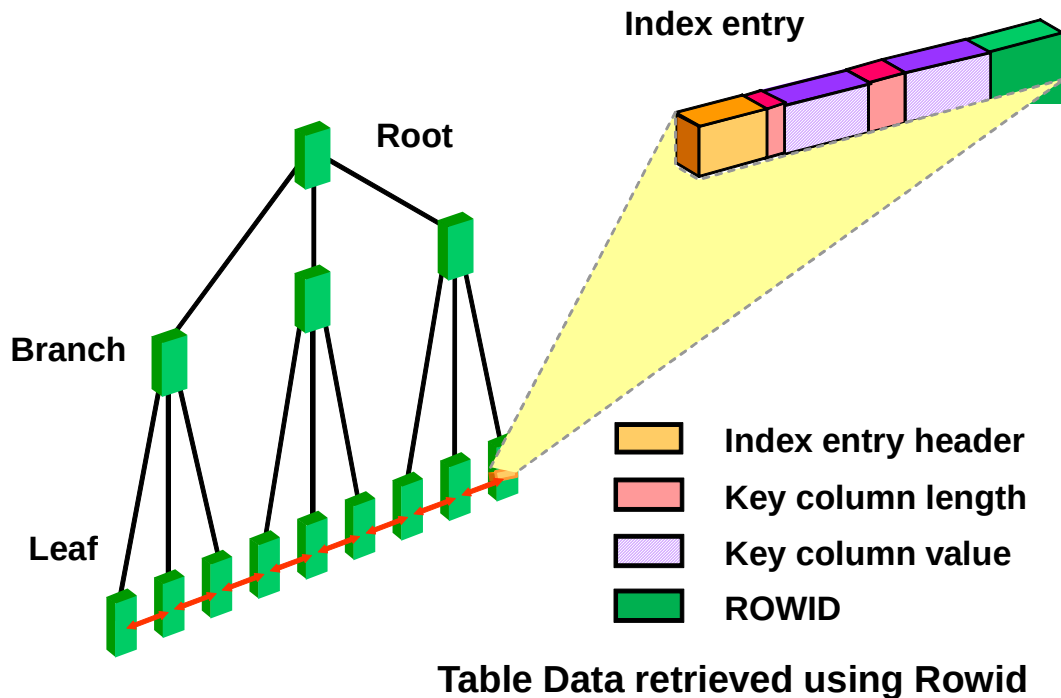
Index organized tables: These are tables stored in a B*Tree structure. Whereas rows of data in a heap table are stored in an unorganized fashion (data goes wherever there is available space), data in an IOT is stored and sorted by primary key. IOTs behave just like “regular” tables as far as your application is concerned.

Bitmap indexes: Normally in a B*Tree, there is a one-to-one relationship between an index entry and a row: an index entry points to a row. With bitmap indexes, a single index entry uses a bitmap to point to many rows simultaneously. They are appropriate for highly repetitive data (data with few distinct values *relative to the total number of rows in the table*) that is mostly read-only. Bitmap indexes should never be considered in an OLTP database for concurrency-related issues.

Bitmap join indexes: A bitmap join index is a bitmap index for the join of two or more tables. A bitmap join index can be used to avoid actual joins of tables, or to greatly reduce the volume of data that must be joined, by performing restrictions in advance. Queries using bitmap join indexes can be sped up via bit-wise operations.

Application domain indexes: These are indexes you build with packages and store yourself, either in Oracle or perhaps even outside of Oracle. You tell the optimizer how selective your index is and how costly it is to execute, and the optimizer will decide whether or not to use your index based on that information.

B*-Tree Indexes



ORACLE

B*-tree Indexes

- Branch blocks hold *<separator key, DBA>* pairs to guide a B*-tree search to a leaf block entry
- Leaf blocks hold *<index key, ROWID>* pairs

Each B*-tree index has a root block as a starting point. Depending on the number of entries, there are one or more branch blocks that can have one or more leaf blocks. The leaf blocks contain all values of the index plus ROWIDs that point to the rows in the associated data segment.

Previous and next block pointers connect the leaf blocks so that they can be traversed from left to right (and vice versa).

Indexes are always balanced, and they grow from the bottom up. In certain situations, the balancing algorithm can cause the B*-tree height to increase unnecessarily. It is possible to reorganize indexes. This is done by `ALTER INDEX ... REBUILD` command.

The internal structure of a B*-tree index allows rapid access to the indexed values. Oracle can directly

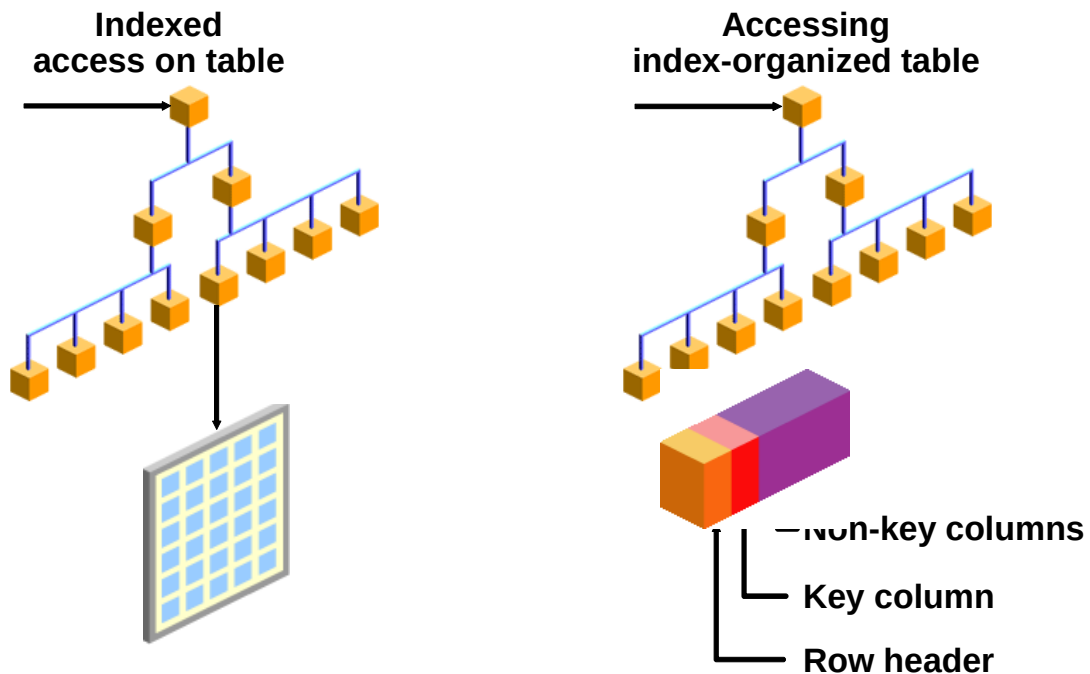
access rows after it has retrieved the address (the ROWID) from the index leaf blocks.

Note.260613.1 Indexes and NULL

Interesting External links

- **How Are NULLs Actually Stored** : <http://richardfoote.files.wordpress.com/2008/01/nulls-dump-demo.txt>
- **Indexing NULLs** : <http://richardfoote.files.wordpress.com/2008/01/indexing-nulls.txt>
- **Unbalanced Indexes?** : <http://www.dbazine.com/oracle/or-articles/jlewis13/>
- **Rebuilding Indexes – why, when, how?** <http://www.jlcomp.demon.co.uk/indexes.doc>
- **comments** : <http://www.jlcomp.demon.co.uk/communication.html>

Index-Organized Tables



ORACLE

Index-Organized Tables: Overview

An index-organized table (IOT) is like a regular table with a concatenated index on all of its columns. However, the key values (for the table and the B*-tree index) are stored in the same segment. An IOT contains:

Primary key values

Other (non-key) column values for the row

The B*-tree structure, which is based on the primary key of the table, is organized like an index. The leaf blocks in this structure contain the rows instead of the row IDs. This means that the rows in the IOT are always maintained in the order of the primary key.

You can create additional indexes on IOTs. The primary key can be composite of many columns. Because large rows of an IOT can destroy the dense and efficient storage of the B*-tree structure, you can store part of the row in another segment, which is called an *overflow area*.

Benefits of IOTs

Index-organized tables provide fast key-based access to table data for queries involving exact match and range searches. Changes to the table data result only in updating the index structure. Also, storage requirements are reduced because key columns are not duplicated in the table and index. The remaining non-key columns are stored in the index structure.

IOTS are particularly useful when you are using applications that must retrieve data based on a primary key and has only a few, relatively short non-key columns.

No special considerations exist for using most SQL statements against an IOT. The structure of the table should be completely transparent to the user.

IOT can be partitioned

[Bug.5095700](#) 10.2.0.3 Partition pruning does not occur DBMS_STATS for partitioned IOTs

[Bug.5236908](#) 10.2.0.4 Primary key index not used for IOT

[Bug.4725385](#) 10.2.0.4 DBMS_STATS on partitioned IOT with additional indexes can be slow

=> have to use cascade =TRUE option get full bug fix benefice

[Bug.5406763](#) 11.1.0.7 Wrong cost when access IOT through secondary index



- Unique
- Range (Descending)
- Skip
- Full & Fast Full
- Index join

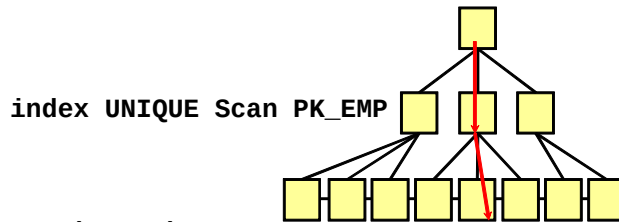


INDEX SCAN

The index contains not only the indexed value but also the rowIDs of rows in the table having that value. Therefore, if the statement accesses other columns in addition to the indexed columns, then Oracle can find the rows in the table by using a table access by rowID.

In that example, 4 rows are retrieved from Table using their Rowid gotten by the index scan.

1.3 - Index Scan (Unique)



```
create unique index PK_EMP on EMP(empno)
```

```
select * from emp where empno = 9999;
```

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		1	32	1
1	TABLE ACCESS BY INDEX ROWID	EMP	1	32	1
* 2	INDEX UNIQUE SCAN	PK_EMP	1		

Predicate Information (identified by operation id):

2 - **access(EMP.EMPNO=9999)**

ORACLE

Index Unique Scan

An *index unique scan* returns, at most, a single rowID. Oracle performs a unique scan if a statement contains a UNIQUE or a PRIMARY KEY constraint that guarantees that only a single row is accessed. This access path is used when all columns of a unique (B-tree) index are specified with equality conditions.

- Key values and Rowid are gotten from Index
- Table rows are gotten using Rowids

Access Conditions can be seen in "**Predicate Information**" Section :

Oracle accesses only matching rows ("**EMP**".**"EMPNO"**=9999)

Reminder : Filter Conditions filter rows after fetch operation and output filtered rows.

Read Plan

ACCESS_PREDICATES	Predicates used to locate rows in an access structure. For example, start or stop predicates for an index range scan.
-------------------	---

```
create unique index PK_EMP on EMP(empno)
select * from emp where empno = 9999;
```

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		1	32	1
1	TABLE ACCESS BY INDEX ROWID	EMP	1	32	1
* 2	INDEX UNIQUE SCAN	PK_EMP	1		

Predicate Information (identified by operation id):

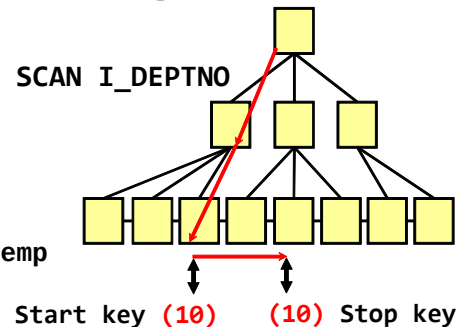
2 - **access**(EMP.EMPNO=9999)

- Using unique index PK_EMP,
- Oracle accesses row for which EMP.EMPNO=9999,
- Oracle gets its rowid, fetch other columns from table EMP
- It outputs final resultset.

1.4 - Index Scan (Range)

```
create index I_DEPTNO on EMP(deptno)
```

```
select /*+ INDEX(EMP I_DEPTNO) */ * from emp  
where deptno = 10 and sal > 1000;
```



Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		4	128	276
* 1	TABLE ACCESS BY INDEX ROWID	EMP	4	128	276
* 2	INDEX RANGE SCAN	I_DEPTNO	5		9

Predicate Information (identified by operation id):

- 1 - **filter**(EMP.SAL>1000) - Throw out non-matching rows
- 2 - **access**(EMP.DEPTNO=10) - means read data from Index, Table

ORACLE

Index Range Scan

An *index range scan* is a common operation for accessing selective data. It can be bounded (on both sides) or unbounded (on one or both sides). Data is returned in the ascending order of index columns. Multiple rows with identical values are sorted in ascending order by rowID.

The optimizer uses a range scan when it finds one or more leading columns of an index specified in conditions (WHERE clause), such as `col1 = :b1`, `col1 < :b1`, `col1 > :b1`, and any combination of the preceding conditions.

Wildcard searches (`col1 like '%ASD'`) should not be in a leading position, as this does not result in a range scan.

Range scans can use unique or non-unique indexes. Range scans can avoid sorting when index columns constitute the ORDER BY clause.

Read Plan from Slide

- Using index I_DEPTNO,

- Oracle accesses rows for which EMP.DEPTNO=10,
- Oracle gets their ROWIDs, fetch other columns from table EMP
- finally, Oracle applies filter EMP.SAL >1000 from these fetched rows to output final resultset.

Note 67409.1: When will an ORDER BY statement use an index ?

Another example

```
SQL> create index IX_A on EMP(DEPTNO asc);
SQL> select * from EMP where deptno < 30 order by deptno asc;
```

EMPNO	ENAME	JOB	SAL	COMM	DEPTNO
7782	CLARK	MANAGER	2450		10
7839	KING	PRESIDENT	5000		10
7934	MILLER	CLERK	1300		10
7369	SMITH	CLERK	800		20
7566	JONES	MANAGER	2975		20
7788	SCOTT	ANALYST	3000		20
7876	ADAMS	CLERK	1100		20
7902	FORD	ANALYST	3000		20

Rows are ordered by DEPTNO ascending without SORT operation using INDEX RANGE SCAN operation

Id	Operation	Name
1	TABLE ACCESS BY INDEX ROWID	EMP
* 2	INDEX RANGE SCAN	IX_A

2 - access(DEPTNO<30)
filter(DEPTNO<30)

Read Plan

- Using index IX_A,
- Oracle accesses rows for which EMP.DEPTNO<30,
- Oracle gets their ROWIDs, fetch other columns from table EMP
- final resultset is sorted from low to high deptno.

Index Range Scan Descending

An index range scan descending is identical to an index range scan, except that the data is returned in descending order. The optimizer uses index range scan descending when an order by descending clause can be satisfied by an index.

The hint `INDEX_DESC(table_alias index_name)` can be used to force this access path if possible.

```
SQL> create index IX_A on EMP(DEPTNO asc);
SQL> select * from EMP where deptno < 30 order by deptno desc;
```

EMPNO	ENAME	JOB	SAL	COMM	DEPTNO
7369	SMITH	CLERK	800		20
7566	JONES	MANAGER	2975		20
7788	SCOTT	ANALYST	3000		20
7876	ADAMS	CLERK	1100		20
7902	FORD	ANALYST	3000		20
7782	CLARK	MANAGER	2450		10
7839	KING	PRESIDENT	5000		10
7934	MILLER	CLERK	1300		10

Rows are ordered by DEPTNO descending without SORT operation using INDEX RANGE SCAN operation

Id	Operation	Name
1	TABLE ACCESS BY INDEX ROWID	EMP
* 2	INDEX RANGE SCAN <i>DESCENDING</i>	IX_A

2 - access(DEPTNO<30)
filter(DEPTNO<30)

Read Plan

- Using index IX_A in descending order,
- Oracle accesses rows for which EMP.DEPTNO<30,
- Oracle gets their ROWIDs, fetch other columns from table EMP
- final resultset is sorted from high to low deptno.



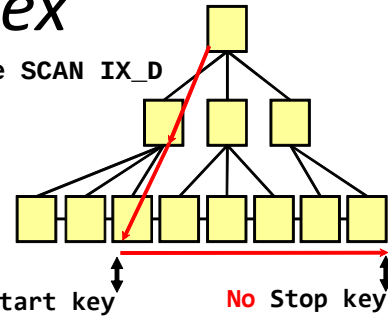
Index Scan (Range)

Descending Index

Index Range SCAN IX_D

```
create index IX_D on EMP(deptno DESC)
```

```
select /*+ INDEX(EMP IX_D) */ * from emp  
where deptno < 30 order by DESC;
```



Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT			1
1	TABLE ACCESS BY INDEX ROWID	EMP	9	1
* 2	INDEX RANGE SCAN	IX_D	1	2

Predicate Information (identified by operation id):

```
2 - access(SYS_OP_DESCEND(EMP.DEPTNO)>HEXTORAW('3EE0FF'))  
    filter(SYS_OP_UNDESCEND(SYS_OP_DESCEND(EMP.DEPTNO))<30)
```

ORACLE

Descending Index

Descending indexes allow for data to be sorted from "big to small" (descending) instead of "small to big" (ascending) in the index structure using INDEX RANGE SCAN .

```
SQL> create index IX_D on EMP(DEPTNO desc);  
SQL> select * from EMP where deptno < 30 order by deptno desc;
```

EMPNO	ENAME	JOB	SAL	COMM	DEPTNO
7369	SMITH	CLERK	800		20
7566	JONES	MANAGER	2975		20
7788	SCOTT	ANALYST	3000		20
7876	ADAMS	CLERK	1100		20
7902	FORD	ANALYST	3000		20
7782	CLARK	MANAGER	2450		10
7839	KING	PRESIDENT	5000		10

Rows are ordered by DEPTNO descending without SORT operation using INDEX RANGE SCAN operation.

Id	Operation	Name

1	TABLE ACCESS BY INDEX ROWID	EMP
* 2	INDEX RANGE SCAN	IX_D

```
2 - access(EMP.SYS_NC00009$>HEXTORAW('3EE0FF') )
      filter(SYS_OP_UNDESCEND(EMP.SYS_NC00009$)<30)
```

Read Plan

- Using index IX_D
- Oracle accesses rows for which EMP.DEPTNO<30, (applying sys_op_descend function -An internal function that takes a value and returns the form that would be stored for that value in a descending index. Essentially doing a one's complement on the bytes and appending an 0xFF byte).
- Oracle gets their rowid, fetch other columns from table EMP

Note : you can have same behaviour, rows ordered in descending order by using INDEX RANGE SCAN DESCENDING operation. (see previous slide comments)

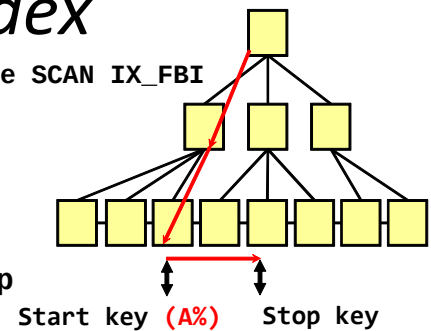


Index Scan (Range)

Function Based Index

```
create index IX_FBI on EMP(UPPER(ename));
```

```
select /*+ INDEX(EMP IX_FBI) */ * from emp  
where upper(ENAME) like 'A%';
```



Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	1
1	TABLE ACCESS BY INDEX ROWID	EMP	1	1
* 2	INDEX RANGE SCAN	IX_FBI	1	2

Predicate Information (identified by operation id):

```
2 - access(UPPER(EMP.ENAME) LIKE 'A%')  
    filter(UPPER(EMP.ENAME) LIKE 'A%')
```

ORACLE

Function Based Index

These are B*Tree (or bitmap) indexes that store the computed result of a function on a row's column(s), not the column data itself. You can consider them an index on a virtual (or derived / hidden) column—in other words, a column that is not physically stored in the table. You can gather statistics on this virtual column.

```
SQL> create index IX_FBI on emp(UPPER(ENAME));
```

In user_tab_cols : there is information on Virtual column created

- Column_name = SYS_NC00008\$
- data_default = UPPER("ENAME")

```
SQL> select * from emp where upper(ENAME) like 'A%';
```

EMPNO	ENAME	JOB	SAL	COMM	DEPTNO
-----	-----	-----	-----	-----	-----
7876	ADAMS	CLERK	1100		20
7499	ALLEN	SALESMAN	1600	300	30

Read Plan from Slide

- Using Function based index IX_FBI
- Oracle accesses rows for which upper(ENAME) like 'A%'.
- Oracle gets their rowid, fetch other columns from table EMP
- Oracle output final resultset.

Interesting External links

Index Only Values Of Interest : Details how a function-based index can be used to index only specific column values of interest, potentially substantially reducing the size of the index

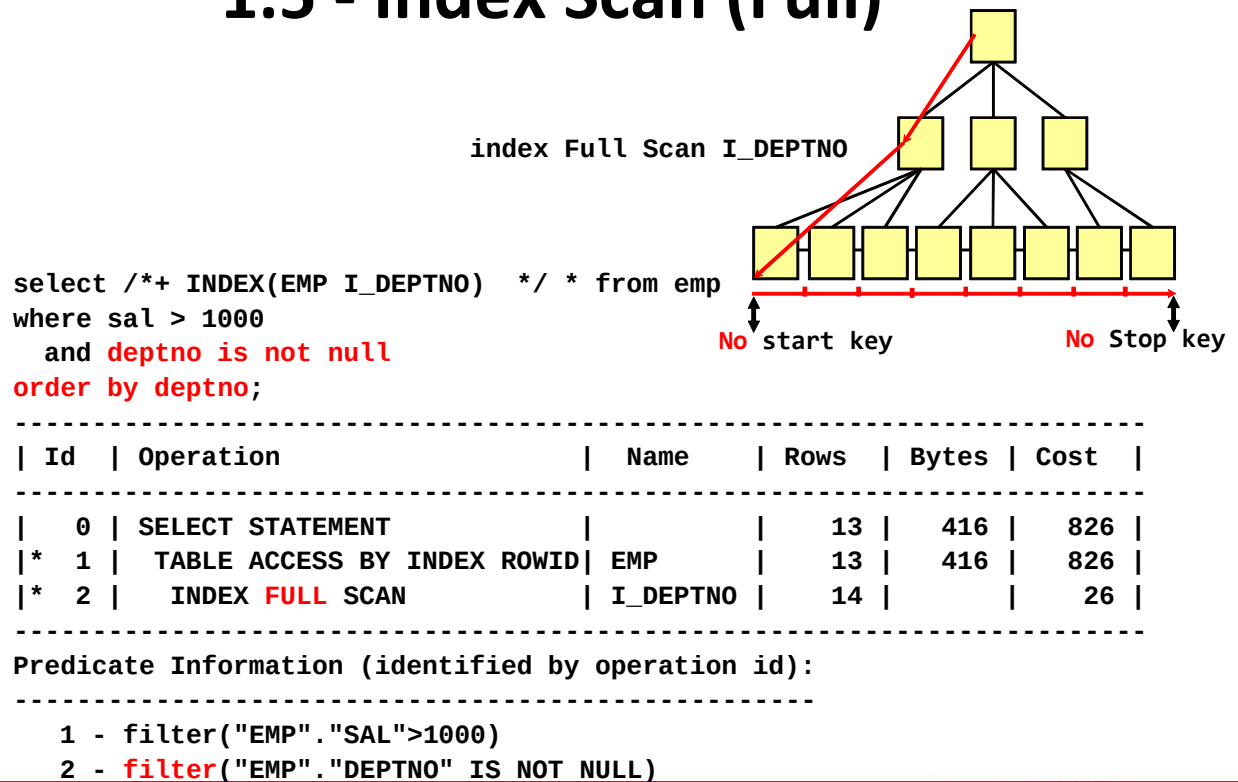
<http://richardfoote.files.wordpress.com/2008/01/index-only-some-values-demo-4.txt>

Function Based Index can be used (before 11g) to better estimate cardinality.

<http://jonathanlewis.wordpress.com/2006/10/31/virtual-columns-revisited/>

In 11g, instead of FBI, use virtual columns statistics [11gR1 Beta1 Multi Col Stats.swf](#)

1.5 - Index Scan (Full)



ORACLE

Full Scans

A *full scan* is available if a predicate references one of the columns in the index. The predicate does not need to be an index driver (leading column). It reads index using single-block IO (unlike a fast full index scan).

A full scan can be used to eliminate a sort operation, because the data is ordered by the index key (first_rows optimizer (see Module 2), NLS_SORT is BINARY). At least one of the index columns is not NULL (see Note 67409.1)

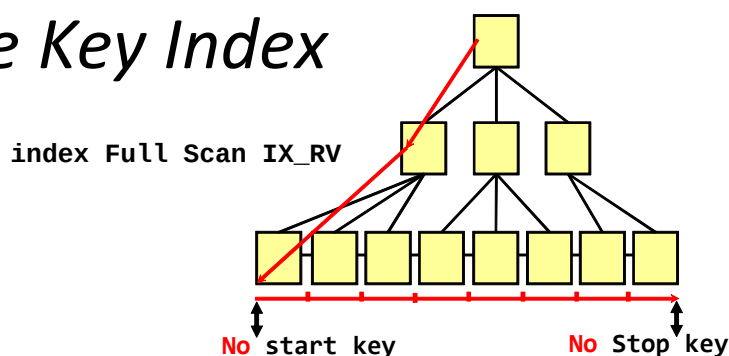
Read Plan

- Using index I_DEPTNO
- Oracle accesses all index entries .
- Oracle gets their rowid, fetch other columns from table EMP

Finally, Oracle applies filter EMP.SAL >1000 from these fetched rows to output final resultset. (Rows are already ordered because it have been accessed by index I_deptno)

Index Scan (Full)

Reverse Key Index



```
select /*+ index(emp)*/ * from emp where empno > 7700;
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		6	358
1	TABLE ACCESS BY INDEX ROWID	EMP	6	358
* 2	INDEX FULL SCAN	IX_RV	6	26

Predicate Information (identified by operation id):

2 - filter(EMP.EMPNO>7700)

ORACLE

Reverse Key Index

```
SQL> create index IX_RV on EMP(empno) reverse;
```

```
SQL> select /*+ index(emp IX_RV)*/ * from emp where empno > 7700;
```

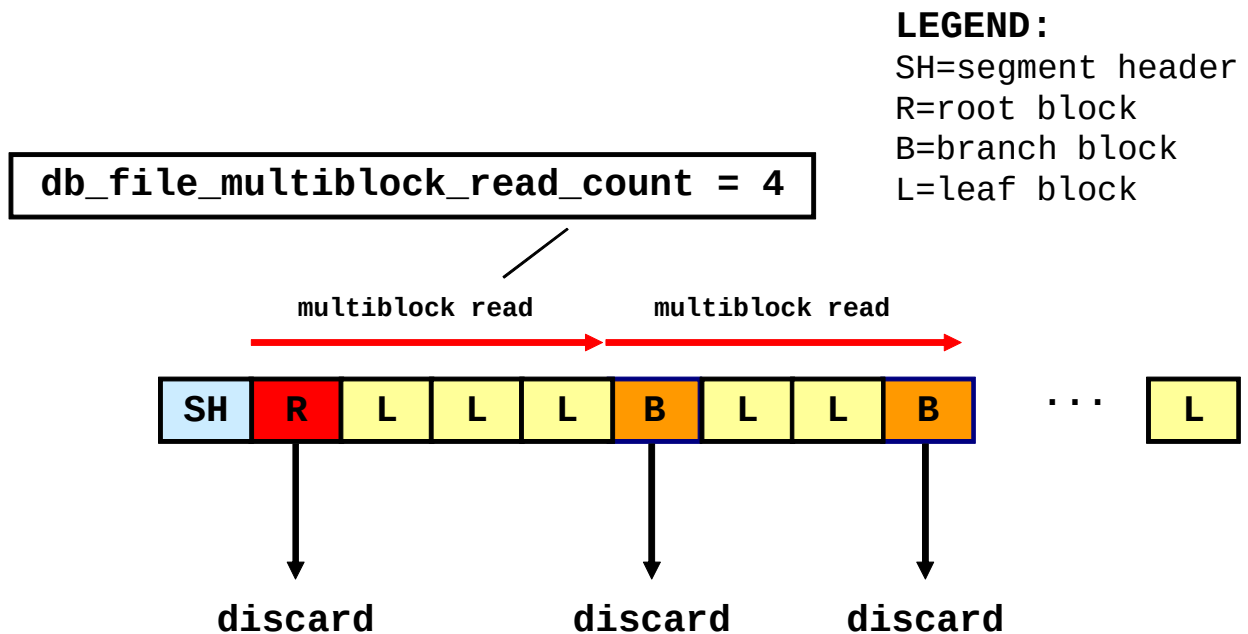
EMPNO	ENAME	JOB	SAL	COMM	DEPTNO
7902	FORD	ANALYST	3000		20
7934	MILLER	CLERK	1300		10
7839	KING	PRESIDENT	5000		10
7844	TURNER	SALESMAN	1500	0	30
7876	ADAMS	CLERK	1100		20
7900	JAMES	CLERK	950		30
7782	CLARK	MANAGER	2450		10
7788	SCOTT	ANALYST	3000		20

Here that is not an equality predicate and we need to do a *FULL scan* to find all **empno** with empno > 7700 (because value are stored like 0077 (that's the reverse) instead of 7700)

Read Plan

- Using index IX_RV
- Oracle accesses all index entries to get key empno > 7700
- Oracle gets their rowid, fetch other columns from table EMP
- Oracle output final resultset

1.6 - Index Scan (Fast Full)



ORACLE

Fast-Full Index Scans

Fast-full index scans are an alternative to full table scans when the index contains all the columns that are needed for the query and at least one column in the index key has the NOT NULL constraint. A fast-full scan accesses the data in the index itself without accessing the table.

It cannot be used to eliminate a sort operation because the data is *not* ordered by the index key or it can be used for aggregate functions min/avg/sum. In this case, CBO must know that all table rows are represented in index (at least 1 NOT NULL column)

It reads the entire index using multiblock reads (unlike a full index scan). Fast-full index scans cannot be performed against bitmap indexes. A fast-full scan is faster than a normal full index scan because it can use multiblock I/O just like a table scan.

```
select /*+ INDEX_FFS(EMP I_DEPTNO) */ deptno
  from emp
 where deptno is not null;
```

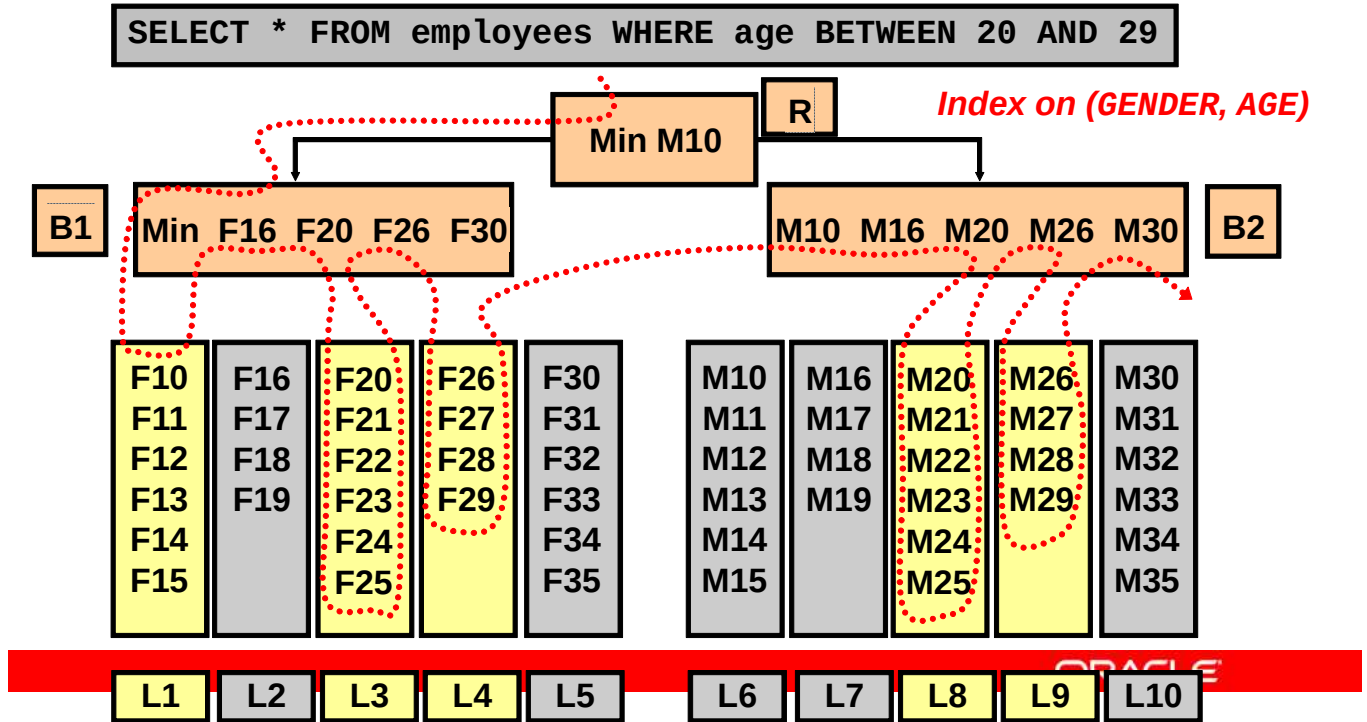
Id	Operation	Name	Rows	Cost
* 1	INDEX FAST FULL SCAN	I_DEPTNO	20	5

1 - filter("EMP"."DEPTNO" IS NOT NULL)

Read Plan

- Oracle fetches rows from index I_DEPTNO
- Oracle output final resultset.

1.7 – Index Scan (Skip) (1)



Index Skip Scan

Index skip scans improve index scans by skipping blocks that could never contain keys matching the filter column values. Scanning index blocks is often faster than scanning table data blocks. Skip scanning can happen when the initial (leading) column of the composite index is not specified in a query.

Suppose that there is a concatenated index on the columns GENDER and AGE in the EMPLOYEES table. This example illustrates how skip scanning is processed to answer the query in the slide.

The Oracle starts from the root of the index [R] and then proceeds to the left-most branch block[B1]. From here, the server identifies a first entry to be F16, goes to the left-most leaf[L1], and starts to scan it because it could contain A25 (ie where the 'gender' is before 'F' in the alphabet). The server identifies that this is not possible because the first entry is F10. It is thus not possible to find an entry such as A25 in this leaf so it can be skipped.

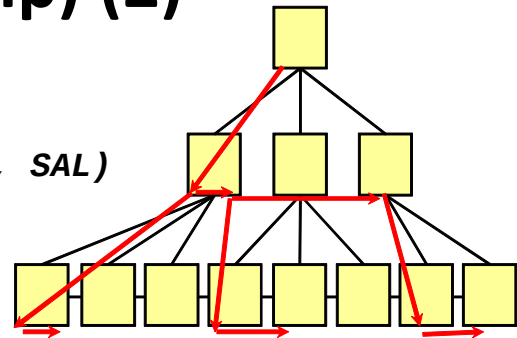
Backtracking to the first branch block [B1], the server identifies that the next subtree (F16) does not need to be scanned because the next entry in that branch block [B1] is F20. Because the server is certain that it is not possible to find a 25 between F16 and F20, the second leaf block [L2] can be skipped.

Returning to the first branch block [B1], the server finds that the following two entries have a common prefix of F2. This identifies possible subtrees to scan. Oracle knows that these subtrees are ordered by age. So the third and fourth leaf blocks [L3 – L4] are scanned and some values are retrieved. By looking at the fourth entry in the first branch block [B1], the server determines that it is no longer possible to find an F2x entry. Thus, it is not necessary to scan that subtree [L5].

The same process continues with the right part of this index. Note that out of a total of 10 leaf blocks, only five are scanned.

Index Scan (Skip) (2)

Index on (DEPTNO, SAL)



```
select /*+ index_ss(EMP IX_SS) */ * from emp where SAL < 1500;
```

Id	Operation	Name	A-Rows	Buffers
1	TABLE ACCESS BY INDEX ROWID	EMP	6	4
* 2	INDEX SKIP SCAN	IX_SS	6	2

```
2 - access("SAL"<1500)
    filter("SAL"<1500)
```

ORACLE

Index Skip Scan - Example

Find employees who have salary less than 1500 using index

There is a concatenated index on deptno and sal

Query does not have predicate on leading column (deptno).

This leading column only has some discrete values (deptno 10, 20 and 30).

So Oracle will pretend the index is really 3 little index structures hidden inside one big one. In example, it would be 3 index structures :

where deptno = 10

where deptno = 20

where deptno = 30

Output is ordered by deptno.

```
SQL> create index IX_SS on EMP(DEPTNO,SAL);
SQL> select /*+ index_ss(EMP IX_SS) */ * from emp where SAL< 1500;
```

EMPNO	ENAME	JOB	HIREDATE	SAL	DEPTNO
7934	MILLER	CLERK	23-JAN-82	1300	10
7369	SMITH	CLERK	17-DEC-80	800	20
7876	ADAMS	CLERK	23-MAY-87	1100	20
7900	JAMES	CLERK	03-DEC-81	950	30
7521	WARD	SALESMAN	22-FEB-81	1250	30
7654	MARTIN	SALESMAN	28-SEP-81	1250	30

Id	Operation	Name	A-Rows	Buffers
1	TABLE ACCESS BY INDEX ROWID	EMP	6	4
* 2	INDEX SKIP SCAN	IX_SS	6	2

Predicate Information (identified by operation id):

```
2 -access("SAL"<1500)
   filter("SAL"<1500)
```

1.8 - Index Scan (Index Join)

```
create index I_ENAME on EMP(ename)
create index I_SAL on EMP(sal)

select /*+ INDEX_JOIN(e) */ ename, sal from emp e;
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		409	66
1	VIEW	index\$_join\$_001	409	66
* 2	HASH JOIN		409	
3	INDEX FAST FULL SCAN	I_ENAME	409	33
4	INDEX FAST FULL SCAN	I_SAL	409	33

ORACLE

Index Scan (Index Join)

An *index join* is a hash join of several indexes that together contain all the table columns that are referenced in the query. If an index join is used, then no table access is needed because all the relevant column values can be retrieved from the indexes. An index join cannot be used to eliminate a sort operation.

```

alter table emp modify (SAL not null, ENAME not null);
create index I_SAL on emp(SAL);
create index I_ENAME on emp(ENAME);
select /*+ INDEX_JOIN(e) */ ename, sal from emp e;

```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		409	66
1	VIEW	index\$_join\$_001	409	66
* 2	HASH JOIN		409	
3	INDEX FAST FULL SCAN	I_ENAME	409	33
4	INDEX FAST FULL SCAN	I_SAL	409	33

Predicate Information (identified by operation id):

```

2 -access(indexjoin$_alias$_003.ROWID=indexjoin$_alias$_002.ROWID)

```

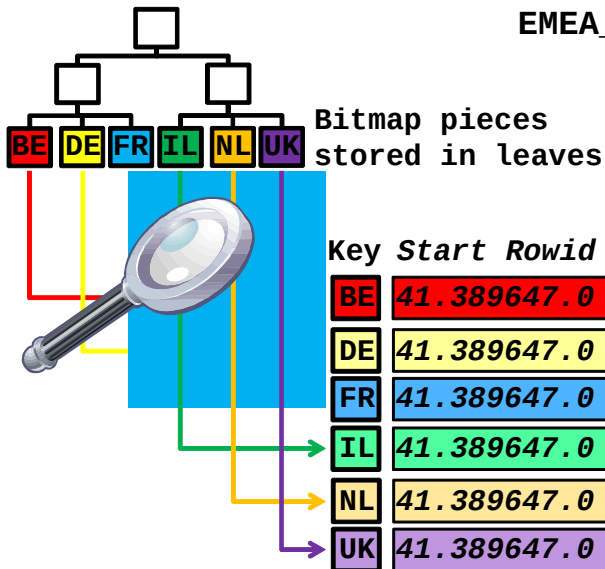
The index join is not a real join operation (note that the example is a single table query) but is built using index accesses followed by a join operation on rowid. The above example assumes you have two separate indexes on the ENAME and SAL columns of the EMP table.

Bitmap Indexes

Index Bitmap IX_B2
stored in B*Tree Index

Table
EMEA_PERF

ID	Name	G	C	Rowid
1	Jan Boonen	M	BE	41.389647.0
2	Andreas Walter	M	DE	41.389647.1
3	Lech Barwicki	M	FR	41.389647.2
4	Muriel FRY	F	FR	41.389647.3
5	Christian Boure	M	FR	41.389647.4
6	Thomas Guilleminot	M	FR	41.389647.5
7	Dena Feinblat	F	IL	41.389647.6
8	Sijmen de Roo	M	NL	41.389647.7
9	Doody Atraktsi	M	IL	41.389647.8
10	James cremonini	M	UK	41.389648.0
11	Steve Dixon	M	UK	41.389648.1
12	Andrew Hambelton	M	UK	41.389648.2



ORACLE

Bitmap indexes:

In a B*Tree, there is a one-to-one relationship between an index entry and a row: an index entry points to a row. A bitmap index is organized as a B*tree index but, with bitmap indexes, a single index entry uses a bitmap to point to many rows simultaneously.

- 1 leaf contains Key, Start Rowid, End Rowid, Bitmap
- Index Leaf blocks are sorted by key then Start Rowid

By example Bitmap with KEY = 'FR' points to rows with rowid (see full DDL and DML next slide)

- 41.389647.2 (File 41 – Block 389647 Row 2)
- 41.389647.3 (File 41 – Block 389647 Row 3)
- 41.389647.4 (File 41 – Block 389647 Row 4)
- 41.389647.5 (File 41 – Block 389647 Row 5)

If a bitmap index involves more than one column, then there is a bitmap for every possible combination.

Each bitmap header stores *start* and *end* ROWIDs. Based on these values, Oracle uses an internal

algorithm to map bitmaps onto ROWIDs. Each position in a bitmap maps to a row in the table. The contents of that position in the bitmap for a particular value indicate whether that row has that value in the bitmap. The value stored is 1 if the row values match the bitmap condition; otherwise it is 0. There is a compression algorithm to store bitmaps (0 are compressed, not 1).

Bitmap Indexes :

- Every bitmap piece has a start ROWID and an end ROWID
- Each indexed column value may have one or more bitmap pieces
- A bitmap piece
 - covers a contiguous set of rows in one or more extents
 - may cross extent boundaries
 - may split within a block
- Branch block format and characteristics are identical to B*Tree indexes
 - No SKIP scan available
- must be non-unique
- include NULL values (B*Tree indexes do not)
- Maximum 30 columns
- Can be locally partitioned (8.1 and above)
- Cannot be globally partitioned
- Can be function-based indexes
- Can be IOT secondary indexes (9.2 and above)
- Can be created on global temporary tables
- Cannot be compressed
- Cannot be reversed

Bitmap can be combined using the logical operations AND, OR, NOT

BITMAP MINUS		Subtracts bits of one bitmap from another. Row source is used for negated predicates. Can be used only if there are nonnegated predicates yielding a bitmap from which the subtraction can take place.
BITMAP OR		Computes the bitwise OR of two bitmaps.
BITMAP AND		Computes the bitwise AND of two bitmaps.

A	B	A AND B	A OR B	NOT A
0	0	0	0	1
0	1	0	1	1

1	0	0	1	0
1	1	1	1	0

Minus Operation is equivalent to A AND NOT B

Performance consideration : Locking

- When a bitmap index is updated, a row lock is taken for the index leaf entry
- The index leaf entry contains a bitmap which may cover many rows across a number of blocks
- No other transaction can update an indexed column in any row covered by that bitmap piece until the original transaction commits or rolls back
- Bitmap indexes are designed for a read only/read mostly environment without any single row DML operations (only bulk loads).
- Bitmap indexes are not appropriate in a transactional system at all. A single bitmap key points to *many* rows in the base table.

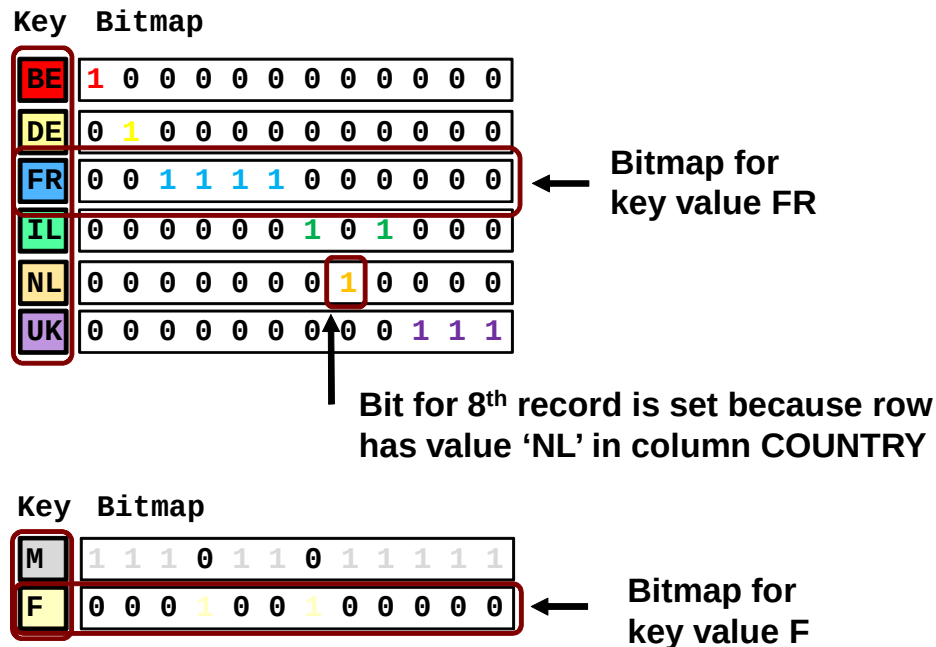
A good introduction about bitmap indexes is available in **Note 70067.1** and **Note 1043819.6**

More details about internal storage can be found in **Note 62529.1**

Interesting External links

- <http://www.dbazine.com/oracle/or-articles/jlewis3>
- <http://www.juliandyke.com/Presentations/BitmapIndexInternals.ppt>

Bitmap Indexes Example



ORACLE

Bitmap Indexes Example

Each bitmap is corresponding to a particular value of the columns involved. In the example, if the COUNTRY column has 6 distinct values, then 6 bitmaps are built. If another COUNTRY value is later added, a new bitmap for that value is created.

In the bitmap for COUNTRY 'BE', the first position contains the value 1, showing that the first row has that value. In the bitmap for COUNTRY 'UK', the first position contains the value 0 because the first row does not have that value. Each row maps to a position in the bitmaps, and the row matching the bitmap's value will contain the digit 1 in that position.

```
create table EMEA_Perf_TEAM_Jan_2007 (
  ID          NUMBER(6),
  NAME        VARCHAR2(50),
  GENDER      VARCHAR2(1),
  COUNTRY     VARCHAR2(2),
  TXT         CHAR(100)) pctfree 85;
```

```

insert into PERF values ( 1 , 'Jan Boonen      ', 'M', 'BE', 'NONE');
insert into PERF values ( 2 , 'Andreas Walter  ', 'M', 'DE', 'NONE');
insert into PERF values ( 3 , 'Lech Barwicki   ', 'M', 'FR', 'NONE');
insert into PERF values ( 4 , 'Muriel FRY      ', 'F', 'FR', 'NONE');
insert into PERF values ( 5 , 'Christian Boure   ', 'M', 'FR', 'NONE');
insert into PERF values ( 6 , 'Thomas Guilleminot', 'M', 'FR', 'NONE');
insert into PERF values ( 7 , 'Dena Feinblat     ', 'F', 'IL', 'NONE');
insert into PERF values ( 8 , 'Sijmen de Roo     ', 'M', 'NL', 'NONE');
insert into PERF values ( 9 , 'Doody Atraktsi    ', 'M', 'IL', 'NONE');
insert into PERF values ( 10 , 'James Cremonini   ', 'M', 'UK', 'NONE');
insert into PERF values ( 11 , 'Steve Dixon       ', 'M', 'UK', 'NONE');
insert into PERF values ( 12 , 'Andrew Hambelton  ', 'M', 'UK', 'NONE');
commit;

```

```

select dbms_rowid.rowid_relative_fno(rowid)||'.'||
       dbms_rowid.rowid_block_number(rowid)||'.'||
       dbms_rowid.rowid_row_number(rowid) as "File.Block.Row"
, id, name, gender, country from perf order by ID;

```

File.Block.Row	ID	NAME	G	CO
41.389647.0	1	Jan Boonen	M	BE
41.389647.1	2	Andreas Walter	M	DE
41.389647.2	3	Lech Barwicki	M	FR
41.389647.3	4	Muriel FRY	F	FR
41.389647.4	5	Christian Boure	M	FR
41.389647.5	6	Thomas Guilleminot	M	FR
41.389647.6	7	Dena Feinblat	F	IL
41.389647.7	8	Sijmen de Roo	M	NL
41.389647.8	9	Doody Atraktsi	M	IL
41.389648.0	10	James Cremonini	M	UK
41.389648.1	11	Steve Dixon	M	UK
41.389648.2	12	Andrew Hambelton	M	UK

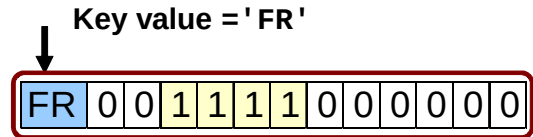
```

create bitmap index IX_B1 on EMEA_PERF_TEAM_Jan_2007(GENDER);
create bitmap index IX_B2 on EMEA_PERF_TEAM_Jan_2007(COUNTRY);

```

1.9 – Bitmap Index Access

```
SQL> SELECT *
      2 FROM EMEA_PERF_TEAM T
      3 WHERE country = 'FR';
```



```
SQL> SELECT * FROM EMEA_PERF_TEAM T WHERE country = 'FR';
```

Id	Operation	Name	Rows	Cost (%CPU)
0	SELECT STATEMENT			1 (100)
1	TABLE ACCESS BY INDEX ROWID	EMEA_PERF_TEAM	4	1 (0)
2	BITMAP CONVERSION TO ROWIDS			
* 3	BITMAP INDEX SINGLE VALUE	IX_B2		

Predicate Information (identified by operation id):

3 - access("COUNTRY"='FR')

ORACLE

Bitmap Index Access

This query scans the bitmap for country 'FR' for positions containing a 1. Positions with a 1 are converted into ROWID and their corresponding rows returned for the query.

Bitmap Index Operations

BITMAP CONVERSION	TO ROWIDS	converts bitmap representations to actual rowids that can be used to access the table.
	FROM ROWIDS	converts the rowids to a bitmap representation
	COUNT	returns the number of rowids if the actual values are not needed.
BITMAP INDEX	SINGLE VALUE	looks up the bitmap for a single key value in the index.

	RANGE SCAN	retrieves bitmaps for a key value range.
	FULL SCAN	performs a full scan of a bitmap index if there is no start or stop key.
BITMAP MERGE		Merges several bitmaps resulting from a range scan into one bitmap.
BITMAP KEY ITERATION		Takes each row from a table row source and finds the corresponding bitmap from a bitmap index. This set of bitmaps are then merged into one bitmap in a following BITMAP MERGE operation.

This query retrieves bitmaps for a key value range. Bitmap are stored in B*Tree structure, but INDEX SKIP SCAN operation is not available.

```
SQL> select /*+ index(T IX_B2) */ *
      from EMEA_PERF_TEAM_Jan_2007 T
      where country > 'FR';
```

```
ID NAME          G CO
--
 7 Dena Feinblat  F IL
 9 Doody Atraktsi M IL
 8 Sijmen de Roo  M NL
10 James Cremonini M UK
11 Steve Dixon    M UK
12 Andrew Hambelton M UK
```

Id	Operation	Name	A-Rows	Buffers
1	TABLE ACCESS BY INDEX ROWID	EMEA_PERF_TEAM	6	4
2	BITMAP CONVERSION TO ROWIDS		6	2
* 3	BITMAP INDEX RANGE SCAN	IX_B2	3	2

Predicate Information (identified by operation id):

```
3 - access("COUNTRY">'FR')
    filter("COUNTRY">'FR')
```

In some cases (such as a query counting the number of rows with COUNTRY >'FR'), the query simply uses the bitmap itself and count the number of 1s (not needing the actual table rows).

```
SQL> select /*+ index(T IX_B2) */ count(*)
      from EMEA_PERF_TEAM_Jan_2007 T
      where country > 'FR';
```

```
      COUNT(*)
-----
          6
```

Id	Operation	Name	A-Rows	Buffers	
1	TABLE ACCESS BY INDEX ROWID	EMEA_PERF_TEAM	1	5	
2	BITMAP CONVERSION COUNT		3	5	
* 3	BITMAP INDEX RANGE SCAN	IX_B2	3	5	

Predicate Information (identified by operation id):

```
-----
3 - access("COUNTRY">'FR')
    filter("COUNTRY">'FR')
```

1.10 - Combining Bitmap Indexes (1)

```
SQL> SELECT *
2 FROM EMEA_PERF_TEAM T
3 WHERE country = 'FR' or gender = 'F';
```

FR 0 0 1 1 1 1 0 0 0 0 0 0

OR

F 0 0 0 1 0 0 1 0 1 0 0 0

0 0 1 1 1 1 1 0 1 0 0 0

1	TABLE ACCESS BY INDEX ROWID	EMEA_PERF_TEAM
2	BITMAP CONVERSION TO ROWIDS	
3	BITMAP OR	
*4	BITMAP INDEX SINGLE VALUE	IX_B1
*5	BITMAP INDEX SINGLE VALUE	IX_B2

4 - access("GENDER"='F')
5 - access("COUNTRY"='FR')

ORACLE

Combining Bitmap Indexes

Due to fast bit-and, bit-minus, and bit-or operations, bitmap indexes are efficient:

- When using IN (value_list)
- When predicates are combined with AND/OR

Bitmap indexes can be used efficiently when a query combines several possible values for a column or when two separately indexed columns are used.

In some cases, a WHERE clause might reference several separately indexed columns:

If both the COUNTRY and GENDER columns have bitmap indexes, a bit-and operation on the two bitmaps would quickly locate the rows that you are looking for. The more complex the compound WHERE clauses become, the more you benefit from bitmap indexing.

Example

Below SQL is showing we can use 2 Bitmap indexes to perform bitmap operation

```
SELECT * FROM EMEA_PERF_TEAM T
WHERE country = 'FR'
      or gender = 'F';
```

Id	Operation	Name	E-Rows	Cost(%CPU)

1	TABLE ACCESS BY INDEX ROWID	EMEA_PERF_TEAM	11	2 (0)
2	BITMAP CONVERSION TO ROWIDS			
3	BITMAP OR			
* 4	BITMAP INDEX SINGLE VALUE	IX_B2		
* 5	BITMAP INDEX SINGLE VALUE	IX_B1		

4 - access("COUNTRY"='FR')				
5 - access("GENDER"='F')				

ID	NAME	G	CO

7	Dena Feinblat	F	IL
5	Christian Boure	M	FR
6	Thomas Guilleminot	M	FR
3	Lech Barwicki	M	FR
4	Muriel FRY	F	FR

```
SQL> SELECT *
      FROM EMEA_PERF_TEAM T
      WHERE country in ('FR','DE');
```

ID	NAME	G	CO

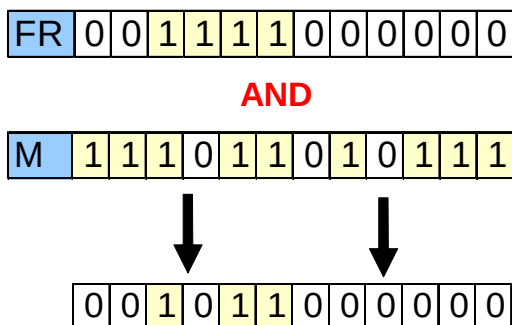
2	Andreas Walter	M	DE
5	Christian Boure	M	FR
6	Thomas Guilleminot	M	FR
3	Lech Barwicki	M	FR
4	Muriel FRY	F	FR

Id	Operation	Name	Rows	Cost(%CPU)
1	INLIST ITERATOR			
2	TABLE ACCESS BY INDEX ROWID	EMEA_PERF_TEAM	5	2 (0)
3	BITMAP CONVERSION TO ROWIDS			
* 4	BITMAP INDEX SINGLE VALUE	IX_B2		

4 - access(("COUNTRY"='DE' OR "COUNTRY"='FR'))

Combining Bitmap Indexes (2)

```
SQL> SELECT *
2 FROM EMEA_PERF_TEAM T
3 WHERE country = 'FR' and gender = 'M';
```



1	TABLE ACCESS BY INDEX ROWID	EMEA_PERF_TEAM
2	BITMAP CONVERSION TO ROWIDS	
3	BITMAP AND	
*4	BITMAP INDEX SINGLE VALUE	IX_B1
*5	BITMAP INDEX SINGLE VALUE	IX_B2

4 - access("GENDER"='M')

5 - access("COUNTRY"='FR')

ORACLE

Combining Bitmap Indexes (2)

Example

```
SQL> select /*+ index_combine(T IX_B1 IX_B2) */ *
from EMEA_PERF_TEAM T
where country = 'FR'
and gender = 'M';
```

ID	NAME	G	CO
5	Christian Boure	M	FR
6	Thomas Guilleminot	M	FR
3	Lech Barwicki	M	FR

Id	Operation	Name	Rows	Cost%CPU)
1	TABLE ACCESS BY INDEX ROWID	EMEA_PERF_TEAM	3	2 (0)
2	BITMAP CONVERSION TO ROWIDS			
3	BITMAP AND			
* 4	BITMAP INDEX SINGLE VALUE	IX_B1		
* 5	BITMAP INDEX SINGLE VALUE	IX_B2		

4 - access(GENDER = 'M')

5 - access(COUNTRY = 'FR')



Bitmap Operations

- BITMAP CONVERSION
 - Can be TO or FROM ROWIDS
- BITMAP INDEX
 - Single value lookup
 - Range scan
 - Full scan
- BITMAP MERGE
- BITMAP AND / OR
- BITMAP MINUS

Id Operation	
0	SELECT STATEMENT
1	TABLE ACCESS BY INDEX ROWID
2	BITMAP CONVERSION TO ROWIDS
3	BITMAP MINUS
4	BITMAP OR
* 5	BITMAP INDEX SINGLE VALUE
* 6	BITMAP INDEX SINGLE VALUE
* 7	BITMAP INDEX SINGLE VALUE

ORACLE

Bitmap Operations

We already explained some of the possible bitmap operations. The slide summarizes all possibilities

BITMAP CONVERSION	TO ROWIDS	converts bitmap representations to actual rowids that can be used to access the table.
	FROM ROWIDS	converts the rowids to a bitmap representation
	COUNT	returns the number of rowids if the actual values are not needed.
BITMAP INDEX	SINGLE VALUE	looks up the bitmap for a single key value in the index.
	RANGE SCAN	retrieves bitmaps for a key value range.
	FULL SCAN	performs a full scan of a bitmap index if there is no start or stop key.

BITMAP MINUS		Subtracts bits of one bitmap from another. Row source is used for negated predicates. Can be used only if there are nonnegated predicates yielding a bitmap from which the subtraction can take place.
BITMAP OR		Computes the bitwise OR of two bitmaps.
BITMAP AND		Computes the bitwise AND of two bitmaps.
BITMAP MERGE		Merges several bitmaps resulting from a range scan into one bitmap.
BITMAP KEY ITERATION		Takes each row from a table row source and finds the corresponding bitmap from a bitmap index. This set of bitmaps are then merged into one bitmap in a following BITMAP MERGE operation.

Example

```
SQL> select /*+ index (T IX_B1 IX_B2) */ *
from EMEA_PERF_TEAM T
where country in ('FR','DE')
and gender is not null;
```

ID	NAME	G	CO
5	Christian Boure	M	FR
6	Thomas Guilleminot	M	FR
2	Andreas Walter	M	DE
3	Lech Barwicki	M	FR
4	Muriel FRY	F	FR

Id	Operation	Name	Rows	Cost (%CPU)
0	SELECT STATEMENT		3	(100)
1	TABLE ACCESS BY INDEX ROWID	EMEA_PERF_TEAM	5	3 (0)
2	BITMAP CONVERSION TO ROWIDS			
3	BITMAP MINUS			
4	BITMAP OR			
* 5	BITMAP INDEX SINGLE VALUE	IX_B2		
* 6	BITMAP INDEX SINGLE VALUE	IX_B2		
* 7	BITMAP INDEX SINGLE VALUE	IX_B1		

Predicate Information (identified by operation id):

- 5 - access("COUNTRY"='DE')
- 6 - access("COUNTRY"='FR')
- 7 - access("GENDER" IS NULL)

BITMAP MINUS -> Oracle takes the second bitmap and negates it—changing ones to zeros, and zeros to ones. The **bitmap minus** operation can then be performed as a **bitmap and** using this negated bitmap.

BITMAP CONVERSION FROM ROWID -> BTREE index converted on the fly by optimizer into BITMAP (cost is lower than other methods) to make available these efficient bitmaps comparison operations. Once the bitmap comparison has been done then the resultant bitmap is converted back into rowids (BITMAP CONVERSION TO ROWIDS) to perform the data lookup



2 - Other Unary Operations

- SORT & HASH (GROUP BY – UNIQUE)
- BUFFER (SORT)
- IN LIST ITERATOR
- VIEW
- COUNT STOP KEY
- (MIN/MAX) / FIRST ROW

ORACLE

Unary Operations

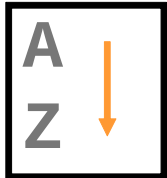
1.1 - Access Path

1.2 - Other Unary Operations

Unary Operation : operations that act on only 1 input

2.1 – SORT & HASH (1)

- **SORT**



- AGGREGATE: Single row from group function
- UNIQUE: To eliminate duplicates
- JOIN: Precedes a merge join
- GROUP BY, ORDER BY: For these operators

- **HASH**

- UNIQUE
- GROUP BY

ORACLE

SORT & HASH (1)

Sort operations result when users specify an operation that requires a sort. Commonly encountered operations include the following:

- **SORT UNIQUE**
It sort output rows to remove duplicates. It occurs if a user specifies a DISTINCT clause or if an operation requires unique values for the next step.
- **SORT AGGREGATE**
does not actually involve a sort. It is used when aggregates are being computed across the whole set of rows.
- **SORT GROUP BY**
is used when aggregates are being computed for different groups in the data. The sort is required to separate the rows into different groups.
- **GROUP BY (NO SORT)**
Same operation as above, rows are already sorted by an index.
- **SORT JOIN**
happens during a sort-merge join if the rows need to be sorted by the join key.
- **SORT ORDER BY**

is required when the statement specifies an ORDER BY that cannot be satisfied by one of the indexes.

- **HASH GROUP BY**

Operation hashing a set of rows into groups for a query with a GROUP BY clause.

- **HASH UNIQUE**

Operation hashing a set of rows to eliminate Duplicates. It occurs if a user specifies a DISTINCT clause or if an operation requires unique values for the next step. It is available since 10.2 if "_gby_hash_aggregation_enabled" = TRUE else SORT UNIQUE is used.

SORT generally means exactly what it says, but notice that operations like COUNT and MIN are shown as a SORT (AGGREGATE).

Note that several SQL language components cause implicit sorts (or hash since 10.2), such as DISTINCT, GROUP BY, UNION, MINUS, and INTERSECT.

on OTN : [TWP_general_sort_performance_10gr2_0605.pdf](#) Sort Performance Improvements in Oracle Database 10g Release 2

SORT & HASH (2)

Don't rely on the GROUP BY or DISTINCT to ORDER BY returned rows.

If you want to have rows ordered, use ORDER BY clause.



ORACLE

SORT & HASH(2)

Bug.4715937 (92) SELECT DISTINCT NOT RETURNING RESULTS IN SORTED ORDER IN 10.2.0.1.0

HASH UNIQUE & SORT UNIQUE Example (10.2)

```
CREATE TABLE t1 (c1 NUMBER);
begin
for i in 1..100 loop
insert into T1 values(mod(i,3));
end loop;
end;

/

! alter session set
! "_gby_hash_aggregation_enabled"
! = FALSE;
```

SELECT DISTINCT (c1) FROM t1;					!	SELECT DISTINCT (c1) FROM t1;					!
C1					!	C1					!
-----					!	-----					!
1					!	0					!
2					!	1					!
0					!	2					!
-----					!	-----					!
Id	Operation	Name	Rows		!	Id	Operation	Name	Rows		!
0	SELECT STATEMENT				!	0	SELECT STATEMENT				!
1	HASH UNIQUE		100		!	1	SORT UNIQUE		100		!
2	TABLE ACCESS FULL	T1	100		!	2	TABLE ACCESS FULL	T1	100		!
-----					!	-----					!

2.2 – Buffer (sort)

- **BUFFER (SORT)**

```
select ename, emp.deptno, dept.deptno, dname
from emp, dept
where ename like 'A%';
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		21	10
1	MERGE JOIN CARTESIAN		21	10
* 2	TABLE ACCESS FULL	EMP	4	2
3	BUFFER SORT		5	8
4	TABLE ACCESS FULL	DEPT	5	2

2 - filter("EMP"."ENAME" LIKE 'A%')

ORACLE

BUFFER SORT

The BUFFER SORT operator uses a temporary table or a sort area in memory to store intermediate data. However, the data is not necessarily sorted. - we just use the same method of storing as sorting does (in memory (PGA) or temp table).

This is needed if we have an operation that needs all the input data before it can start (See Merge Join Cartesian Slide).

So buffer sort is using the buffering mechanism of a traditional sort, but it is not really doing the sort itself. Oracle is simply buffering the data (in UGA – PGA if dedicated) to avoid multiple tablescans against real data blocks.

The whole “sort” mechanism is re-used, eg including the swap to disk when not enough sort area memory is available but without sorting the data.

What’s the difference between a temporary table and a buffer sort ?

- temporary table - in SGA

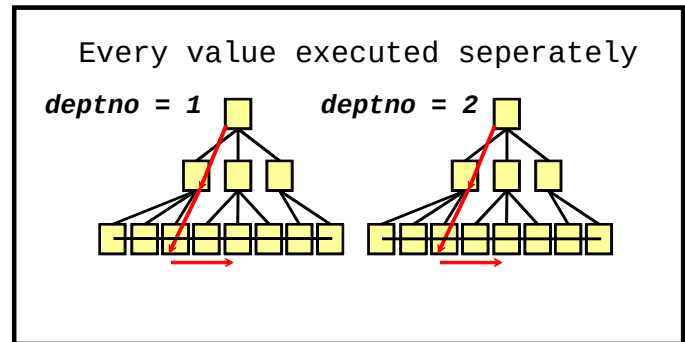
- buffer sort - in UGA

Both of these are “swapable” in the temporary files. The biggest difference is that the temporary table has to go through the db cache, which introduces latching effects that don’t appear for buffer sorts.

2.3 – Inlist Iterator

```
create index I_DEPTNO
on emp(deptno);

select * from emp
where deptno in (1,2);
select * from emp
where deptno = 1
   or deptno =2 ;
```



Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		5	2
1	INLIST ITERATOR			
2	TABLE ACCESS BY INDEX ROWID	EMP	5	2
* 3	INDEX RANGE SCAN	I_DEPTNO	5	1

Predicate Information (identified by operation id):

3 - access(DEPTNO=1 OR DEPTNO=2)

ORACLE

IN-LIST ITERATOR

The **IN-list iterator** is used when a query contains an **IN** clause with values.

"in" is the same as a list of ORs on one column.

Iteration over enumerated value-list : Every value is executed separately. Inlist iterator will also apply to such a list of Ors.

Inlist iterator is sort of "for loop", eg: deptno in (1,2) might be doing 2 index probes in "a loop"

The execution plan is identical to what would result for a statement with an equality clause instead of **IN** except for one additional step. That extra step occurs when the **IN-list iterator** feeds the equality clause with unique values from the **IN-list**.

Alternative handling is **UNION ALL** of each value or a **FILTER** of the values against all the rows.

The optimizer uses an **IN-list iterator** when an **IN** clause is specified with values, and the optimizer finds a selective index for that column. If there are multiple **OR** clauses using the same index, then the

optimizer chooses this operation rather than **CONCATENATION** or **UNION ALL**, because it is more efficient.

There are no hints for this operation.

Event **10157**, **10181** for not using in-list. .

Event **10157**, 00000, "CBO **disable** index access path for **in-list**"

Event **10181**, 00000, "CBO **disable** multi-column **in-list** processing"

```
alter session set events '10157 trace name context forever, level 1'
```

```
alter session set events '10181 trace name context forever, level 1'
```

2.4 – View (1)

```
create view V as select /*+ NO_MERGE */ DEPTNO, sal
from emp ;

select * from V;
```

Id	Operation	Name	Rows	Cost (%CPU)
0	SELECT STATEMENT			3 (100)
1	VIEW	V	14	3 (0)
2	TABLE ACCESS FULL	EMP	14	3 (0)

ORACLE

VIEW (1)

Each query produces a variable set of data in the form of a table. A view simply gives a name to this set of data.

Hint NO_MERGE in create View statement is there just to illustrate the VIEW Keyword. Without that hint, View is MERGED and VIEW keyword is not in plan

When Views are referenced in a query they can be handled in 2 ways by Oracle. If a number of conditions are met they can be merged in to the main query (which means that the view text is re-written as a join with the other tables in the query) or they can be left as standalone views and selected from directly as with a table (predicates can also be pushed into or pulled out of the views as long as certain conditions are met)

When a view is **NOT** merged, we can see **VIEW** operation.

View operation is executed separately

<http://jonathanlewis.wordpress.com/2009/06/25/explain-view/>

Usually, when issuing a SELECT statement against a view, the defining query of the view is **MERGED** into the SELECT statement, and they are parsed as a single statement.

Example :

VIEW VM is MERGED and it does not appear in plan.

```
create view VM as
select DEPTNO, sal
from emp ;

select vm.deptno,d.dname,vm.sal
  from VM, dept d
 where vm.deptno = d.deptno
```

```
DEPTNO DNAME                SAL
-----
10 ACCOUNTING              2450
...
```

Id	Operation	Name	Rows
0	SELECT STATEMENT		
1	MERGE JOIN		14
2	TABLE ACCESS BY INDEX ROWID	DEPT	4
3	INDEX FULL SCAN	PK_DEPT	4
* 4	SORT JOIN		14
5	TABLE ACCESS FULL	EMP	14

Predicate Information (identified by operation id):

```
4 - access("DEPTNO"="D"."DEPTNO")
    filter("DEPTNO"="D"."DEPTNO")
```



View (2)

- View operation is executed separately
 - All rows from View are returned (Id=2)
 - This resultset is then joined to DEPT with nested loops

Id	Operation	Name	Rows	Cost (%CPU)
0	SELECT STATEMENT			7 (100)
1	NESTED LOOPS		3	7 (15)
2	VIEW		3	4 (25)
3	HASH GROUP BY		3	4 (25)
4	TABLE ACCESS FULL	EMP	14	3 (0)
5	TABLE ACCESS BY INDEX ROWID	DEPT	1	1 (0)
* 6	INDEX UNIQUE SCAN	PK_DEPT	1	1 (0)

Predicate Information (identified by operation id):

6 - access("V"."DEPTNO"="D"."DEPTNO")

ORACLE

VIEW (2)

Sometimes a view is "**NON-MERGEABLE**" and must be executed independently (in a separate query block). In this case, we see operation **VIEW**" in the explain plan. The VIEW keyword indicates that the view will be executed as a separate query block.

For example, views containing GROUP BY functions are non-mergeable.

create a VIEW based on EMP table to display SUM of employees salary per departement.

create view V as select DEPTNO,sum(sal) SUM_SAL from emp group by deptno;

```
SQL> select * from V;
```

DEPTNO	SUM_SAL
30	9400
20	10875

Id	Operation	Name	Rows	Cost (%CPU)
0	SELECT STATEMENT			4 (100)
1	VIEW	V	3	4 (25)
2	HASH GROUP BY		3	4 (25)
3	TABLE ACCESS FULL	EMP	14	3 (0)

INLINE VIEW : a view definition within the from clause of a query (V is inline view)

```
SQL> select v.*,d.dname
       from (select DEPTNO, sum(sal) SUM_SAL
             from emp
             group by deptno) v, dept d
       where v.deptno=d.deptno;
```

DEPTNO	SUM_SAL	DNAME
30	9400	SALES
20	10875	RESEARCH
10	8750	ACCOUNTING

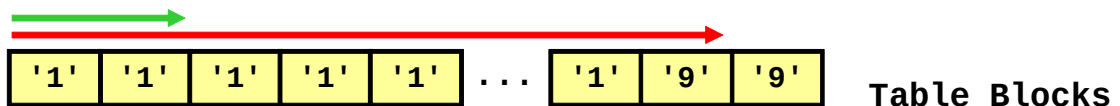
2.5 – Count Stop Key

- **COUNT STOPKEY**

- rownum < ~value~
- STOP previous operation when count is reached

- **EXAMPLE**

- Fill a table with plenty of '1' and only some '9' at the end.
- Run a query to output the ten first '1'
- Run same query with the ten first '9'
-  If the value in question is frequently occurring (e.g. '1') then only a small number of buffers are required (Buffers = 12)
-  If the value in question is NOT frequently occurring (e.g. '9'), then almost the whole table needs to be retrieved (Buffers = 3984)



ORACLE

COUNT STOPKEY

Count operation where the number of rows returned is limited by the ROWNUM expression in the WHERE clause. Count stop key terminates the current operation when condition (count) is reached.

Example

Script to create and load data into Table + Actual plans including IOSTATS :

```
create table tc (C1 char(1000), c2 char(1000), c3 char(1000),c4
char(1000));
begin
for i in 1..4000 loop
insert into TC values (1,1,1,1);
end loop;
for i in 1..100 loop
insert into TC values (9,9,9,9);
```

```

end loop;
end;
/

```

Below are set of SQLs to show that the number of buffer gets depends of WHERE predicates.

How to generate such plan is explained in Part 2 : Interpreting The Plan

Buffers Column from Actual Plan is reporting Nb buffer gets.

```

select count(*)
  from (select /*+ NO_MERGE */ *
        from TC
        where C1 ='1' and rownum < 10);

```

Id	Operation	Name	Starts	A-Rows	Buffers
1	SORT AGGREGATE		1	1	12
2	VIEW		1	9	12
* 3	COUNT STOPKEY		1	9	12
* 4	TABLE ACCESS FULL	TC	1	9	12

Predicate Information (identified by operation id):

3 - filter(ROWNUM<10)

4 - filter("C1"='1') - Only 12 Buffers to get 10 times value '1'

```

select count(*)
  from (select /*+ NO_MERGE */ *
        from TC
        where C1 ='9'
        and rownum < 10);

```

Id	Operation	Name	Starts	A-Rows	Buffers
1	SORT AGGREGATE		1	1	3984
2	VIEW		1	9	3984
* 3	COUNT STOPKEY		1	9	3984

* 4	TABLE ACCESS FULL TC		1	9	3984

Predicate Information (identified by operation id):

3 - filter(ROWNUM<10)

4 - filter("C1"='9') - Need 3984 Buffers to get 10 times value '9'

Only **12** buffer gets when C='1' and rownum < 10

But **3984** buffer gets when C='9' and rownum < 10

2.6 – (Min/Max) / First Row

- Stop accessing the data after first value is returned
- That's an optimization introduced on 8i
- Works with Index Range / Full scan

```
select min(id) from t;          -> MIN(ID) = 1
```

Id	Operation	Name	A-Rows	Buffers
1	SORT AGGREGATE		1	5
2	INDEX FULL SCAN (MIN/MAX)	T_IND	1	5

ORACLE

MIN/MAX First Rows

This is an optimisation to get Min/Max value. Amount of blocks read may vary with INDEX storage.

Buffers Column from Actual plan is Nb blocks get

Index T_IND size is over 3900 blocks with value from 1 to 1000000

```
select min(id) from t;          -> MIN(ID) = 1
```

Id	Operation	Name	A-Rows	Buffers
1	SORT AGGREGATE		1	5
2	INDEX FULL SCAN (MIN/MAX)	T_IND	1	5

```
delete t where id <50000;
select /* after delete */ min(id) from t;    -> MIN(ID) = 50000
```

Id	Operation	Name	A-Rows	Buffers
1	SORT AGGREGATE		1	141
2	INDEX FULL SCAN (MIN/MAX)	T_IND	1	141

```
alter index tt_ind coalesce;
select /* after coalesce */ min(id) from t;    -> MIN(ID) = 50000
```

Id	Operation	Name	A-Rows	Buffers
1	SORT AGGREGATE		1	3
2	INDEX FULL SCAN (MIN/MAX)	T_IND	1	3

```
select min(id) FROM t WHERE id > 500000;    -> MIN(ID)= 500001
```

Id	Operation	Name	A-Rows	Buffers
1	SORT AGGREGATE		1	3
2	FIRST ROW		1	3
* 3	INDEX RANGE SCAN (MIN/MAX)	TT_IND	1	3

Predicate Information (identified by operation id):

```
3 - access("ID"> 500000)
```



2.7 – Partition Access Path

- Only CBO seulement
- Partition (pruning)
 - compilation / execution
 - Depend on partition type
- Same Access Path for pruned partitions
- Partition wise join

ORACLE

PARTITION Access Path

2.7.1 – Partition Range (1)

- PARTITION RANGE SINGLE
 - WHERE n1 = 3 AND d1 = to_date('2007-07-19',...); → Pstart 31 Pstop 31
 - WHERE n1 = :n AND d1 = to_date(:d1,'yyyy-MM-DD'); → Key Key
- PARTITION RANGE INLIST
 - WHERE n1 in (1,3) AND d1 = to_date('2007-07-19',...); → Key(I) Key(I)
- PARTITION RANGE ITERATOR
 - WHERE n1 = 3 → 1 37
- PARTITION RANGE ALL
 - WHERE n3 > 3000; → 1 48
- PARTITION RANGE EMPTY
 - WHERE n1 = 5 AND d1 = to_date('2007-07-19',...); → INVALID INVALID
- PARTITION RANGE OR
 - WHERE n1 = 3 OR d1 = to_date('2007-07-19',...); → KEY(OR) KEY(OR)
- PARTITION RANGE SUBQUERY
 - WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 and tx.id=3; → KEY(SQ) KEY(SQ)

t & tx Range Partition (n1,d1)

n1 ↑	37	38	39	40	41	42	43	44	45	46	47	48	
	25	26	27	28	29	30	31	32	33	34	35	36	
	13	14	15	16	17	18	19	20	21	22	23	24	
	1	2	3	4	5	6	7	8	9	10	11	12	d1 →

PARTITION Range Scans

```
SQL> CREATE TABLE t (
  2   id NUMBER,
  3   d1 DATE,
  4   n1 NUMBER,
  5   n2 NUMBER,
  6   n3 NUMBER,
  7   pad VARCHAR2(4000),
  8   CONSTRAINT t_pk PRIMARY KEY (id)
  9 )
10 PARTITION BY RANGE (n1, d1) (
11   PARTITION t_1_jan_2007 VALUES LESS THAN (1, to_date('2007-02-01','yyyy-mm-dd')),
12   PARTITION t_1_feb_2007 VALUES LESS THAN (1, to_date('2007-03-01','yyyy-mm-dd')),
13   PARTITION t_1_mar_2007 VALUES LESS THAN (1, to_date('2007-04-01','yyyy-mm-dd')),
14   PARTITION t_1_apr_2007 VALUES LESS THAN (1, to_date('2007-05-01','yyyy-mm-dd')),
...
54   PARTITION t_4_aug_2007 VALUES LESS THAN (4, to_date('2007-09-01','yyyy-mm-dd')),
```

```

55     PARTITION t_4_sep_2007 VALUES LESS THAN (4, to_date('2007-10-01','yyyy-mm-dd')),
56     PARTITION t_4_oct_2007 VALUES LESS THAN (4, to_date('2007-11-01','yyyy-mm-dd')),
57     PARTITION t_4_nov_2007 VALUES LESS THAN (4, to_date('2007-12-01','yyyy-mm-dd')),
58     PARTITION t_4_dec_2007 VALUES LESS THAN (4, to_date('2008-01-01','yyyy-mm-dd'))
59     -- , PARTITION t_maxvalue VALUES LESS THAN (MAXVALUE, MAXVALUE)
60 );
SQL> execute dbms_random.seed(0)
SQL> INSERT INTO t
  2  SELECT rownum AS id,
  3         trunc(to_date('2007-01-01','yyyy-mm-dd')+rownum/27.4) AS d1,
  4         1+mod(rownum,4) AS n1,
  5         255+mod(trunc(dbms_random.normal*1000),255) AS n2,
  6         round(4515+dbms_random.normal*1234) AS n3,
  7         dbms_random.string('p',255) AS pad
  8  FROM dual
  9  CONNECT BY level <= 10000
10  ORDER BY dbms_random.value;
SQL> BEGIN
  2    dbms_stats.gather_table_stats(
  3        ownname      => user,
  4        tabname       => 'T',
  5        estimate_percent => 100,
  6        method_opt    => 'for all columns size skewonly',
  7        cascade       => TRUE
  8    );
  9  END;
10  /
SQL> SELECT partition_name, partition_position, num_rows
  2  FROM user_tab_partitions
  3  WHERE table_name = 'T'
  4  ORDER BY partition_position;

```

PARTITION_NAME	PARTITION_POSITION	NUM_ROWS
-----	-----	-----
T_1_JAN_2007	1	212
T_1_FEB_2007	2	192
T_1_MAR_2007	3	212
T_1_APR_2007	4	205
...		
T_4_SEP_2007	45	206
T_4_OCT_2007	46	212
T_4_NOV_2007	47	206
T_4_DEC_2007	48	212

```

SQL> CREATE TABLE tx AS SELECT * FROM t;

```

```
SQL> ALTER TABLE tx ADD CONSTRAINT tx_pk PRIMARY KEY (id);
SQL> BEGIN
  2   dbms_stats.gather_table_stats(
  3       ownname      => user,
  4       tabname       => 'TX'
  5   );
  6 END;
  7 /
```

SQL> REM SINGLE

The **WHERE** clause contains two restrictions: one for each column of the partition key. In this type of situation, the query optimizer recognizes that only a single partition contains relevant data. As a result, the operation **PARTITION RANGE SINGLE** appears in the execution plan. It is essential to understand that its child operation (**TABLE ACCESS FULL**) is not a full table scan over the whole table. Instead, only a single partition is accessed. This is confirmed by the value of the column **Starts** as well. Which partition is accessed is specified by the columns **Pstart** and **Pstop**.

```
SQL> SELECT * FROM t WHERE n1 = 3 AND d1 = to_date('2007-07-19','yyyy-mm-dd');
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows	

1	PARTITION RANGE SINGLE		1	7	31	31	7	
* 2	TABLE ACCESS FULL	T	1	7	31	31	7	

Predicate Information (identified by operation id):

```
-----
  2 - filter(("D1"=TO_DATE(' 2007-07-19 00:00:00', 'syyy-MM-dd hh24:mi:ss')
        AND "N1"=3))
```

Whenever a bind variable is used in a restriction, the query optimizer is no longer able to determine which partitions need to be accessed at parse time. In such cases, partition pruning is performed at runtime. The execution plan does not change, but the values of the columns **Pstart** and **Pstop** are set to **KEY**. This indicates that partition pruning occurs, but that at parse time, the query optimizer ignores which partition contains relevant data.

```
SQL> VARIABLE n1 NUMBER
SQL> EXECUTE :n1 := 3
SQL> VARIABLE d1 VARCHAR2(10)
SQL> EXECUTE :d1 := '2007-07-19'
SQL> SELECT * FROM t WHERE n1 = :n1 AND d1 = to_date(:d1,'yyyy-mm-dd');
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows	

1	PARTITION RANGE SINGLE		1	7	KEY	KEY	7	

```
|* 2 | TABLE ACCESS FULL | T | 1 | 7 | KEY | KEY | 7 |
```

 Predicate Information (identified by operation id):

```
2 - filter(("D1"=TO_DATE(:D1,'yyyy-mm-dd') AND "N1"=:N1))
```

It is possible to trace execution and so get information about partition which has been pruned :

```
alter session set events '10128 trace name context forever, level 2';
```

Must create kkpap table before to set event

```
DROP TABLE kkpap_pruning;
CREATE TABLE kkpap_pruning (
partition_count NUMBER,
iterator VARCHAR2(32),
partition_level VARCHAR2(32),
order_pt VARCHAR2(12),
call_time VARCHAR2(12),
part# NUMBER,
subp# NUMBER,
abs# NUMBER);
```

Note 166118.1 *Partition Pruning/Elimination*

SQL> REM INLIST

```
SQL> SELECT max(n3) FROM t WHERE n1 IN (1,3) AND d1 = to_date('2007-07-19','yyyy-mm-dd');
```

```
-----
| Id | Operation | Name | Starts | E-Rows | Pstart| Pstop | A-Rows |
-----
| 1 | SORT AGGREGATE | | 1 | 1 | | | 1 |
| 2 | PARTITION RANGE INLIST | | 1 | 14 | KEY(I | KEY(I | 13 |
|* 3 | TABLE ACCESS FULL | T | 2 | 14 | KEY(I | KEY(I | 13 |
-----
```

 Predicate Information (identified by operation id):

```
3 - filter(("D1"=TO_DATE(' 2007-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
INTERNAL_FUNCTION("N1")))
```

If you are working with Oracle9i, you should be very careful because the operation PARTITION RANGE SINGLE is not always shown as part of the execution plan. Consequently, the plan might be easily misunderstood. From this, it is important to display columns Pstart and Pstop, else you would think that no partition pruning takes place.

SQL> REM ITERATOR

If restriction uses a less-than condition (<) instead of being based on an equality condition (=) Thus, the operation becomes a PARTITION RANGE ITERATOR, and the columns Pstart and Pstop show the range of partitions that are accessed . In addition, the column Starts shows that operation 2 is executed only once, but operation 3 is executed once per partition. In other words, seven full partition scans are executed.

```
SELECT max(n3) FROM t WHERE n1 = 3
      AND d1 BETWEEN to_date('2007-03-06','yyyy-mm-dd')
                  AND to_date('2007-07-19','yyyy-mm-dd')
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION RANGE ITERATOR		1	941	27	31	932
* 3	TABLE ACCESS FULL	T	5	941	27	31	932

Predicate Information (identified by operation id):

```
3 - filter(("N1"=3
      AND "D1"<=TO_DATE(' 2007-07-19 00:00:00', 'yyyy-mm-dd hh24:mi:ss')
      AND "D1">=TO_DATE(' 2007-03-06 00:00:00', 'yyyy-mm-dd hh24:mi:ss')))
```

The operation PARTITION RANGE ITERATOR is also used when a restriction is based on the leading part of the partition key only. Note that partition 37 is accessed as well. This is because rows that have column n1 equal to 3 would be stored in that partition if column d1 had a value later than the 31st of January 2007.

```
SQL> SELECT max(n3) FROM t WHERE n1 = 3;
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION RANGE ITERATOR		1	2500	25	37	2500
* 3	TABLE ACCESS FULL	T	13	2500	25	37	2500

Predicate Information (identified by operation id):

```
3 - filter("N1"=3)
```

SQL> REM EMPTY

A particular operation, PARTITION RANGE EMPTY, appears in execution plans when the query optimizer recognizes that no partition is able to store processing-relevant data. For example, the following query is looking for data that has no partition where it could be stored (column n1 is out of

range). It is also important to note that not only are the columns Pstart and Pstop set to the value INVALID, but in addition, only operation 2 is executed (consuming no resource at all since, basically, it is a no-op operation).

```
SQL> SELECT max(n3) FROM t WHERE n1 = 5 AND d1 = to_date('2007-07-19','yyyy-mm-dd');
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION RANGE EMPTY		1	2	INVAL	INVAL	0
* 3	TABLE ACCESS FULL	T	0	2	INVAL	INVAL	0

Predicate Information (identified by operation id):

```
3 - filter(("D1"=TO_DATE(' 2007-07-19 00:00:00', 'yyyy-mm-dd hh24:mi:ss')
AND "N1"=5))
```

SQL> REM ALL

If no restriction is applied to the partition key, all partitions must be accessed. In such a case, the execution plan contains the operation PARTITION RANGE ALL, and the columns Starts, Pstart, and Pstop clearly show that all partitions are accessed.

```
SQL> SELECT max(n3) FROM t WHERE n3 BETWEEN 6000 AND 7000;
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION RANGE ALL		1	947	1	48	935
* 3	TABLE ACCESS FULL	T	48	947	1	48	935

Predicate Information (identified by operation id):

```
3 - filter(("N3">=6000 AND "N3"<=7000))
```

This very same execution plan is also used when inequalities are used as restrictions on the partition key. The following query is an example:

```
SQL> SELECT max(n3) FROM t WHERE n1 != 3 AND d1 != to_date('2007-07-19','yyyy-mm-dd');
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION RANGE ALL		1	7479	1	48	7480
* 3	TABLE ACCESS FULL	T	48	7479	1	48	7480

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

```
3 - filter(("N1"<>3
      AND "D1"<>TO_DATE(' 2007-07-19 00:00:00', 'yyyy-mm-dd hh24:mi:ss')))
```

Another case where the very same execution plan is used is when a restriction on the partition key is based on an expression or function. (to_char here)

```
SQL> SELECT max(n3) FROM t WHERE to_char(n1,'S9') = '+3'
      AND to_char(d1,'yyyy-mm-dd') = '2007-07-19';
```

```
-----
| Id | Operation                | Name | Starts | E-Rows | Pstart| Pstop | A-Rows |
-----
|  1 | SORT AGGREGATE           |      |       1 |      1 |      |      |      1 |
|  2 | PARTITION RANGE ALL      |      |       1 |     25 |      1 |     48 |       7 |
|*  3 | TABLE ACCESS FULL      | T    |       1 |     25 |      1 |     48 |       7 |
-----
```

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

```
3 - filter((TO_CHAR("N1")='+3'
      AND TO_CHAR(INTERNAL_FUNCTION("D1"),'yyyy-mm-dd')='2007-07-19'))
```

This means that to take advantage of partition pruning, not only should you have a restriction based on the partition key, but you should also not apply an expression or function to it. If applying an expression is a must, as of Oracle Database 11g, it is possible to choose a virtual column as the partition key.

SQL> REM PARTITION RANGE OR

Up to Oracle Database 10g Release 1, a WHERE clause that contains several predicates (on the partition key) that are *combined by OR conditions might lead to a full table scan*. In other words, the query optimizer is not always able to take advantage of partition pruning.

With 10g Release 2 onward, this limitation no longer applies. When this type of pruning is used, the operation PARTITION RANGE OR appears in the execution plan. Also note that the columns Pstart and Pstop are set to the value KEY(OR) . In the following example, according to the column Starts, 18 partitions are accessed. There are 18 because although the restriction applied to column n1 causes partitions 25 to 37 to be accessed, the restriction applied to column d1 causes partitions 1, 3, 15, 27, and 39 to be accessed (partition 1 is necessary in order to find out whether there are rows with column n1 containing values less than 1).

```
SQL> SELECT max(n3) FROM t WHERE n1 = 3 OR d1 = to_date('2007-03-06','yyyy-mm-dd')
```

```
-----
| Id | Operation                | Name | Starts | E-Rows | Pstart| Pstop | A-Rows |
-----
|  1 | SORT AGGREGATE           |      |       1 |      1 |      |      |      1 |
-----
```

	2		PARTITION RANGE OR		1		2521		KEY(OR		KEY(OR		2520			
*	3		TABLE ACCESS FULL		T		18		2521		KEY(OR		KEY(OR		2520	

Predicate Information (identified by operation id):

```
3 - filter(("N1"=3
          OR "D1"=TO_DATE(' 2007-03-06 00:00:00', 'yyyy-mm-dd hh24:mi:ss')))
```

SQL> REM MULTI-COLUMN

If the partition key is composed of several columns, it is important to observe what happens when a restriction is not defined for every column.

In earlier versions, the query optimizer uses partition pruning only when restrictions are applied to the leading part of the partition key. For example, if a partition key consists of the three columns c1, c2, and c3, partition pruning is possible only if a restriction on c1 or on both c1 and c2 is specified. This means that a restriction on both c2 and c3 does not lead to partition pruning.

This restriction has been removed in Oracle 10g Release 2 and was also back ported in the patchset 9.2.0.8 and 10.1.0.5.

The concept of multicolumn pruning is quite simple: independently of which columns a restriction is defined on, partition pruning always occurs.

The partition key of the test table is composed of two columns, the following query is an example with restriction applied to the second column. Note that the operation PARTITION RANGE MULTI-COLUMN and the value of the columns Pstart and Pstop confirm that partition pruning takes place; however, no information is provided about which partitions are accessed

```
SQL> SELECT max(n3) FROM t WHERE d1 = to_date('2007-07-19','yyyy-mm-dd');
```

	Id		Operation		Name		Starts		E-Rows		Pstart		Pstop		A-Rows	
	1		SORT AGGREGATE				1		1						1	
	2		PARTITION RANGE MULTI-COLUMN				1		27		KEY(MC		KEY(MC		27	
*	3		TABLE ACCESS FULL		T		8		27		KEY(MC		KEY(MC		27	

Predicate Information (identified by operation id):

```
3 - filter("D1"=TO_DATE(' 2007-07-19 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
```

Until now, all restrictions used for partition pruning were based on literal values or bind variables. However, it is not uncommon for restrictions to actually be join conditions. Whenever a join is based on a partition key, the query optimizer has to choose between three strategies.

SQL> REM Without subquery

The first strategy is to elude partition pruning. The following query (note that the table tx is a copy of the table t; the only difference is that table tx is not partitioned) illustrates this, where no partition pruning on the table t is performed. In fact, since operation 4 is a `PARTITION RANGE ALL`, operation 5 is processed for all partitions. In this particular example, this execution plan is highly inefficient. This is especially the case given that the selectivity of the query is very strong.

```
SQL> SELECT /*+ leading(tx) use_hash(t) */ max(t.n3) FROM tx, t
      WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
* 2	HASH JOIN		1	7			7
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
5	PARTITION RANGE ALL		1	10000	1	48	10000
6	TABLE ACCESS FULL	T	48	10000	1	48	10000

Predicate Information (identified by operation id):

- 2 - access("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1")
- 4 - access("TX"."ID"=19)

The second strategy is to execute the join with the operation `NESTED LOOPS` and define the table, which the partition pruning should occur on, as the second child. The operation `NESTED LOOPS` is a related-combine operation, and therefore, its first child controls the execution of the second child. In following example, the operation `PARTITION RANGE ITERATOR` and the values of the columns `Pstart` and `Pstop` confirm that partition pruning takes place. According to the column `Starts`, a single partition is accessed. This strategy performs well only if the number of rows returned by the first child of the operation `NESTED LOOP` (in this case operation 3) is low. Otherwise, it is even possible that the same partition is accessed several times by the second child (in this case operation 5).

```
SQL> ALTER SESSION SET "_subquery_pruning_enabled" = FALSE;
SQL> SELECT /*+ leading(tx) use_nl(t) */ max(t.n3) FROM tx, t
      WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	NESTED LOOPS		1	7			7
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
5	PARTITION RANGE ITERATOR		1	7	KEY	KEY	7
* 6	TABLE ACCESS FULL	T	1	7	KEY	KEY	7

Predicate Information (identified by operation id):

```
-----  
4 - access("TX"."ID"=19)  
6 - filter(("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1"))
```

The third strategy is to execute the join with the operation HASH JOIN or MERGE JOIN. No regular partition pruning based on the join condition is possible, however, with these join methods But the query optimizer can take advantage of another type of partition pruning, the *subquery pruning*. The idea here is *to find out which partitions of the second child should be accessed, by executing a recursive query* (on the table accessed by the first child) to retrieve the columns used in the join condition that maps to the partition keys of the second child. Then, by consulting the partition definitions of the second child stored in the data dictionary, the partitions to be accessed by the second child are identified, and so it is possible to scan only them. Note that the operation PARTITION RANGE SUBQUERY and the value of the columns Pstart and Pstop confirm that partition pruning takes place. According to the column Starts, a single partition is accessed.

SQL> REM With subquery pruning

```
SQL> ALTER SESSION SET "_subquery_pruning_enabled" = TRUE;  
SQL> ALTER SESSION SET "_subquery_pruning_cost_factor"=1;  
SQL> ALTER SESSION SET "_subquery_pruning_reduction"=100;  
SQL> SELECT /*+ leading(tx) use_hash(t) */ max(t.n3) FROM tx, t  
        WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows	

1	SORT AGGREGATE		1	1			1	
* 2	HASH JOIN		1	7			7	
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1	
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1	
5	PARTITION RANGE SUBQUERY		1	10000	KEY(SQ	KEY(SQ	212	
6	TABLE ACCESS FULL	T	1	10000	KEY(SQ	KEY(SQ	212	

Predicate Information (identified by operation id):

```
-----  
2 - access("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1")  
4 - access("TX"."ID"=19)
```

What actually happens here is that in order to find out which partitions have to be accessed, the operation PARTITION RANGE SUBQUERY executes the following query recursively. This recursive query retrieves the number of the partitions that contain relevant data with the function `tbl$or$idx$part$num`. With this information, operation 6 can take advantage of partition pruning. For example, in this case, only partition 37 needs to be scanned:

```
SQL> SELECT DISTINCT tbl$or$idx$part$num("T", 0, 1, 0, "N1", "D1") AS part_num  
2 FROM (SELECT "TX"."N1" "N1", "TX"."D1" "D1"  
3 FROM "TX" "TX")
```

```

4 WHERE "TX"."ID"=19)
5 ORDER BY 1;
PART_NUM
-----
37

```

Clearly, it makes sense to use this third technique only when the overhead caused by the execution of the recursive query is less than the gain because of partition pruning. For the query used in this example, the efficiency of the execution plans resulting from the second and third strategies are very similar. Nevertheless, if selectivity were weaker, the execution plan resulting from the third strategy would be more efficient.

Note that the operation `PARTITION RANGE SUBQUERY` has been introduced in Oracle10g Release 2. In previous releases, where subquery pruning is also available, the operation `PARTITION RANGE ITERATOR` is used instead.

Note 179518.1 *Partition Pruning and Joins*

Note 209070.1 *Partition Pruning based on Joins to Partitioning Criteria Stored in Dim Tables*

2.7.1 – Partition Range (2)

- PARTITION RANGE SUBQUERY
 - WHERE **tx.d1 = t.d1 AND tx.n1 = t.n1** and tx.id=3; → Pstart Pstop
KEY(SQ) KEY(SQ)
- PARTITION RANGE ITERATOR
 - WHERE **n1 = 3** → 25 37
- PARTITION RANGE Multi Column
 - WHERE **d1 = to_date('2007-07-19', 'yyyy-MM-DD');** → KEY(MC) KEY(MC)

t & tx

Range Partition (n1,d1)

n1 ↑	37	38	39	40	41	42	43	44	45	46	47	48
	25	26	27	28	29	30	31	32	33	34	35	36
	13	14	15	16	17	18	19	20	21	22	23	24
	1	2	3	4	5	6	7	8	9	10	11	12
	d1 →											

ORACLE

PARTITION Range Scans

SQL> REM ITERATOR

SELECT max(n3) FROM t WHERE n1 = 3

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION RANGE ITERATOR		1	941	27	31	932
* 3	TABLE ACCESS FULL	T	5	941	27	31	932

Predicate Information (identified by operation id):

```

3 - filter(("N1"=3
      AND "D1"<=TO_DATE(' 2007-07-19 00:00:00', 'syyy-MM-dd hh24:mi:ss')
      AND "D1">=TO_DATE(' 2007-03-06 00:00:00', 'syyy-MM-dd hh24:mi:ss')))
```

SQL> REM MULTI-COLUMN

```
SELECT max(n3) FROM t WHERE n1 = 3
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION RANGE ITERATOR		1	2500	25	37	2500
* 3	TABLE ACCESS FULL	T	13	2500	25	37	2500

Predicate Information (identified by operation id):

```
3 - filter("N1"=3)
```

```
SQL> SELECT max(n3) FROM t WHERE d1 = to_date('2007-07-19','yyyy-mm-dd');
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION RANGE MULTI-COLUMN		1	27	KEY(MC	KEY(MC	27
* 3	TABLE ACCESS FULL	T	8	27	KEY(MC	KEY(MC	27

Predicate Information (identified by operation id):

```
3 - filter("D1"=TO_DATE(' 2007-07-19 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
```

SQL> REM Without subquery

```
SQL> ALTER SESSION SET "_subquery_pruning_enabled" = FALSE;
```

```
SQL> ALTER SESSION SET "_subquery_pruning_enabled" = FALSE;
```

```
SQL> SELECT /*+ leading(tx) use_nl(t) */ max(t.n3) FROM tx, t
        WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	NESTED LOOPS		1	7			7
3	TABLE ACCESS BY INDEX ROWI	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
5	PARTITION RANGE ITERATOR		1	7	KEY	KEY	7
* 6	TABLE ACCESS FULL	T	1	7	KEY	KEY	7

Predicate Information (identified by operation id):

```
4 - access("TX"."ID"=19)
```

```
6 - filter(("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1"))
```

```
SQL> SELECT /*+ leading(tx) use_hash(t) */ max(t.n3) FROM tx, t
        WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
* 2	HASH JOIN		1	7			7
3	TABLE ACCESS BY INDEX ROWI	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
5	PARTITION RANGE ALL		1	10000	1	48	10000
6	TABLE ACCESS FULL	T	48	10000	1	48	10000

Predicate Information (identified by operation id):

- ```

2 - access("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1")
4 - access("TX"."ID"=19)

```

```
SQL> SELECT /*+ leading(tx) use_merge(t) */ max(t.n3) FROM tx, t
 WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;
```

| Id  | Operation                  | Name  | Starts | E-Rows | Pstart | Pstop | A-Rows |
|-----|----------------------------|-------|--------|--------|--------|-------|--------|
| 1   | SORT AGGREGATE             |       | 1      | 1      |        |       | 1      |
| 2   | MERGE JOIN                 |       | 1      | 7      |        |       | 7      |
| 3   | TABLE ACCESS BY INDEX ROWI | TX    | 1      | 1      |        |       | 1      |
| * 4 | INDEX UNIQUE SCAN          | TX_PK | 1      | 1      |        |       | 1      |
| * 5 | FILTER                     |       | 1      |        |        |       | 7      |
| 6   | PARTITION RANGE ALL        |       | 1      | 10000  | 1      | 48    | 10000  |
| 7   | TABLE ACCESS FULL          | T     | 48     | 10000  | 1      | 48    | 10000  |

Predicate Information (identified by operation id):

- ```

4 - access("TX"."ID"=19)
5 - filter(("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1"))

```

SQL> REM With subquery pruning

```
SQL> ALTER SESSION SET "_subquery_pruning_enabled" = TRUE;
SQL> ALTER SESSION SET "_subquery_pruning_cost_factor"=1;
SQL> ALTER SESSION SET "_subquery_pruning_reduction"=100;
SQL> SELECT /*+ leading(tx) use_hash(t) */ max(t.n3) FROM tx, t
        WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
* 2	HASH JOIN		1	7			7
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
5	PARTITION RANGE SUBQUERY		1	10000	KEY(SQ	KEY(SQ	212
6	TABLE ACCESS FULL	T	1	10000	KEY(SQ	KEY(SQ	212

Predicate Information (identified by operation id):

- 2 - access("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1")
 4 - access("TX"."ID"=19)

```
SQL> SELECT /*+ leading(tx) use_merge(t) */ max(t.n3) FROM tx, t
        WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	MERGE JOIN		1	7			7
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
* 5	FILTER		1				7
6	PARTITION RANGE SUBQUERY		1	10000	KEY(SQ	KEY(SQ	212
7	TABLE ACCESS FULL	T	1	10000	KEY(SQ	KEY(SQ	212

Predicate Information (identified by operation id):

- 4 - access("TX"."ID"=19)
 5 - filter(("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1"))

2.7.2 – Partition Hash

- PARTITION HASH SINGLE
 - WHERE n1 = 3 AND d1 = to_date('2007-07-19',...); → Pstart 23 Pstop 23
 - WHERE n1 = :n AND d1 = to_date(:d1,'yyyy-MM-DD'); → Key Key
- PARTITION HASH INLIST
 - WHERE n1 in (1,3) AND d1 = to_date('2007-07-19',...); → Key(I) Key(I)
 - WHERE n1 = 1 AND d1 in (to_date('2007-07-19',...);
- PARTITION HASH ALL → 1 32
 - WHERE n1 = 3 ;
- PARTITION HASH SUBQUERY → KEY(SQ) KEY(SQ)
 - WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 and tx.id=3;

t & tx

PARTITION BY HASH (n1, d1)

PARTITIONS 32

25	26	27	28	29	30	31	32
17	18	19	20	21	22	23	24
9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8

ORACLE

PARTITION Hash Scans

```
SQL> DROP TABLE t;
SQL> CREATE TABLE t (
  2   id NUMBER,
  3   d1 DATE,
  4   --d1 DATE NOT NULL,
  5   n1 NUMBER,
  6   --n1 NUMBER NOT NULL,
  7   n2 NUMBER,
  8   n3 NUMBER,
  9   pad VARCHAR2(4000),
 10  CONSTRAINT t_pk PRIMARY KEY (id)
 11 )
 12 PARTITION BY HASH (n1, d1) PARTITIONS 32;
```

```
SQL> execute dbms_random.seed(0)
```

```

SQL> INSERT INTO t
  2  SELECT rownum AS id,
  3         trunc(to_date('2007-01-01','yyyy-mm-dd')+rownum/27.4) AS d1,
  4         1+mod(rownum,4) AS n1,
  5         255+mod(trunc(dbms_random.normal*1000),255) AS n2,
  6         round(4515+dbms_random.normal*1234) AS n3,
  7         dbms_random.string('p',255) AS pad
  8  FROM dual
  9  CONNECT BY level <= 10000
 10  ORDER BY dbms_random.value;

```

```

SQL> BEGIN
  2  dbms_stats.gather_table_stats(
  3      ownname      => user,
  4      tabname      => 'T',
  5      estimate_percent => 100,
  6      method_opt    => 'for all columns size skewonly',
  7      cascade       => TRUE
  8  );
  9  END;
 10  /

```

```

SQL> SELECT partition_name, partition_position, num_rows
  2  FROM user_tab_partitions
  3  WHERE table_name = 'T'
  4  ORDER BY partition_position;

```

PARTITION_NAME	PARTITION_POSITION	NUM_ROWS
SYS_P85	1	342
SYS_P86	2	300
SYS_P87	3	286
SYS_P88	4	267
...		
SYS_P113	29	255
SYS_P114	30	319
SYS_P115	31	265
SYS_P116	32	295

```

SQL> CREATE TABLE tx AS SELECT * FROM t;
SQL> ALTER TABLE tx ADD CONSTRAINT tx_pk PRIMARY KEY (id);

```

```

SQL> BEGIN
  2  dbms_stats.gather_table_stats(
  3      ownname      => user,

```

```

4      tabname      => 'TX'
5  );
6  END;
7  /

```

SQL> REM SINGLE

```

SQL> SELECT max(n3) FROM t
      WHERE n1 = 3 AND d1 = to_date('2007-07-19','yyyy-mm-dd');

```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION HASH SINGLE		1	2	23	23	7
* 3	TABLE ACCESS FULL	T	1	2	23	23	7

Predicate Information (identified by operation id):

```

3 - filter(("D1"=TO_DATE(' 2007-07-19 00:00:00', 'syyy-MM-dd hh24:mi:ss')
      AND "N1"=3))

```

```

SQL> VARIABLE n1 NUMBER

```

```

SQL> EXECUTE :n1 := 3

```

```

SQL> VARIABLE d1 VARCHAR2(10)

```

```

SQL> EXECUTE :d1 := '2007-07-19'

```

```

SQL> SELECT max(n3) FROM t
      WHERE n1 = :n1 AND d1 = to_date(:d1,'yyyy-mm-dd');

```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION HASH SINGLE		1	2	KEY	KEY	7
* 3	TABLE ACCESS FULL	T	1	2	KEY	KEY	7

Predicate Information (identified by operation id):

```

3 - filter(("N1"=:N1
      AND "D1"=TO_DATE(:D1,'yyyy-mm-dd')))

```

```

SQL> SELECT max(n3) FROM t
      WHERE n1 IN (3) AND d1 IN (to_date('2007-07-19','yyyy-mm-dd'));

```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION HASH SINGLE		1	2	23	23	7
* 3	TABLE ACCESS FULL	T	1	2	23	23	7

Predicate Information (identified by operation id):

```

3 - filter(("D1"=TO_DATE(' 2007-07-19 00:00:00', 'yyyy-mm-dd hh24:mi:ss')
AND "N1"=3))

```

SQL> REM INLIST

```

SQL> SELECT max(n3) FROM t
WHERE n1 IN (1,3) AND d1 = to_date('2008-07-19','yyyy-mm-dd');

```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION HASH INLIST		1	6	KEY(I)	KEY(I)	0
* 3	TABLE ACCESS FULL	T	2	6	KEY(I)	KEY(I)	0

Predicate Information (identified by operation id):

```

3 - filter(("D1"=TO_DATE(' 2008-07-19 00:00:00', 'yyyy-mm-dd hh24:mi:ss')
AND INTERNAL_FUNCTION("N1")))

```

SQL> REM ALL

```

SQL> SELECT max(n3) FROM t WHERE n1 = 3;

```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION HASH ALL		1	2500	1	32	2500
* 3	TABLE ACCESS FULL	T	32	2500	1	32	2500

Predicate Information (identified by operation id):

```

3 - filter("N1"=3)

```

```

SQL> SELECT max(n3) FROM t
WHERE n1 IS NULL;

```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION HASH ALL		1	1	1	32	0
* 3	TABLE ACCESS FULL	T	32	1	1	32	0

Predicate Information (identified by operation id):

3 - filter("N1" IS NULL)

```
SQL> SELECT max(n3) FROM t
      WHERE n1 = 3 AND d1 BETWEEN to_date('2007-03-06','yyyy-mm-dd')
      AND to_date('2007-07-19','yyyy-mm-dd');
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION HASH ALL		1	941	1	32	932
* 3	TABLE ACCESS FULL	T	32	941	1	32	932

Predicate Information (identified by operation id):

3 - filter(("N1"=3
AND "D1"<=TO_DATE(' 2007-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss')
AND "D1">=TO_DATE(' 2007-03-06 00:00:00', 'syyyy-mm-dd hh24:mi:ss')))

```
SQL> SELECT max(n3) FROM t
      WHERE n1 = 3 AND d1 < to_date('2007-07-19','yyyy-mm-dd');
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION HASH ALL		1	1367	1	32	1363
* 3	TABLE ACCESS FULL	T	32	1367	1	32	1363

Predicate Information (identified by operation id):

3 - filter(("N1"=3
AND "D1"<TO_DATE(' 2007-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss')))

```
SQL> SELECT max(n3) FROM t
      WHERE n1 != 3 AND d1 != to_date('2007-07-19','yyyy-mm-dd');
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION HASH ALL		1	7479	1	32	7480
* 3	TABLE ACCESS FULL	T	32	7479	1	32	7480

Predicate Information (identified by operation id):

```

3 - filter(("N1"<>3
      AND "D1"<>TO_DATE(' 2007-07-19 00:00:00', 'syyy-MM-dd hh24:mi:ss')))
```

SQL> REM OR condition

```

SQL> SELECT max(n3) FROM t
      WHERE n1 = 3 OR d1 = to_date('2007-03-06','yyyy-mm-dd');
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION HASH ALL		1	2521	1	32	2520
* 3	TABLE ACCESS FULL	T	32	2521	1	32	2520

Predicate Information (identified by operation id):

```

3 - filter(("N1"=3
      OR "D1"=TO_DATE(' 2007-03-06 00:00:00', 'syyy-MM-dd hh24:mi:ss')))
```

SQL> REM Without subquery

```

SQL> ALTER SESSION SET "_subquery_pruning_enabled" = FALSE;
```

```

SQL> SELECT /*+ leading(tx) use_nl(t) */ max(t.n3) FROM tx, t
      WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	NESTED LOOPS		1	7			7
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
5	PARTITION HASH ITERATOR		1	7	KEY	KEY	7
* 6	TABLE ACCESS FULL	T	1	7	KEY	KEY	7

Predicate Information (identified by operation id):

```

4 - access("TX"."ID"=19)
6 - filter(("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1"))

```

```

SQL> SELECT /*+ leading(tx) use_hash(t) */ max(t.n3) FROM tx, t
      WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;

```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
* 2	HASH JOIN		1	7			7
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
5	PARTITION HASH ALL		1	10000	1	32	10000
6	TABLE ACCESS FULL	T	32	10000	1	32	10000

Predicate Information (identified by operation id):

```

2 - access("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1")
4 - access("TX"."ID"=19)

```

```

SQL> SELECT /*+ leading(tx) use_merge(t) */ max(t.n3) FROM tx, t
      WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;

```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	MERGE JOIN		1	7			7
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
* 5	FILTER		1				7
6	PARTITION HASH ALL		1	10000	1	32	10000
7	TABLE ACCESS FULL	T	32	10000	1	32	10000

Predicate Information (identified by operation id):

```

4 - access("TX"."ID"=19)
5 - filter(("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1"))

```

```

SQL> REM With subquery pruning

```

```

SQL> ALTER SESSION SET "_subquery_pruning_enabled" = TRUE;

```

```

SQL> ALTER SESSION SET "_subquery_pruning_cost_factor"=1;

```

```
SQL> ALTER SESSION SET "_subquery_pruning_reduction"=100;
```

```
SQL> SELECT /*+ leading(tx) use_hash(t) */ max(t.n3) FROM tx, t
        WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
* 2	HASH JOIN		1	7			7
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
5	PARTITION HASH SUBQUERY		1	10000	KEY(SQ	KEY(SQ	301
6	TABLE ACCESS FULL	T	1	10000	KEY(SQ	KEY(SQ	301

Predicate Information (identified by operation id):

- ```

2 - access("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1")
4 - access("TX"."ID"=19)

```

```
SQL> SELECT /*+ leading(tx) use_merge(t) */ max(t.n3) FROM tx, t
 WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;
```

| Id  | Operation                   | Name  | Starts | E-Rows | Pstart | Pstop  | A-Rows |
|-----|-----------------------------|-------|--------|--------|--------|--------|--------|
| 1   | SORT AGGREGATE              |       | 1      | 1      |        |        | 1      |
| 2   | MERGE JOIN                  |       | 1      | 7      |        |        | 7      |
| 3   | TABLE ACCESS BY INDEX ROWID | TX    | 1      | 1      |        |        | 1      |
| * 4 | INDEX UNIQUE SCAN           | TX_PK | 1      | 1      |        |        | 1      |
| * 5 | FILTER                      |       | 1      |        |        |        | 7      |
| 6   | PARTITION HASH SUBQUERY     |       | 1      | 10000  | KEY(SQ | KEY(SQ | 301    |
| 7   | TABLE ACCESS FULL           | T     | 1      | 10000  | KEY(SQ | KEY(SQ | 301    |

Predicate Information (identified by operation id):

- ```

4 - access("TX"."ID"=19)
5 - filter(("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1"))

```



2.7.3 – Partition List

		Pstart	Pstop
• PARTITION LIST SINGLE			
• WHERE n1 = 3 AND d1 = to_date('2007-07-19',...);	—————→	31	31
• WHERE n1 = :n AND d1 = to_date(:d1,'yyyy-MM-DD');	—————→	Key	Key
• PARTITION LIST INLIST			
• WHERE n1 = (1,3) AND d1 = to_date('2007-07-19',...);	—————→	Key(I)	Key(I)
• PARTITION LIST ALL			
• WHERE n3 > 3000;	—————→	1	48
• PARTITION LIST SUBQUERY			
• WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 and tx.id=3;	—————→	KEY(SQ)	KEY(SQ)
• PARTITION LIST EMPTY			
• WHERE n1 = 5 AND d1 < to_date('2007-07-19',...);	—————→	INVALID	INVALID

t & tx List Partition (n1)

1	2	3	4	5
---	---	---	---	---

ORACLE

PARTITION List Scans

```
SQL> CREATE TABLE t (  
2   id NUMBER,  
3   d1 DATE,  
4   n1 NUMBER,  
5   n2 NUMBER,  
6   n3 NUMBER,  
7   pad VARCHAR2(4000),  
8   CONSTRAINT t_pk PRIMARY KEY (id)  
9 )  
10 PARTITION BY LIST (n1) (  
11   PARTITION t_1 VALUES (1),  
12   PARTITION t_2 VALUES (2),  
13   PARTITION t_3 VALUES (3),  
14   PARTITION t_4 VALUES (4),  
15   PARTITION t_null VALUES (NULL)  
16 );
```

```
SQL> execute dbms_random.seed(0)
```

```
SQL> INSERT INTO t
```

```
2  SELECT rownum AS id,  
3         trunc(to_date('2007-01-01','yyyy-mm-dd')+rownum/27.4) AS d1,  
4         1+mod(rownum,4) AS n1,  
5         255+mod(trunc(dbms_random.normal*1000),255) AS n2,  
6         round(4515+dbms_random.normal*1234) AS n3,  
7         dbms_random.string('p',255) AS pad  
8  FROM dual  
9  CONNECT BY level <= 10000  
10 ORDER BY dbms_random.value;
```

```
SQL> BEGIN
```

```
2  dbms_stats.gather_table_stats(  
3      ownname      => user,  
4      tabname      => 'T',  
5      estimate_percent => 100,  
6      method_opt   => 'for all columns size skewonly',  
7      cascade      => TRUE  
8  );  
9  END;  
10 /
```

```
SQL> SELECT partition_name, partition_position, num_rows
```

```
2  FROM user_tab_partitions  
3  WHERE table_name = 'T'  
4  ORDER BY partition_position;
```

PARTITION_NAME	PARTITION_POSITION	NUM_ROWS
T_1	1	2500
T_2	2	2500
T_3	3	2500
T_4	4	2500
T_NULL	5	0

```
SQL> CREATE TABLE tx AS SELECT * FROM t;
```

```
SQL> ALTER TABLE tx ADD CONSTRAINT tx_pk PRIMARY KEY (id);
```

```
SQL> BEGIN
```

```
2  dbms_stats.gather_table_stats(  
3      ownname => user, tabname => 'TX');  
4  END;  
5  /
```

SQL> REM LIST SINGLE

```
SQL> SELECT max(n3) FROM t
      WHERE n1 = 3 AND d1 = to_date('2007-07-19','yyyy-mm-dd');
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION LIST SINGLE		1	7	3	3	7
* 3	TABLE ACCESS FULL	T	1	7	3	3	7

3 - filter("D1"=TO_DATE(' 2007-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))

```
SQL> VARIABLE n1 NUMBER
```

```
SQL> EXECUTE :n1 := 3
```

```
SQL> VARIABLE d1 VARCHAR2(10)
```

```
SQL> EXECUTE :d1 := '2007-07-19'
```

```
SQL> SELECT max(n3) FROM t
      WHERE n1 = :n1 AND d1 = to_date(:d1,'yyyy-mm-dd');
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION LIST SINGLE		1	7	KEY	KEY	7
* 3	TABLE ACCESS FULL	T	1	7	KEY	KEY	7

3 - filter("D1"=TO_DATE(:D1,'yyyy-mm-dd'))

SQL> REM INLIST

```
SQL> SELECT max(n3) FROM t
      WHERE n1 IN (1,3) AND d1 = to_date('2007-07-19','yyyy-mm-dd');
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION LIST INLIST		1	14	KEY(I	KEY(I	13
* 3	TABLE ACCESS FULL	T	2	14	KEY(I	KEY(I	13

3 - filter("D1"=TO_DATE(' 2007-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))

SQL> REM EMPTY

```
SQL> SELECT max(n3) FROM t
      WHERE n1 = 5 AND d1 = to_date('2007-07-19','yyyy-mm-dd');
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION LIST EMPTY		1	2	INVAL	INVAL	0
* 3	TABLE ACCESS FULL	T	0	2	INVAL	INVAL	0

```
3 - filter(("D1"=TO_DATE(' 2007-07-19 00:00:00', 'syyy-MM-DD HH24:MI:SS') AND
"N1"=5))
```

SQL> REM ALL

```
SQL> SELECT max(n3) FROM t
      WHERE n3 BETWEEN 6000 AND 7000;
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	PARTITION LIST ALL		1	947	1	5	935
* 3	TABLE ACCESS FULL	T	5	947	1	5	935

```
3 - filter(("N3">=6000 AND "N3"<=7000))
```

```
SQL> REM Subquery
```

SQL> REM Without subquery

```
SQL> ALTER SESSION SET "_subquery_pruning_enabled" = FALSE;
```

```
SQL> SELECT /*+ leading(tx) use_nl(t) */ max(t.n3) FROM tx, t
      WHERE tx.d1 = t.d1 AND tx.n1 = t.n1
      AND tx.id = 19;
```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	NESTED LOOPS		1	7			7
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
5	PARTITION LIST ITERATOR		1	7	KEY	KEY	7
* 6	TABLE ACCESS FULL	T	1	7	KEY	KEY	7

```
4 - access("TX"."ID"=19)
```

```
6 - filter(("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1"))
```

```
SQL> SELECT /*+ leading(tx) use_hash(t) */ max(t.n3) FROM tx, t
```

WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
* 2	HASH JOIN		1	7			7
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
5	PARTITION LIST ALL		1	10000	1	5	10000
6	TABLE ACCESS FULL	T	5	10000	1	5	10000

2 - access("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1")

4 - access("TX"."ID"=19)

SQL> SELECT /*+ leading(tx) use_merge(t) */ max(t.n3) FROM tx, t
WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	MERGE JOIN		1	7			7
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
* 5	FILTER		1				7
6	PARTITION LIST ALL		1	10000	1	5	10000
7	TABLE ACCESS FULL	T	5	10000	1	5	10000

4 - access("TX"."ID"=19)

5 - filter(("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1"))

SQL> REM With subquery pruning

SQL> ALTER SESSION SET "_subquery_pruning_enabled" = TRUE;

SQL> ALTER SESSION SET "_subquery_pruning_cost_factor"=1;

SQL> ALTER SESSION SET "_subquery_pruning_reduction"=100;

SQL> SELECT /*+ leading(tx) use_hash(t) */ max(t.n3) FROM tx, t
WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
* 2	HASH JOIN		1	7			7
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
5	PARTITION LIST SUBQUERY		1	10000	KEY(SQ	KEY(SQ	2500
6	TABLE ACCESS FULL	T	1	10000	KEY(SQ	KEY(SQ	2500

```

2 - access("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1")
4 - access("TX"."ID"=19)

```

```

SQL> SELECT /*+ leading(tx) use_merge(t) */ max(t.n3) FROM tx, t
      WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19;

```

Id	Operation	Name	Starts	E-Rows	Pstart	Pstop	A-Rows
1	SORT AGGREGATE		1	1			1
2	MERGE JOIN		1	7			7
3	TABLE ACCESS BY INDEX ROWID	TX	1	1			1
* 4	INDEX UNIQUE SCAN	TX_PK	1	1			1
* 5	FILTER		1				7
6	PARTITION LIST SUBQUERY		1	10000	KEY(SQ	KEY(SQ	2500
7	TABLE ACCESS FULL	T	1	10000	KEY(SQ	KEY(SQ	2500

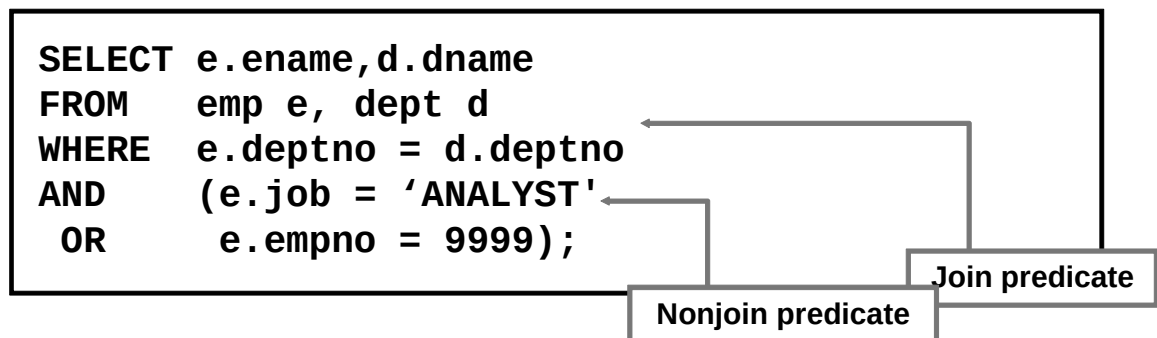
```

4 - access("TX"."ID"=19)
5 - filter(("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1"))

```

3 Joins

- A join defines the relationship between 2 row sources.
- A join is a method of combining data from 2 data sources
- It is controlled by join predicates which define how the objects are related



ORACLE

Unary Operations

- 1 - Access Path
- 2 - Other Unary Operations

Binary Operations

3 - Joins

N-ary Operations

- 4 - Other N-ary Operations

Unary Operations : operations that act on only 1 input

Binary Operations : operations that act on 2 inputs like Joins

N-ary Operations : operations that act several inputs like Relational Operator

A row source is a set of data that can be accessed in a query. It can be a table, an index, a NON-mergeable View, or even the result set of a join tree consisting of many different objects.

Only two row sources can be joined in a single operation.

A join predicate is a predicate in the WHERE clause that combines the columns of two of the tables in the join.

A nonjoin predicate is a predicate in the WHERE clause that references only one table.

A join operation combines the output from two row sources (such as tables or views) and returns one resulting row source (data set). Oracle supports different join methods such as the following:

- Nested loop join
- Hash join (only available with CBO)
- Sort-merge join

The corresponding ANSI join is :

```
SELECT e.ename,d.dname
FROM   emp e JOIN dept d
ON     (e.deptno = d.deptno)
AND    (e.job = 'ANALYST' OR e.empno=9999);
```

The optimizer can use join techniques when executing other kinds of statements where more than one table is referenced.



3.1 - Join Methods

- Nested Loops
- Merge Join
- Hash Join

ORACLE

JOIN METHODS

Nested loop join: Useful when small subsets of data are being joined and if the join condition is an efficient way of accessing the second table.

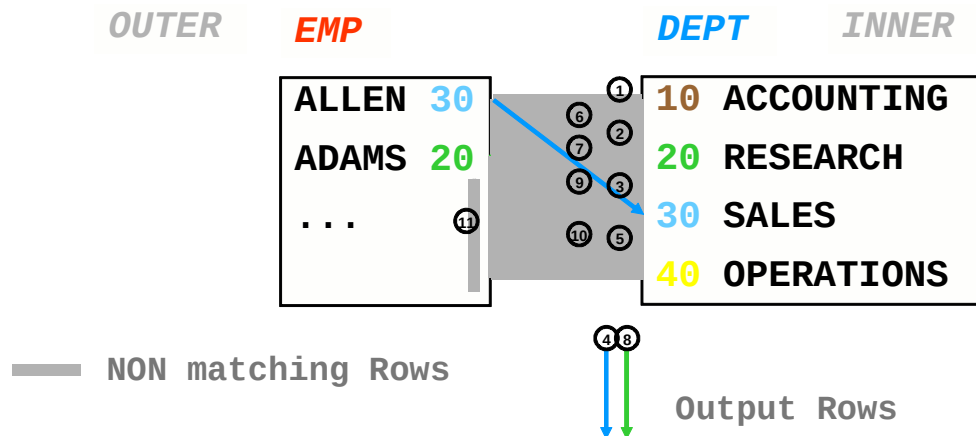
Hash join: Used for joining large data sets. The optimizer uses the smaller of two tables or data sources to build a hash table on the join key in memory. It then scans the larger table, probing the hash table to find the joined rows. This method is best used when the smaller table fits in available memory. The cost is then limited to a single read pass over the data for the two tables.

Sort-merge join: Can be used to join rows from two independent sources. Hash joins generally perform better than sort-merge joins. On the other hand, sort-merge joins can perform better than hash joins if 1 or 2 row sources are already sorted.

Following slides are detailing these join Methods.

3.1.1 - Nested Loop (1)

```
SQL> select ename, e.deptno, d.deptno, dname
2   from emp e, dept d
3   where e.deptno = d.deptno
4     and ename like 'A%'
```



ORACLE

NESTED LOOP

One of the two tables is defined as the *outer* table (or the *driving* table or the *left hand* side). The other table is called the *inner* table (or the *right hand* side).

For each row in the outer (driving) table (that match the single table predicates), all rows in the inner table that satisfy the join predicate (matching rows) are retrieved.

- ① 1st Row from Outer Table (ALLEN - 30) – 1st Row from INNER table (10)
- ② 1st Row from Outer Table (ALLEN - 30) – 2nd Row from INNER table (20)
- ③ 1st Row from Outer Table (ALLEN - 30) – 3rd Row from INNER table (30)
- ④ Previous row matched and so row is returned
- ⑤ 1st Row from Outer Table (ALLEN - 30) – 4th Row from INNER table (40)
- ⑥ 2nd Row from Outer Table (ADAMS - 20) – 1st Row from INNER table (10)
- ⑦ 2nd Row from Outer Table (ADAMS - 20) – 2nd Row from INNER table (20)
- ⑧ Previous row matched and so row is returned
- ⑨ 2nd Row from Outer Table (ADAMS - 20) – 3rd Row from INNER table (30)
- ⑩ 2nd Row from Outer Table (ADAMS - 20) – 4th Row from INNER table (40)
- ⑪ no more rows in outer table match like 'A%'

Any nonjoin predicates on the inner table are considered after this initial retrieval, unless a composite index (combining both the join and the nonjoin predicate) is used.

Code to emulate a nested loop join might look something like :

```
for r1 in (select rows from EMP that match single table predicate ) loop
    for r2 in (select rows from DEPT that match current row from EMP) loop
        output values from current row of EMP and current row of DEPT
    end loop
end loop
```

It is possible to process equijoin and below Non-equi join with NESTED LOOPS.

Execution plan displayed as follow :

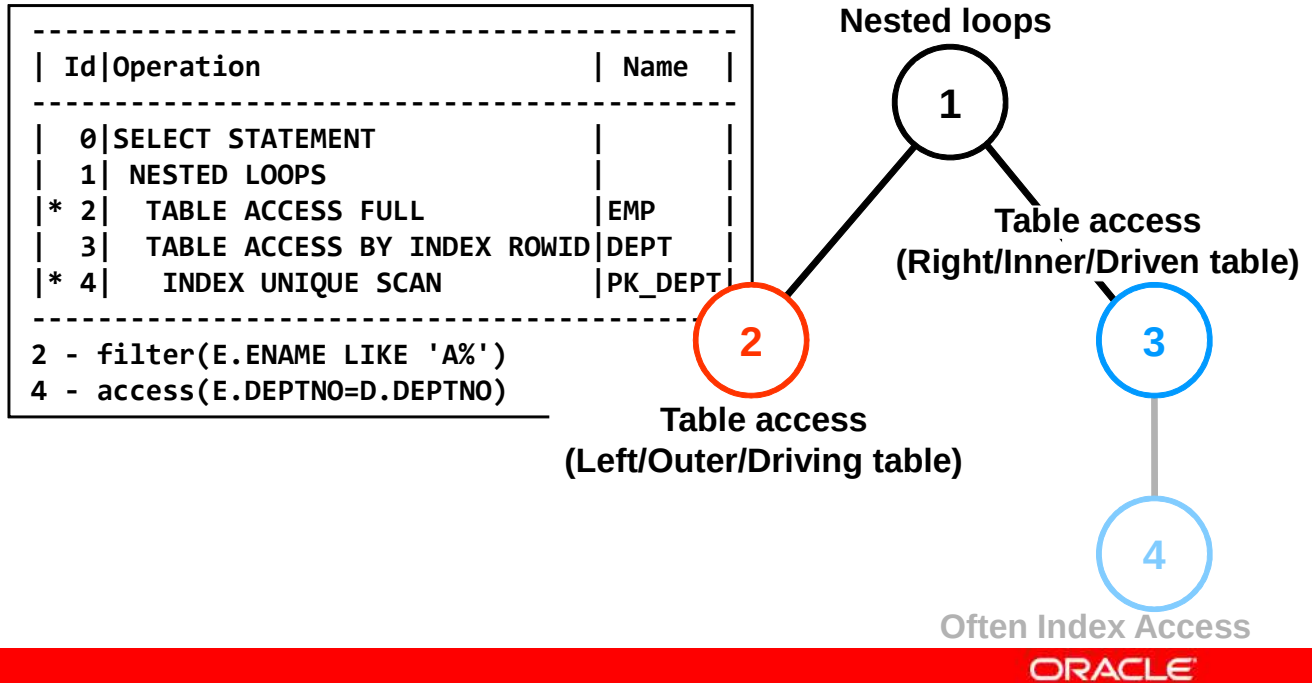
```
select ename, e.deptno, d.deptno, dname
from emp e, dept d
where e.deptno = d.deptno
and ename like 'A%';
```

ENAME	DEPTNO	DEPTNO	DNAME
-----	-----	-----	-----
ALLEN	30	30	SALES
ADAMS	20	20	RESEARCH
-----	-----	-----	-----
Id	Operation	Name	Rows Cost
-----	-----	-----	-----
1	NESTED LOOPS		2 4
* 2	TABLE ACCESS FULL	EMP	2 2
3	TABLE ACCESS BY INDEX ROWID	DEPT	1 1
* 4	INDEX UNIQUE SCAN	PK_DEPT	1
-----	-----	-----	-----
2 - filter("E"."ENAME" LIKE 'A%')			
4 - access("E"."DEPTNO"="D"."DEPTNO")			

Read Plan

- Oracle fetches rows from table EMP
- Then, Oracle applies filter EMP.ENAME = like 'A%' from these fetched
- Using EMP.DEPTNO value from EMP and index PK_DEPT,
- Oracle accesses rows for which EMP.DEPTNO=DEPTNO=DEPTNO,
- Oracle gets their rowid, fetch dname column from table DEPT
- finally, Oracle outputs final resultset.

Nested Loop Join Plan (2)



NESTED LOOP

One of the two tables is defined as the *outer* table (or the *driving* table or the left hand side).

The other table is called the *inner* table (or the right hand side).

For each row in the outer (driving) table (that match the single table predicates), all rows in the inner table that satisfy the join predicate (matching rows) are retrieved.

Actual Execution plan – Explanation

There will be further explanations in Part2 on these displayed columns.

When dbms_xplan package is used with "IOSTATS LAST" parameters to display execution plan

- E-Rows Estimated Rows (from CBO computations)
- A-Rows Actual Rows (from execution row source statistics or execution trace)

- Starts Number of time operation is started

Id	Operation	Name	Starts	E-Rows	A-Rows
1	NESTED LOOPS		1	2	2
* 2	TABLE ACCESS FULL	EMP	1	2	2
3	TABLE ACCESS BY INDEX ROWID	DEPT	2	1	2
* 4	INDEX UNIQUE SCAN	PK_DEPT	2	1	2

2 - filter("ENAME" LIKE 'A%')

4 - access("E"."DEPTNO"="D"."DEPTNO")

Id 2 -> Starts = 1 means that "TABLE ACCESS FULL EMP" is executed once

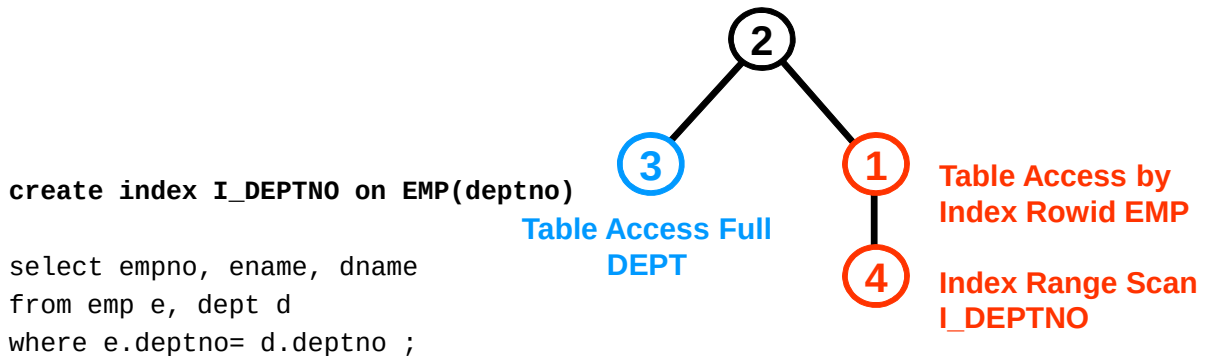
A-Rows = 2 means 2 Actual rows were returned by this step

Id 3 & 4 -> Starts = 2 this means this is executed twice (once for each row returned by outer table in the Id=2 step)

Id 4 -> E-Rows = 1 Estimated Row (1 row to be returned by Index Unique Scan)

A-Rows = 2 Actual Rows (The Index Unique Scan is executed twice and returns 2*1 row)

Nested Loop – Prefetching (3)



Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		14	6
1	TABLE ACCESS BY INDEX ROWID	EMP	4	1
2	NESTED LOOPS		14	6
3	TABLE ACCESS FULL	DEPT	4	2
* 4	INDEX RANGE SCAN	I_DEPTNO	5	

4 - access(EMP.DEPTNO=DEPT.DEPTNO)

ORACLE

NESTED LOOP – PREFETCHING

Oracle introduces an internal optimization called **data block prefetching**.

Objectives : Improve I/O utilization, therefore response time, of index access with table lookup by batching rowid lookups into parallel block reads.

Functionality: Buffer cache adjusts prefetch size according to system load. Execution engine activates prefetching according to index clustering and buffer cache miss rate. Rowids across join keys can be batched so that prefetching won't be limited by the join key selectivity.

This change to the plan output is not considered a different execution plan. It does not affect the join order, join method, access method, or parallelization scheme.

- Available if inner access path is Index Range Scan,
- NOT Available if inner access path is Index Unique Scan

Plan used to be :

Id	Operation	Name
1	NESTED LOOPS	
2	TABLE ACCESS FULL	DEPT
3	TABLE ACCESS BY INDEX ROWID	EMP
* 4	INDEX RANGE SCAN	I_DEPTNO

Data block prefetching is used by table lookup. When an index access path is chosen and the query cannot be satisfied by the index alone, the data rows indicated by the rowid also need to be fetched. This rowid to data row access (table lookup) is improved using data block prefetching which involves reading an array of blocks which are pointed at by an array of qualifying rowids.

Without data block prefetching, accessing a large number of rows using a poorly clustered b-tree index could be expensive. Each row accessed by the index would likely be in a separate data block and thus would require a separate I/O operation

With data block prefetching, Oracle delays data blocks reads until multiple rows specified by the underlying index are ready to be accessed and then retrieves multiple data blocks at once, rather than reading a single data block at a time.

<http://www.oracle.com/technology/products/oracle9i/daily/Aug20.html>

Note 264532.1 : Multi Join Key Pre-fetching

Note 406966.1 : Table Prefetching causes intermitent Wrong Results in 9iR2, 10gR1 and 10gR2

```
create table test(id1 number not null,id2 number not null,id3 number not null);
begin
for i in 1..100000
loop
insert into test(id1, id2, id3) values (i, i, i);
end loop;
commit;
end;
/
create unique index test_idx on park(id1, id3);
exec dbms_stats.gather_table_stats(user,'TEST',cascade=>true,estimate_percent=>100,method_opt=>'FOR ALL
COLUMNS SIZE 254',no_invalidate=> false);
```

10g sans Prefetching

```
SQL> select /*+ use_nl(A B) */max(a.id2) from test a, test b where a.id1=b.id2;
100000
```

Id	Operation	Name	Starts	E-Rows	Cost	A-Rows	Buffers	
1	SORT AGGREGATE		1	1		1	200K	
2	NESTED LOOPS		1	100K	200K	100K	200587	
3	TABLE ACCESS FULL	TEST	1	100K	48	100K	310	
4	TABLE ACCESS BY INDEX ROWID	TEST		100K	1	2	100K	200277
* 5	INDEX RANGE SCAN	TEST_IDX		100K	1	1	100K	100277

Predicate Information (identified by operation id):

5 - access("A"."ID1"="B"."ID2")

10g avec Prefetching

```
SQL> SQL> select /*+ use_nl(A B) */max(a.id2) from test a, test b where a.id1=b.id2;
100000
```

Id	Operation	Name	Starts	E-Rows	Cost	A-Rows	Buffers	
1	SORT AGGREGATE		1	1		1	100859	
2	TABLE ACCESS BY INDEX ROWID	TEST		1	2	100K	100859	
3	NESTED LOOPS		1	100K	200K	200K	100587	
4	TABLE ACCESS FULL	TEST		1	100K	48	100K	310
* 5	INDEX RANGE SCAN	TEST_IDX		100K	1	1	100K	100277

Predicate Information (identified by operation id):

5 - access("A"."ID1"="B"."ID2")

11g - Nested Loops Batching

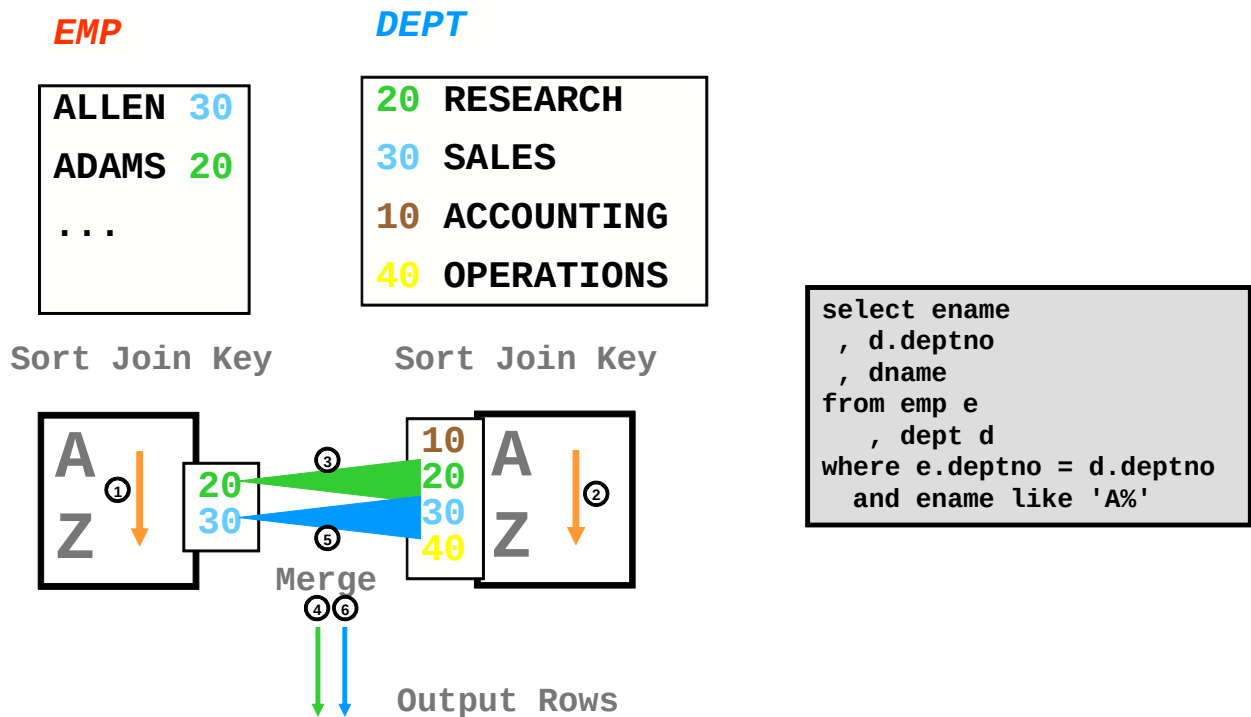
```
SQL> select /*+ use_nl(A B) */max(a.id2) from test a, test b where a.id1=b.id2;
100000
```

Id	Operation	Name	Starts	E-Rows	Cost (%CPU)	A-Rows	Buffers	
0	SELECT STATEMENT		1		200K(100)	1	3695	
1	SORT AGGREGATE		1	1		1	3695	
2	NESTED LOOPS		1			100K	3695	
3	NESTED LOOPS		1	100K	200K (1)	100K	3423	
4	TABLE ACCESS FULL	TEST		1	100K 103 (1)	100K	316	
* 5	INDEX RANGE SCAN	TEST_IDX		100K	1 (0)	100K	3107	
6	TABLE ACCESS BY INDEX ROWID	TEST		100K	1 2 (0)	100K	272	

Predicate Information (identified by operation id):

5 - access("A"."ID1"="B"."ID2")

3.1.2 - Sort-Merge (1)



ORACLE

SORT MERGE (1)

A sort-merge join is executed as follows:

- ① Get the first data set, using any access and filter predicates and sort it on the join columns
- ② Get the second data set, using any access and filter predicates and sort it on the join columns

For each row in the first data set, find the start point in the second data set and scan until you hit a row that does not join. The merge operation combines the two sorted row sources to retrieve every pair of rows that contain matching values for the columns used in the join predicate.

- ③ 1st Row from Left Table (ADAMS - 20) – start point in right Table INNER table (20) until (30)
- ④ 1 row is returned
- ⑤ 2nd Row from Left Table (ALLEN - 30) – start point in right Table INNER table (30) until (40)
- ⑥ 1 row is returned

```

explain plan for
select /*+ leading(e) use_merge(d) */ ename,e.deptno, d.deptno,
      from emp e, dept d
      where e.deptno = d.deptno
            and ename like 'A%';

```

ENAME	DEPTNO	DEPTNO	DNAME
ADAMS	20	20	RESEARCH
ALLEN	30	30	SALES

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		3	60	7
1	MERGE JOIN		3	60	7
2	SORT JOIN		3	27	4
* 3	TABLE ACCESS FULL	EMP	3	27	2
* 4	SORT JOIN		4	44	4
5	TABLE ACCESS FULL	DEPT	4	44	2

```

3 - filter("E"."ENAME" LIKE 'A%')
4 - access("E"."DEPTNO"="D"."DEPTNO")
    filter("E"."DEPTNO"="D"."DEPTNO")

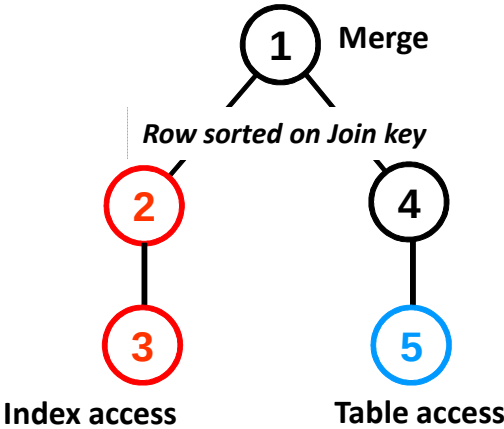
```

You can process equijoin and below Non-equi join with Sort-Merge.

- $t1.col > t2.col$ ou
- $t1.col \geq t2.col$ ou
- $t1.col < t2.col$ ou
- $t1.col \leq t2.col$
- $NOT\ t1.col <> t2.col$



Sort-Merge Join Plan (2)



Id	Operation	Name
1	MERGE JOIN	
* 2	TABLE ACCESS BY INDEX ROWID	EMP
3	INDEX FULL SCAN	I_DEPTNO
* 4	SORT JOIN	
5	TABLE ACCESS FULL	DEPT

2 - filter(ENAME LIKE 'A%')

4 - access(E.DEPTNO=D.DEPTNO)
filter(E.DEPTNO=D.DEPTNO)



Example : plan with MERGE join using INDEX to SORT first data set

```
create index I_DEPTNO on EMP(deptno)

select /*+ leading(e) use_merge(d) index(e) */
       ename, e.deptno, d.deptno, dname
  from emp e, dept d
 where e.deptno = d.deptno and ename like 'A%';
```

ENAME	DEPTNO	DEPTNO	DNAME
ADAMS	20	20	RESEARCH
ALLEN	30	30	SALES

Id	Operation	Name	Starts	E-Rows	A-Rows
----	-----------	------	--------	--------	--------

	1	MERGE JOIN				1	2	2
	*	2	TABLE ACCESS BY INDEX ROWID		EMP		1	2
		3	INDEX FULL SCAN		I_DEPTNO		1	14
	*	4	SORT JOIN				2	4
		5	TABLE ACCESS FULL		DEPT		1	4

```

2 - filter(ENAME LIKE 'A%')
4 - access(E.DEPTNO=D.DEPTNO)
    filter(E.DEPTNO=D.DEPTNO)

```

If First row source has already been sorted in a previous operation (there is an index on join column), the sort-merge operation skips the sort on that row source.

When you do a MERGE JOIN, you need to fetch ALL rows from the 2 row sources before to return the first row to the next operation.

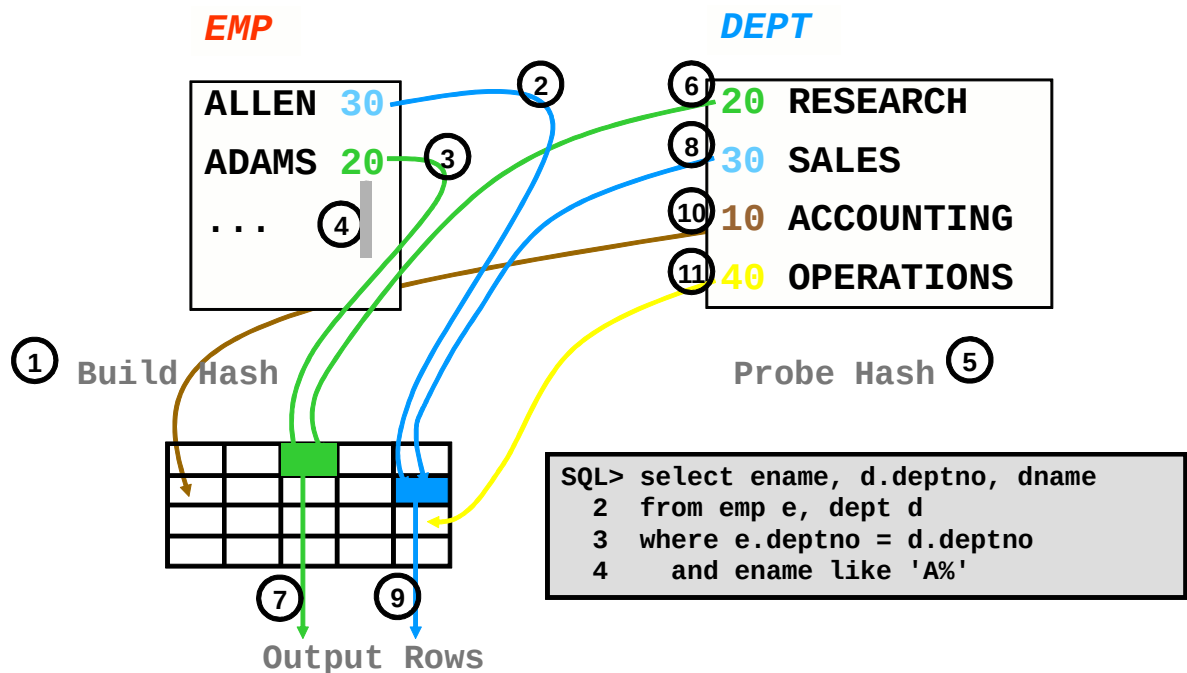
Sorting could make this join technique expensive, especially if sorting cannot be performed in memory.

The optimizer can choose a sort-merge join over a hash join for joining large amounts of data if either of the following conditions is true:

The join condition between two tables is not an equijoin.

If sorting is already required by other operations (or data is already pre-sorted), the optimizer may find it is cheaper to use a sort-merge than a hash join.

3.1.3 - Hash Join (1)



ORACLE

HASH JOIN (1)

The hash join is only available with CBO. Hash joins are performed *only* for *equijoins*.

- ① Oracle reads the first data set and builds an array of “hash buckets” in memory.

A hash bucket is little more than a location that acts as the starting point for a linked list of rows from the build table. A row belongs to a hash bucket if the bucket number matches the result that Oracle gets by applying an internal hashing function to the join column(s) of the row.

- ② hash 1st join value from EMP table
- ③ hash 2nd join value from EMP table
- ④ and so on

- ⑤ Oracle starts to read the second table, using whatever access mechanism is most appropriate for acquiring the rows

Oracle uses the same hash function on the join column(s) to calculate the number of the relevant hash bucket. Oracle then checks to see if there are any rows in that bucket—this is known as probing the hash table.

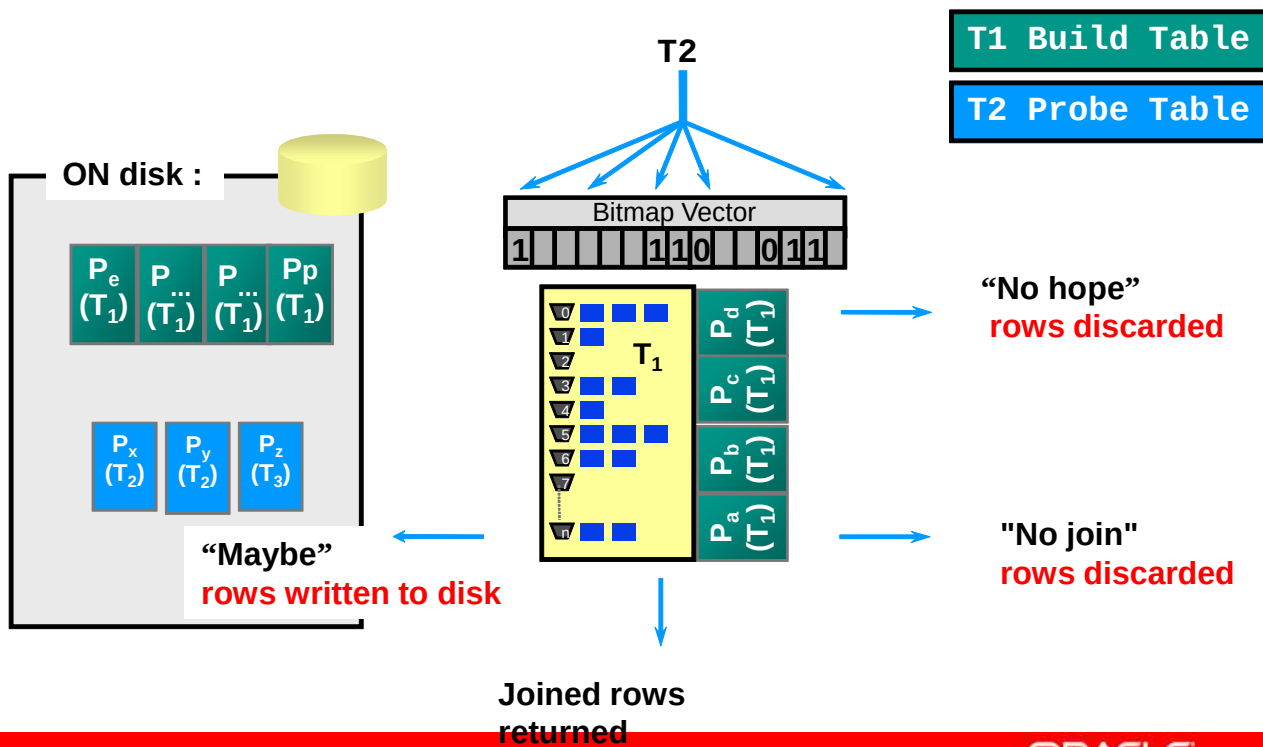
If there are some rows in the relevant bucket, Oracle does an exact check on the join column(s) to see if there is a proper match. It is always possible for two different input values to a hashing function to collide on the same output value. Since it is possible for rows with different values in the join column(s) to be in the same hash bucket, we have to do the exact check. Any rows that survive the exact check can immediately be reported (or passed on to the next step in the execution plan).

- ⑥ Oracle hash 1st row from Right Table DEPT(20)
- ⑦ 1 row is returned.
- ⑧ Oracle hash 2nd row from Right Table DEPT (30)
- ⑨ 1 row is returned.

If there are no rows in the relevant bucket, Oracle can immediately discard the row from the probe table.

- 10 Oracle hash 3rd row from Right Table DEPT (10)
 - 11 Oracle hash 4th row from Right Table DEPT (40)
-
- When you do a NESTED LOOPS, you'll return rows to next operation as soon as you find them
 - When you do a MERGE JOIN, you need to fetch ALL rows from the 2 row sources before to return the first row to the next operation.
 - When you do a HASH JOIN, you need to fetch ALL rows from the smallest row source before to return the first row to next operation.

Hash Join Detail (2)



HASH JOIN (2)

When the "Hash Table" from the "build" table (T1) is larger than the available memory, Oracle adopts a more complex strategy for handling the hash table.

The result is that some of the build table has to be dumped on disk. Oracle splits it into "p" disjoint partitions (using a new hash function HF_P). Some partitions are dumped to disk and subsequently reread.

To discard rows as soon as possible, A *bitmap vector* is built in memory.

1) Start with Build Table T1 scan, you have p input partitions (HF_P with p partitions)

- 4 of them (Pa,Pb,Pc,Pd) are loaded into memory and the corresponding hash table is built (HF with n buckets as previous slide).
- The p hash filters are stored in a bitmap format (bitmap vector), for all hash values occurring in all T1 input partitions.

2) Now the Probe table (T2) scan starts.

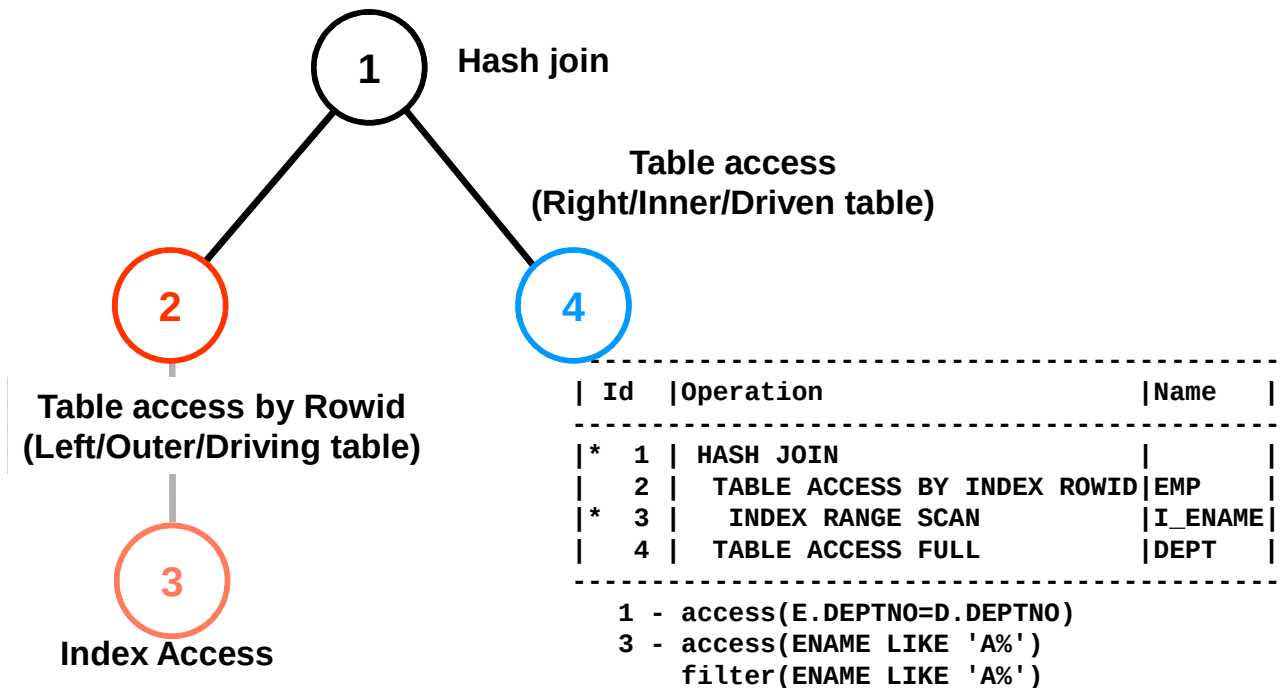
For each row the key value is hashed and checked against the bitmap.

- If the bitmap shows a zero (value does not occur) you are sure that this row will never match with any input table row; the row can be rejected (the “**no hope**” exit).
- If the bitmap shows that the value occurs (or might occur), and the corresponding partition to look for is Pa, Pb, Pc, or Pd, then there are two possibilities:
 - You find a match with the partitions a, b, c, or d (“**Joined rows returned**”)
 - You don’t find a match (“**No join rows discarded**”)
- If the bitmap shows a one for any partition that is currently not in memory, it is written to a temporary segment on disk based upon the partition (the “**maybe rows**”).

After the first pass, you end up with corresponding partition sets, and the same algorithm can be used again. When a partition couple is processed, the smallest one is used as the build input and largest one is used for probe; this could be the T2 or the T1 partition. This is also known as dynamic role reversal.

If the smaller of the two inputs is too large and does not fit in memory then read the smaller build input into memory in chunks and iterate over the probe. This is called the nested hash loops join.

Hash Join Plan (3)



ORACLE

HASH JOIN(3)

The slide illustrates a parse tree (sequence of steps) for the hash join.

Steps 2 and 4 perform table access of the tables that must be joined (EMP is accessed by an index).

Step 1 builds a hash table from the rows delivered by step 2 and probes it with each row coming from step 4.

Example

```
select /*+ leading(e) use_hash(d) index(e) */
       ename, e.deptno, d.deptno, dname
  from emp e, dept d
 where e.deptno = d.deptno and ename like 'A%';
```

ENAME

DEPTNO

DEPTNO DNAME

```
-----
ADAMS                20          20 RESEARCH
ALLEN                30          30 SALES
```

	Id	Operation	Name	Starts	E-Rows	A-Rows
*	1	HASH JOIN		1	3	2
	2	TABLE ACCESS BY INDEX ROWID	EMP	1	3	2
*	3	INDEX RANGE SCAN	I_ENAME	1	3	2
	4	TABLE ACCESS FULL	DEPT	1	4	4

```
1 - access("E"."DEPTNO"="D"."DEPTNO")
3 - access("ENAME" LIKE 'A%')
    filter("ENAME" LIKE 'A%')
```

When Are Hash Joins Used?

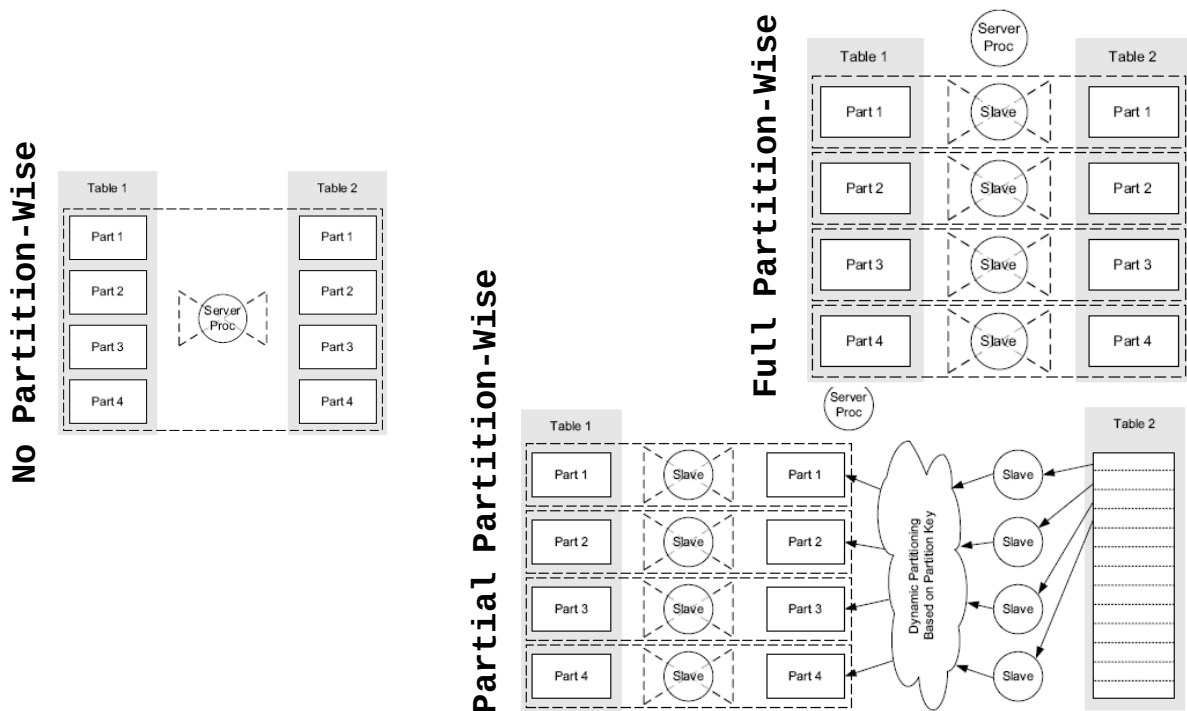
The optimizer uses a hash join to join two tables if they are joined using an equijoin and if either of the following conditions is true:

A large amount of data needs to be joined.

A large fraction of the table needs to be joined.

Apply the USE_HASH hint to advise the optimizer to use a hash join when joining two tables together.

Partition-Wise Join



ORACLE

Partition-Wise Join

Partitioning for Performance

By limiting the amount of data to be examined or operated on, partitioning provides a number of performance benefits. These features include:

Partitioning Pruning: Partitioning pruning (a.k.a. Partition elimination) is the simplest and also the most substantial means to improve performance using partitioning. Partition pruning can often improve query performance by several orders of magnitude.

Partition-wise Joins: Partitioning can also improve the performance of multi-table joins, by using a technique known as partition-wise joins. Partition-wise joins can be applied when two tables are being joined together, and at least one of these tables is partitioned on the join key. Partition-wise joins break a large join into smaller joins of 'identical' data sets for the joined tables. 'Identical' here is defined as covering exactly the same set of partitioning key values on both sides of the join, thus ensuring that only a join of these 'identical' data sets will produce a result and that other data sets do not have to be considered. Oracle is using either the fact of already (physical) equi-partitioned tables for the join or is transparently redistributing (= "repartitioning") one table at runtime to create equi-partitioned data sets matching the partitioning of the other table, completing the overall join in less time. This offers significant performance benefits both for serial and parallel execution.

With partition-wise joins, the join operation is broken into smaller joins that are performed sequentially or in parallel. Another way of looking at partition-wise joins is that they minimize the amount of data exchanged among parallel slaves during the execution of parallel joins by taking into account data distribution.

Examples Partition-wise Joins

```
drop table t1;
drop table t2;
drop table t3;
SET ECHO ON
REM
REM Setup test environment
REM
CREATE TABLE t1 (
  id NUMBER,
  d1 DATE,
  n1 NUMBER,
  n2 NUMBER,
  n3 NUMBER,
  pad VARCHAR2(4000),
  CONSTRAINT t1_pk PRIMARY KEY (id)
)
PARTITION BY HASH (id) PARTITIONS 4;
CREATE TABLE t2 (
  id NUMBER,
  d1 DATE,
  n1 NUMBER,
  n2 NUMBER,
  n3 NUMBER,
  pad VARCHAR2(4000),
  CONSTRAINT t2_pk PRIMARY KEY (id)
)
PARTITION BY HASH (id) PARTITIONS 4;
CREATE TABLE t3 (
  id NUMBER,
  d1 DATE,
  n1 NUMBER,
  n2 NUMBER,
  n3 NUMBER,
  pad VARCHAR2(4000),
  CONSTRAINT t3_pk PRIMARY KEY (id));

execute dbms_random.seed(0)

INSERT INTO t1
SELECT rownum AS id,
       trunc(to_date('2007-01-01','yyyy-mm-dd')+rownum/27.4) AS d1,
       1+mod(rownum,400) AS n1,
       255+mod(trunc(dbms_random.normal*1000),255) AS n2,
       round(4515+dbms_random.normal*1234) AS n3,
       dbms_random.string('p',255) AS pad
FROM dual
CONNECT BY level <= 10000
ORDER BY dbms_random.value;

INSERT INTO t2 select * from t1;
```

```

INSERT INTO t3 select * from t1;

BEGIN
  dbms_stats.gather_table_stats(
    ownname => user, tabname => 'T1', estimate_percent => 100,
    method_opt => 'for all columns size skewonly',
    cascade => TRUE
  );
END;
/

BEGIN
  dbms_stats.gather_table_stats(
    ownname => user, tabname => 'T2', estimate_percent => 100,
    method_opt => 'for all columns size skewonly',
    cascade => TRUE
  );
END;
/

BEGIN
  dbms_stats.gather_table_stats(
    ownname => user, tabname => 'T3', estimate_percent => 100,
    method_opt => 'for all columns size skewonly',
    cascade => TRUE
  );
END;
/

SELECT partition_name, partition_position, num_rows
FROM user_tab_partitions
WHERE table_name IN ('T1','T2','T3')
ORDER BY partition_position;

alter table t1 parallel 4;
alter table t2 parallel 4;
alter table t3 parallel 4;

select count(*) from (
  SELECT /*+ no_merge ordered use_hash(t2) full(T1) full(T2) pq_distribute(t2 none
partition) */
    * FROM t1, t2
    WHERE t2.id= t1.id );
@C

select count(*) from (
  SELECT /*+ no_merge ordered use_hash(t3) full(T1) full(T3) pq_distribute(t3 none
partition) */
    * FROM t1, t3
    WHERE t3.id= t1.id );
@C

```

FULL Partition-wise Joins

In the following example, T1 and T2 are joined on their hash partitioning columns. This enables use of full partition-wise join. The PX PARTITION HASH row source appears on top of the join row source in the plan table output. Each parallel slave performs the join of an entire hash partition of T1 with an entire partition of T2.

```
select count(*) from (
  SELECT /*+ no_merge ordered use_hash(t2) full(T1) full(T2) */ *
    FROM t1, t2
   WHERE t2.id= t1.id );
```

@C

Id	Operation	Name	E-Rows	Pstart	Pstop	TQ	IN-OUT	PQ Distrib	A-Rows
1	SORT AGGREGATE		1						1
2	PX COORDINATOR								4
3	PX SEND QC (RANDOM)	:TQ10000	1			Q1,00	P->S	QC (RAND)	0
4	SORT AGGREGATE		1			Q1,00	PCWP		0
5	PX PARTITION HASH ALL		10000	1	4	Q1,00	PCWC		0
6	VIEW		10000			Q1,00	PCWP		0
* 7	HASH JOIN		10000			Q1,00	PCWP		0
8	TABLE ACCESS FULL	T1	10000	1	4	Q1,00	PCWP		0
9	TABLE ACCESS FULL	T2	10000	1	4	Q1,00	PCWP		0

Predicate Information (identified by operation id):

```
7 - access("T2"."ID"="T1"."ID")
```

PARTIAL Partition-wise Joins

The execution plan shows that the table T3 is scanned serially and all rows with the same partitioning column value of T1 (id) are sent through a PART (KEY), or partition key, table queue to the same slave doing the partial partition-wise join.

In the following example, T1 is joined on the partitioning column and is parallelized. This enables use of partial partition-wise join, because the T3 table is not partitioned. Oracle

dynamically partitions the T3 table before the join. This implies that the table T3 is re-partitioned based on the join column id to be sent to the parallel slaves executing the scan of T1 and the join.

```
alter table t3 parallel 1;
select count(*) from (
  SELECT /*+ no_merge ordered use_hash(t3) full(T1) full(T3)
        pq_distribute(t3 none partition) */
    * FROM t1, t3
  WHERE t3.id= t1.id );
```

@C

Id	Operation	Name	E-Rows	Pstart	Pstop	TQ	IN-OUT	PQ Distrib	A-Rows

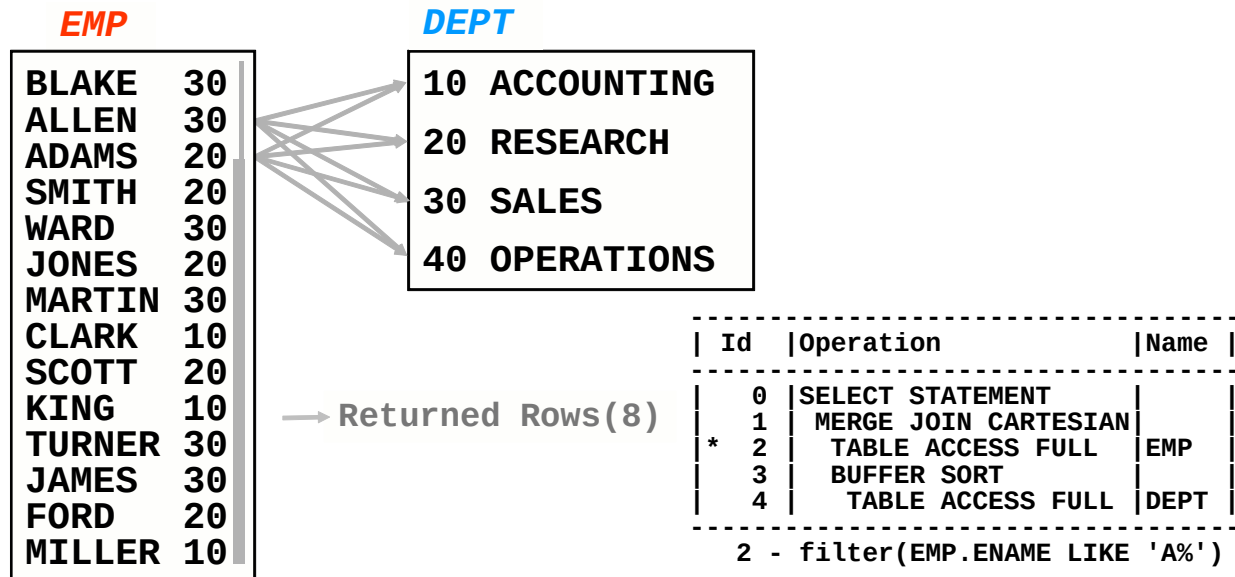
1	SORT AGGREGATE		1						1
2	PX COORDINATOR								4
3	PX SEND QC (RANDOM)	:TQ10001	1			Q1,01	P->S	QC (RAND)	0
4	SORT AGGREGATE		1			Q1,01	PCWP		0
5	VIEW		10000			Q1,01	PCWP		0
* 6	HASH JOIN		10000			Q1,01	PCWP		0
7	PX PARTITION HASH ALL		10000	1	4	Q1,01	PCWC		0
8	TABLE ACCESS FULL	T1	10000	1	4	Q1,01	PCWP		0
9	BUFFER SORT					Q1,01	PCWC		0
10	PX RECEIVE		10000			Q1,01	PCWP		0
11	PX SEND PARTITION (KEY)	:TQ10000	10000				S->P	PART (KEY)	0
12	TABLE ACCESS FULL	T3	10000						10000

Predicate Information (identified by operation id):

6 - access("T3"."ID"="T1"."ID")

3.1.4 - Cartesian "Join"

```
SQL> select ename, e.deptno, d.deptno, dname
       2   from emp e, dept d where ename like 'A%'
```



ORACLE

CARTESIAN PRODUCT

A Cartesian product is done when there are no join conditions between two row sources and there is no alternative method of accessing the data.

Not a true join as such

Typically a result of a coding mistake where a join is omitted

Can be useful in some circumstances.

It's like a nested loops join but with no elimination – first row source is read and then for every row, all the rows are returned from the other row source.

Example

```
select ename, emp.deptno, dept.deptno, dname
       from emp, dept
       where ename like 'A%';
```

ENAME	DEPTNO	DEPTNO	DNAME
ALLEN	30	10	ACCOUNTING
ALLEN	30	20	RESEARCH
ALLEN	30	30	SALES
ALLEN	30	40	OPERATIONS
ADAMS	20	10	ACCOUNTING
ADAMS	20	20	RESEARCH
ADAMS	20	30	SALES
ADAMS	20	40	OPERATIONS

```

explain plan for
select ename, emp.deptno, dept.deptno, dname
  from   emp, dept
 where  ename like 'A%';

```

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		21	651	10
1	MERGE JOIN CARTESIAN		21	651	10
* 2	TABLE ACCESS FULL	EMP	4	80	2
3	BUFFER SORT		5	55	8
4	TABLE ACCESS FULL	DEPT	5	55	2

2 - filter("EMP"."ENAME" LIKE 'A%')

The CARTESIAN keyword indicates that you are doing a Cartesian product.

Bug 2177511 : Interesting comments MJC and BUFFER (SORT) vs Nested Loops



3.2 - Join Types

- A join operation (whatever the join method) combines the output from two row sources and returns one resulting row source.
- Join operation types include the following :
 - Join (Equijoin – Non-equijoin)
 - Outer Join
 - Semi join - EXISTS Subquery
 - Anti Join - NOT IN Subquery
 - Star Join (Optimisation)

ORACLE

Join operation types include the following:

Join (equijoin and Non equijoin) Return rows that match predicate join

Outer Join Return rows that match predicate join AND row when no match is found

Semi join – processes EXISTS Subquery - Return rows that match EXISTS subquery. Only has to find one match in inner table, then stop search

Anti Join – processes NOT IN Subquery - Return rows from with no match in NOT IN subquery. Stop as soon as 1 match is found

Star Join (Optimisation to better handle a Data model FACT + Dimension table data model) – This is not a join type, just a name for an implementation of a performance optimisation.

Antijoin and Semijoin are considered to be joined types, even though the SQL constructs that cause them are subqueries. Antijoin and Semi join are internal optimizations algorithms used to “flatten” subquery constructs in such a way that they can be resolved in a join-like way

Join Type Choice

On 9i transformation in SEMI or ANTI is based on heuristic rules. When query is transformed in SEMI join, then (CBO) might consider to eliminate this SEMI join for Cost consideration (this does not exist with ANTI join)

On 10g, (CBO) chooses an appropriate join type/method based on calculated costs.

See more information in Query Transformation Section



3.2.1 - Join

- Equi join

```
SELECT e.ename, d.dname FROM emp e,  
dept d where e.deptno = d.deptno;
```

- NON Equi join

```
SELECT e.ename, e.sal, s.grade FROM  
emp e, salgrade s where e.sal between  
s.losal and s.hisal;
```

ORACLE

Join

The join condition determines whether a join is an equijoin or a non-equijoin. An equijoin is a join with a join condition containing an equality operator. When a join condition relates two tables by an operator other than equality, it is a non-equijoin.

Equijoins are the most commonly used. An example of an equijoin:

```
SELECT e.ename, d.dname  
FROM emp e INNER JOIN dept d ON e.deptno = d.deptno
```

```
SELECT e.ename, d.dname  
FROM emp e , dept d where e.deptno = d.deptno
```

```
SELECT /*+ ordered use_hash(s) */  
      e.ename, e.sal, s.grade  
FROM emp e ,salgrade s  
where e.sal = s.hisal;
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		477	5
* 1	HASH JOIN		477	5
2	TABLE ACCESS FULL	EMP	14	2
3	TABLE ACCESS FULL	SALGRADE	409	2

1 - access(E.SAL = S.HISAL)

Non-equi joins are less frequently used.

If you have a non-equi join (theta-join), a hash join is not possible.

```
SELECT /*+ ordered use_hash(s) */
       e.ename, e.sal, s.grade
FROM   emp e ,salgrade s
where  e.sal between s.hisal and s.hisal;
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		14	30
1	NESTED LOOPS		14	30
2	TABLE ACCESS FULL	EMP	14	2
* 3	TABLE ACCESS FULL	SALGRADE	1	2

3 - filter(E.SAL >= S.HISAL AND E.SAL <= S.HISAL)

3.2.2 - Outer Joins

- A join returns rows where criteria match the specified join condition:
 - where emp.deptno = dept.deptno
- An outer join also returns a row if no match is found:
 - where emp.deptno(+)=dept.deptno

DEPTNO	DNAME	EMPNO	ENAME
10	ACCOUNTING	7782	CLARK
30	SALES	7900	JAMES
40	OPERATIONS		

ORACLE

Join Types: Outer Joins

The simple join is the most commonly used within Oracle. Other joins open up extra functionality but have much more specialized uses.

The outer join operator is placed on the deficient side of the query. In other words, it is placed against the table that has the missing join information.

Outer Join Example

Consider EMP and DEPT. There may be a department that has NO employees. If EMP and DEPT are joined together this particular department would not appear in the output because there is no row that matches the join condition for that department. By using the outer join the missing department can be displayed.

Left column from below chart is showing output from Join

Right Column from below chart is showing output from OUTER join

select d.deptno,d.dname,e.empno	select d.deptno,d.dname,e.empno
,e.ename from emp e, dept d	,e.ename from emp e, dept d
where e.deptno=d.deptno;	where e.deptno(+) =d.deptno;

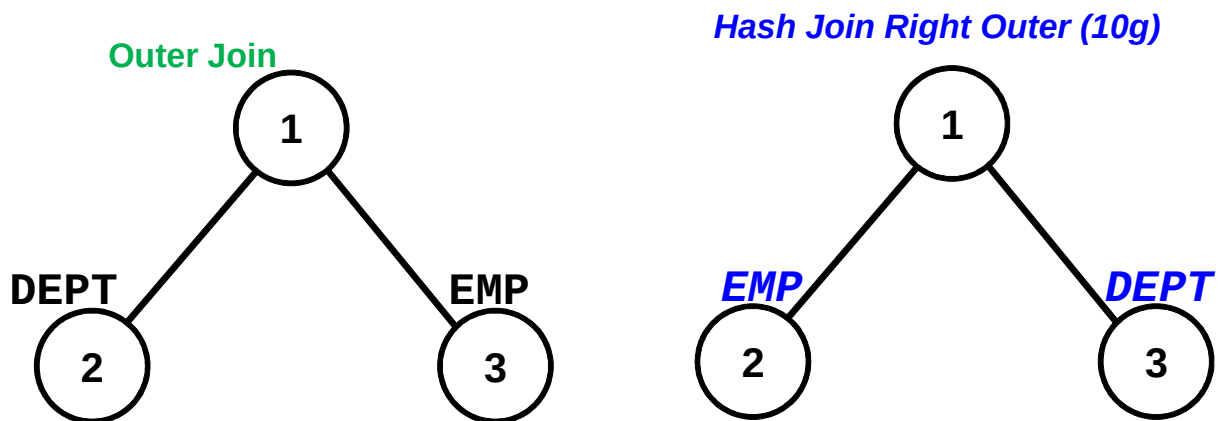
**

-----	-----
DEPTNO DNAME EMPNO ENAME	DEPTNO DNAME EMPNO ENAME
-----	-----
10 ACCOUNTING 7934 MILLER	10 ACCOUNTING 7782 CLARK
10 ACCOUNTING 7839 KING	10 ACCOUNTING 7839 KING
10 ACCOUNTING 7782 CLARK	10 ACCOUNTING 7934 MILLER
20 RESEARCH 7902 FORD	20 RESEARCH 7369 SMITH
20 RESEARCH 7876 ADAMS	20 RESEARCH 7566 JONES
20 RESEARCH 7788 SCOTT	20 RESEARCH 7788 SCOTT
20 RESEARCH 7566 JONES	20 RESEARCH 7876 ADAMS
20 RESEARCH 7369 SMITH	20 RESEARCH 7902 FORD
30 SALES 7900 JAMES	30 SALES 7499 ALLEN
30 SALES 7844 TURNER	30 SALES 7521 WARD
30 SALES 7698 BLAKE	30 SALES 7654 MARTIN
30 SALES 7654 MARTIN	30 SALES 7698 BLAKE
30 SALES 7521 WARD	30 SALES 7844 TURNER
30 SALES 7499 ALLEN	30 SALES 7900 JAMES
	40 OPERATIONS

**

1.3.2 - Outer Joins (2)

- where emp.deptno(+) = dept.deptno
 - outer join
 - Hash Join Right Outer



Join Types: Outer Joins

```
select d.deptno,d.dname,e.empno
,e.ename from emp e, dept d
where e.deptno(+) = d.deptno;
```

The simple join is the most commonly used within Oracle. Other joins open up extra functionality but have much more specialized uses.

The outer join operator is placed on the deficient side of the query. In other words, it is placed against the table that has the missing join information. HASH JOIN (RIGHT OUTER) is an internal optimization to perform the outer join.

Outer Join with Nested Loops :

Left/driving Table is always table whose rows are being preserved (DEPT in our example).

- For each row from DEPT
- Look for a matching row in EMP

- If none is found, output DEPT values with NULL values for the EMP columns
- If row is found, output DEPT values with these EMP values

Hash Outer joins :

Until 9i, The Left/Outer table whose rows are being preserved, is used to build the hash table, and the Right/Inner table is used to probe the hash table.

- DEPT table is scanned and hashed
- EMP table is scanned and for each row
- DEPT hash table is probed for matches. When a match is found, the row is output and the entry in the hash table is marked as matched to a row.
- After EMP table is exhausted, DEPT hash table is gone over once again, and any rows that are not marked as matched are output with NULL values for the EMP columns.

Since 10g, HASH JOIN (RIGHT OUTER) is available. Oracle hashes table whose rows are **NOT** being preserved, and then read the table whose rows are being preserved, probing the hash table to see whether there was a row to join to.

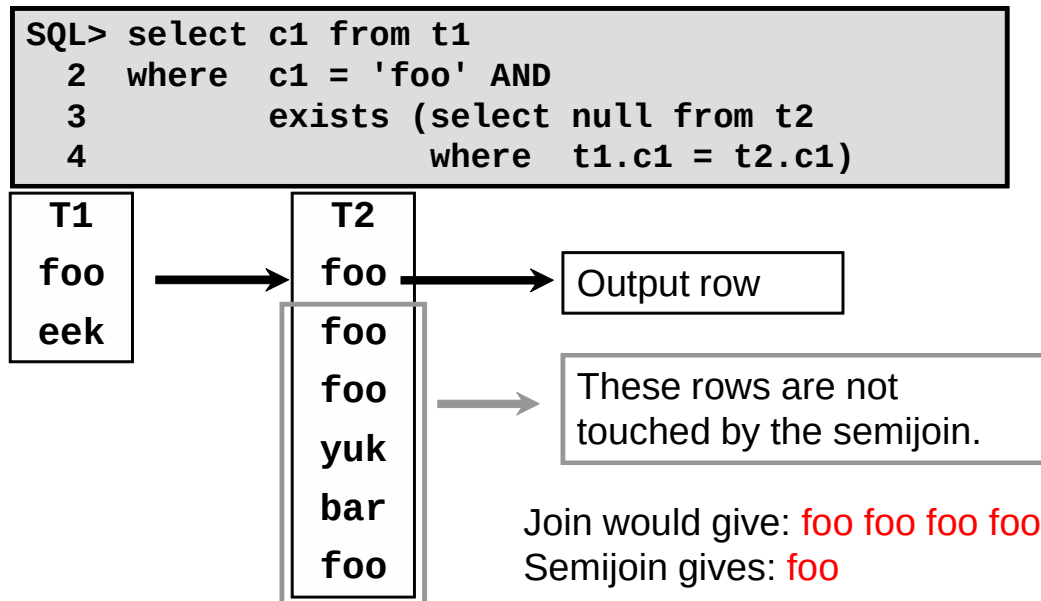
```
SQL_ID 1zdV488aktwCg, child number 0
-----
select /*+ leading(e) USE_HASH(d) SWAP_JOIN_INPUTS(e) full(e) full(d) */
d.deptno,d.dname,e.empno,e.ename from emp e, dept d where
e.deptno(+) = d.deptno
Plan hash value: 3590956717
-----
| Id | Operation          | Name | St | E-Ro | Cost (%CPU) | E-Time | A-Rows |
-----
|* 1 | HASH JOIN RIGHT OUTER|      | 1 | 14 |      7 (15) | 00:00:01 | 15 |
| 2 | TABLE ACCESS FULL | EMP  | 1 | 14 |      3 (0) | 00:00:01 | 14 |
| 3 | TABLE ACCESS FULL | DEPT | 1 | 4 |      3 (0) | 00:00:01 | 4 |
-----
Predicate Information (identified by operation id):
-----
1 - access("E"."DEPTNO"="D"."DEPTNO")
```

Here, hint SWAP_JOIN_INPUTS has been used to force HASH JOIN RIGHT OUTER and swap inputs from hash join.

Source Blog From Dev Team : <http://optimizermagic.blogspot.com/search?updated-max=2008-01-22T15%3A34%3A00-08%3A00&max-results=7>

3.2.3 - Semi-Joins

Semijoins only look for the first match



ORACLE

Semi - Joins

Semijoins return a result when you hit the first joining record. A semijoin is an internal way of transforming an exists subquery into a join; there is nowhere that you can see this occurring.

Semijoin returns rows that match an EXISTS subquery without duplicating rows from the left side of the predicate when multiple rows on the right side satisfy the criteria of the subquery.

Example:

In the diagram, only the first `foo` record is returned as a join result. This prevents scanning huge numbers of duplicate rows in a table when all you are interested in is if there are any matches.

Note: When subquery is NOT unnested, similar result could be achieved by using a FILTER operation and scanning a row source until a match was found, then returning.

The optimizer can use nested loops, merge-join or hash-join algorithm to perform semi-join.

```

SeLeCt deptno, dname
from dept
where exists (select 1
              from emp
              where emp.deptno=dept.deptno);

```

```

DEPTNO DNAME
-----

```

```

    20 RESEARCH
    30 SALES
    10 ACCOUNTING

```

Id	Operation	Name	Rows	Cost

0	SELECT STATEMENT			5
* 1	HASH JOIN SEMI		409	5
2	TABLE ACCESS FULL	DEPT	409	2
3	TABLE ACCESS FULL	EMP	14	2

```

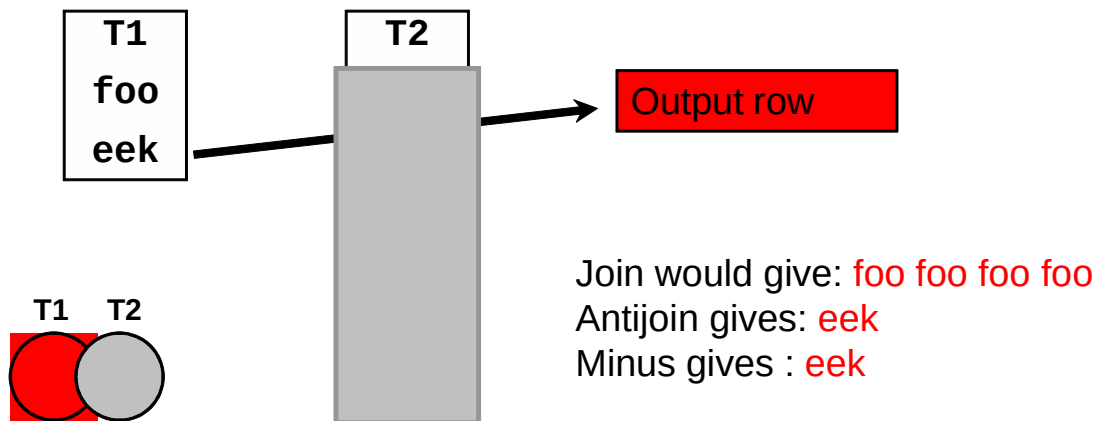
1 - access("DEPTNO"="DEPTNO")

```

3.2.4 - Antijoins

```
SQL> select c1 from t1
2 where c1 not in (select c1 from t2)
```

Reverse of what would have been returned by a join: in case of a match the Oracle server does *not* return a row.



ORACLE

Antijoin

Returns rows that fail to match (NOT IN) the subquery on the right side.

For example, an antijoin can select a list of department which do not have any employee:

```
select deptno, dname
  from dept
 where deptno not in (select deptno from emp);
```

The optimizer uses a nested loops algorithm for NOT IN subqueries by default. However, if the MERGE_AJ, HASH_AJ, or NL_AJ hint is used and various required conditions are met, the NOT IN uncorrelated subquery can be changed into a sort merge or hash antijoin.

Although antijoins are mostly transparent to the user, it is useful to know that these join types exist and could help explain unexpected performance changes between releases.

```

alter table emp modify deptno not null;
create index I_DEPTNO on EMP(DEPTNO);
select deptno, dname from dept
where deptno not in (select deptno from emp);

```

DEPTNO DNAME

40 OPERATIONS

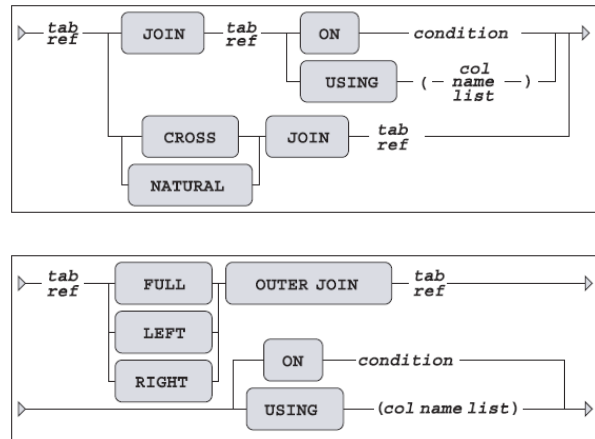
Id	Operation	Name	Starts	E-Rows	A-Rows	

1	NESTED LOOPS ANTI		1	1	1	
2	TABLE ACCESS FULL	DEPT	1	4	4	
* 3	INDEX RANGE SCAN	I_DEPTNO	4	9	3	

3 - access("DEPTNO"="DEPTNO")

3.2.5 - ANSI Join Syntax

- Cross / Natural Join
- Left / Right / Full Outer Join



ORACLE

ANSI JOIN SYNTAX

A CROSS join is the cross-product of two tables. It is the equivalent of a Cartesian product.

A NATURAL join selects rows from the tables which have equal values in all matched columns (same column names). If the columns having the same names have different datatypes, an error is returned.

The join of two tables returning only matched rows is considered an INNER JOIN.

A join between two tables which returns the result of the INNER join as well as unmatched rows from the LEFT (or RIGHT) table is a LEFT (or RIGHT) OUTER join. INNER and OUTER JOINS were available in prior releases. The '+' symbol was used on one of the tables to create outer joins.

FULL Outer Join returns every row from Left Table, as well every row in Right Table. There is NO oracle syntax equivalence.

Note 137286.1 Oracle9i ANSI/ISO SQL:1999-Compliant SQL

Use ANSI JOIN syntax might lead to use CBO (See **Bug 3101779**)

<http://optimizermagic.blogspot.com/2007/12/outerjoins-in-oracle.html>



4 - Other N-ary Operations

- **FILTER**
- **CONCATENATION**
- **CONNECT BY**
- **UNION ALL / UNION**
- **INTERSECT**
- **MINUS**

ORACLE

Unary Operations

1 - Access Path

2 - Other Unary Operations

Binary Operations

3 - Joins

N-ary Operations

4 - Other N-ary Operations

Unary Operations : operations that act on only 1 input

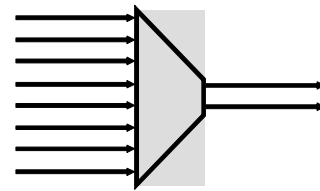
Binary Operations : operations that act on 2 inputs like Joins

N-ary Operations : operations that act several inputs like Relational Operator



4.1 – Filter (1)

- Accepts a set of rows
- Eliminates some of them
- Returns the rest



ORACLE

FILTER

Operation accepting a set of rows, eliminates some of them, and returns the rest.

Example WITHOUT FILTER operation

```
select DEPTNO, sum(sal) SUM_SAL
  from emp group by deptno ;
```

DEPTNO	SUM_SAL	Id	Operation	Name	Rows	Cost(%CPU)
30	9400	0	SELECT STATEMENT			4 (100)
20	10875	1	HASH GROUP BY		3	4 (25)
10	8750	2	TABLE ACCESS FULL	EMP	14	3 (0)

Example WITH FILTER operation

```
select DEPTNO, sum(sal) SUM_SAL from emp
group by deptno having sum(sal) > 9000;
```

DEPTNO	SUM_SAL	-----					
		Id	Operation	Name	Rows	Cost(%CPU)	
-----		-----					
30	9400	0	SELECT STATEMENT			4 (100)	
20	10875	*	1 FILTER				
		2	HASH GROUP BY		3	4 (25)	
		3	TABLE ACCESS FULL	EMP	14	3 (0)	

1 - filter(SUM("SAL")>9000)							

→ Filtered Row is deptno=10 (because sum(sal)=8750)

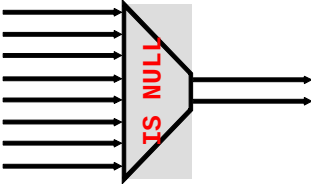


Filter (2)

```
select deptno, dname from dept
where not exists (select 1 from emp
                  where deptno=dept.deptno);
```

Id	Operation	Name	Starts	E-Rows	A-Rows
* 1	FILTER		1		1
2	TABLE ACCESS FULL	DEPT	1	4	4
* 3	INDEX RANGE SCAN	I_DEPTNO	4	2	3

- 1 - filter(IS NULL)
- 3 - access(EMP.DEPTNO=:B1)



filter is [NOT EXISTS (select 1 from emp
where emp.deptno = :b1)]

Filter(2)

A **FILTER** operation is any operation that discards rows returned by another step but is not involved in retrieving the rows itself. All sorts of operations can be filters, including sub-queries and single table predicates.

See Column explanation from Actual PLaN in slide

```
SeLeCt deptno, dname
  from dept
 where not exists (select 1 from emp where emp.deptno=dept.deptno);
```

```
DEPTNO DNAME
-----
40 OPERATIONS
```

- DEPT is accessed once (See plan in Slide) (Operation Id=2 – Starts=1)
- For each row from DEPT, EMP is accessed by its index on deptno (I_DEPTNO)
- Operation 3 checks if dept.deptno exists in emp table (emp.deptno = :B1 is TRUE)
- Operation 3 is done as many as the number of rows in DEPT (Starts = 4)
- The FILTER operation is applied, for each row, AFTER DEPT rows are fetched and discard rows for which operation "3" returned at least 1 row
"NOT EXIST (select 1 from emp where emp.deptno=:B1)" is TRUE
 -> deptno 10 – 20 and 30 are filtered /discarded

```
SeLeCt deptno, dname from dept
where exists (select 1 from emp where emp.deptno=dept.deptno);
DEPTNO DNAME
```

```
10 ACCOUNTING
20 RESEARCH
30 SALES
```

Id		Operation	Name	Starts	E-Rows	A-Rows

*	1	FILTER		1		3
	2	TABLE ACCESS FULL	DEPT	1	4	4
*	3	INDEX RANGE SCAN	I_D	4	2	3

- ```
1 - filter(IS NOT NULL)
3 - access("EMP"."DEPTNO"=:B1)
```

(IS NOT NULL) is TRUE if [EXISTS (select 1 from emp where emp.deptno = det.deptno)] is TRUE



## 4.2 - Concatenation

```
create index I_SAL on emp(sal);
create index I_DEPTNO on emp(deptno);
```

```
select /*+ USE_CONCAT */ * from emp where deptno =1 or sal =2 ;
```

| Id  | Operation                   | Name     | Rows | Bytes | Cost |
|-----|-----------------------------|----------|------|-------|------|
| 0   | SELECT STATEMENT            |          | 3    | 111   | 4    |
| 1   | CONCATENATION               |          |      |       |      |
| 2   | TABLE ACCESS BY INDEX ROWID | EMP      | 1    | 37    | 2    |
| * 3 | INDEX RANGE SCAN            | I_SAL    | 1    |       | 1    |
| * 4 | TABLE ACCESS BY INDEX ROWID | EMP      | 2    | 74    | 2    |
| * 5 | INDEX RANGE SCAN            | I_DEPTNO | 2    |       | 1    |

```
3 - access(SAL=2)
** 4 - filter(LNNVL(SAL=2))
5 - access(DEPTNO=1)

** LNNVL generated by 'OR'
```

ORACLE

### CONCATENATION

Concatenates the rows returned by two or more rowsets.

This works like an UNION ALL and does not remove duplicate.

It is used with OR expansions

BUT OR does not return duplicate rows,

→ so for each component it appends a **negation** of the previous components(LNNVL.)

### CONCATENATION

- BRANCH 1 - SAL=2
- BRANCH 2 - DEPTNO = 1 AND NOT row in Branch 1

LNNVL function is generated by 'OR' clause to process this negation.

The function **LNNVL()** returns true if the predicate is NULL or FALSE

Hence filter(**LNNVL**(SAL=2)) : will return all rows for which **SAL != 2** or **SAL is NULL**

-----

```
select /*+ USE_CONCAT */ *
 from emp
 where deptno =1 or sal =2 or ename like 'S%';
```

| EMPNO | ENAME | JOB     | MGR  | HIREDATE  | SAL  | DEPTNO |
|-------|-------|---------|------|-----------|------|--------|
| 7788  | SCOTT | ANALYST | 7566 | 19-APR-87 | 3000 | 20     |
| 7369  | SMITH | CLERK   | 7902 | 17-DEC-80 | 800  | 20     |

| Id  | Operation                   | Name  | Rows | Cost (%CPU) |
|-----|-----------------------------|-------|------|-------------|
| 0   | SELECT STATEMENT            |       |      | 6 (100)     |
| 1   | CONCATENATION               |       |      |             |
| 2   | TABLE ACCESS BY INDEX ROWID | EMP   | 1    | 2 (0)       |
| * 3 | INDEX RANGE SCAN            | I_SAL | 1    | 1 (0)       |
| * 4 | TABLE ACCESS BY INDEX ROWID | EMP   | 1    | 2 (0)       |
| * 5 | INDEX RANGE SCAN            | I_DPT | 1    | 1 (0)       |
| * 6 | TABLE ACCESS BY INDEX ROWID | EMP   | 2    | 2 (0)       |
| * 7 | INDEX RANGE SCAN            | I_ENM | 2    | 1 (0)       |

- 3 - access("SAL"=2)
- 4 - filter(LNNVL("SAL"=2))
- 5 - access("DEPTNO"=1)
- 6 - filter((LNNVL("SAL"=2) AND LNNVL("DEPTNO"=1)))
- 7 - access("ENAME" LIKE 'S%')
- filter("ENAME" LIKE 'S%')

**Bug 5129407** Suboptimal plan for SQL with multiple OR branches is explaining when/how Or-expansion is done or not (\*\*\*) JKLEBANE 08/31/06 02:25 pm (\*\*\*)



## 4.3 - Connect by

- connect by "without filtering"
  - 2 child operations
    - start with
    - Connect By
- connect by "with filtering"
  - 3 child operations
    - start with
    - Connect By
    - switch to no filtering, only if PGA is too small

ORACLE

### **CONNECT BY**

Queries containing a CONNECT BY clause must return the rows in a hierarchical structure.

The START WITH and CONNECT BY clauses define how to:

- Identify a starting point (root) for the tree structure.
- Specify how you can walk up or down the tree structure from any row.

**START WITH** : Specifies a condition that identifies the row(s) to be used at the root(s) of a hierarchical query. The START WITH clause can contain many rows and so there can be several trees.

**CONNECT BY** : produces a hierarchy in depth-first order from a sorted set of rows. It has two methods for collecting this set of rows into the sort: with filtering and without filtering.

When using **no-filtering**, connect by just fetches all the "base" rows into the sort.

When using **filtering**, connect by fetches one level at a time (essentially a breadth-first approach). In order for the filtering method to work, duplicates need to be detected at the time the rows are put in

the sort (connect by operation), otherwise connect by may enter an infinite loop (see bug 2425469).

Between 9.2.0.7 & 10.2.0.4 duplicates are not detected at the time the rows are put in the sort. And so, "connect by" will switch to the no-filtering method if the sort has to spill to disk (that is the 3rd branch of CONNECT BY which get blocks if SORT spill on disk)

With Fix from bug 5065418, duplicates are detected when rows are inserted into SORT even if it has to spil to disk (This fix does not remove this 3rd Branch in 10g. 3rd Branch is removed in 11g).

[http://download-uk.oracle.com/docs/cd/B19306\\_01/server.102/b14200/queries003.htm](http://download-uk.oracle.com/docs/cd/B19306_01/server.102/b14200/queries003.htm)

Some nice examples : <http://www.psoug.org/reference/connectby.html>

### ***Example of Query using CONNECT BY clause.***

### ***Resultset can be generated by any of below plans***

```
SELECT LPAD(' ', 2*LEVEL, ' ') || ename empName, job
, sys_connect_by_path(ename, '/') cbp
FROM emp e
START with ename='KING' CONNECT BY PRIOR empno = mgr;
```

| EMPNAME | JOB       | CBP                     |
|---------|-----------|-------------------------|
| KING    | PRESIDENT | /KING                   |
| JONES   | MANAGER   | /KING/JONES             |
| SCOTT   | ANALYST   | /KING/JONES/SCOTT       |
| ADAMS   | CLERK     | /KING/JONES/SCOTT/ADAMS |
| FORD    | ANALYST   | /KING/JONES/FORD        |
| SMITH   | CLERK     | /KING/JONES/FORD/SMITH  |
| BLAKE   | MANAGER   | /KING/BLAKE             |
| ALLEN   | SALESMAN  | /KING/BLAKE/ALLEN       |
| WARD    | SALESMAN  | /KING/BLAKE/WARD        |
| MARTIN  | SALESMAN  | /KING/BLAKE/MARTIN      |
| TURNER  | SALESMAN  | /KING/BLAKE/TURNER      |
| JAMES   | CLERK     | /KING/BLAKE/JAMES       |
| CLARK   | MANAGER   | /KING/CLARK             |
| MILLER  | CLERK     | /KING/CLARK/MILLER      |

14 rows selected.

### ***with hint /\*+ CONNECT\_BY\_FILTERING \*/***

When using filtering, connect by fetches one level at a time (essentially a breadth-first approach).

- CONNECT BY PUMP starts as many as level
- For each level (here 4)
- do the Join either by Hash Join or Nested Loops if Index on EMP.MGR

| -----  |                           |      |    |        |      |                     |
|--------|---------------------------|------|----|--------|------|---------------------|
| Id     | Operation                 | Name | St | A-Rows | Buff |                     |
| -----  |                           |      |    |        |      |                     |
| * 1    | CONNECT BY WITH FILTERING |      | 1  | 14     | 21   |                     |
| * 2    | START WITH clause         | EMP  | 1  | 1      | 7    | * START with Clause |
| 3      | Join                      |      | 4  | 13     | 14   |                     |
| 5      | CONNECT BY PUMP           |      | 4  | 14     | 0    |                     |
| 6      | TABLE ACCESS              | EMP  | 14 | 13     | 14   |                     |
| 8      | TABLE ACCESS FULL         | EMP  | 0  | 0      | 0    | ** 3rd -----        |
| -----  |                           |      |    |        |      |                     |
| Branch |                           |      |    |        |      |                     |

## Nested loops example

| -----                      |                             |       |    |        |      |                     |
|----------------------------|-----------------------------|-------|----|--------|------|---------------------|
| Id                         | Operation                   | Name  | St | A-Rows | Buff |                     |
| -----                      |                             |       |    |        |      |                     |
| * 1                        | CONNECT BY WITH FILTERING   |       | 1  | 14     | 21   |                     |
| * 2                        | TABLE ACCESS FULL           | EMP   | 1  | 1      | 7    | * START with Clause |
| 3                          | NESTED LOOPS                |       | 4  | 13     | 14   |                     |
| 4                          | BUFFER SORT                 |       | 4  | 14     | 0    |                     |
| 5                          | CONNECT BY PUMP             |       | 4  | 14     | 0    |                     |
| 6                          | TABLE ACCESS BY INDEX ROWID | EMP   | 14 | 13     | 14   |                     |
| * 7                        | INDEX RANGE SCAN            | I_MGR | 14 | 13     | 4    |                     |
| 8                          | TABLE ACCESS FULL           | EMP   | 0  | 0      | 0    | ** 3rd Branch       |
| -----                      |                             |       |    |        |      |                     |
| 1 - access(MGR=PRIOR NULL) |                             |       |    |        |      |                     |
| 2 - filter(ENAME='KING')   |                             |       |    |        |      |                     |
| 7 - access(MGR=PRIOR NULL) |                             |       |    |        |      |                     |

## Hash Join example

| ----- |                           |     |    |        |      |                     |
|-------|---------------------------|-----|----|--------|------|---------------------|
| Id    | Operation                 | Nam | St | A-Rows | Buff |                     |
| ----- |                           |     |    |        |      |                     |
| * 1   | CONNECT BY WITH FILTERING |     | 1  | 14     | 35   |                     |
| * 2   | TABLE ACCESS FULL         | EMP | 1  | 1      | 7    | * START with Clause |
| * 3   | HASH JOIN                 |     | 4  | 13     | 28   |                     |
| 4     | CONNECT BY PUMP           |     | 4  | 14     | 0    |                     |

|   |                   |     |   |    |    |                                                 |
|---|-------------------|-----|---|----|----|-------------------------------------------------|
| 5 | TABLE ACCESS FULL | EMP | 4 | 56 | 28 |                                                 |
| 6 | TABLE ACCESS FULL | EMP | 0 | 0  | 0  | ** 3rd Branch only if Id=1 has to spill on disk |

-----

1 - access(MGR=PRIOR NULL)  
2 - filter(ENAME='KING')  
3 - access(MGR=PRIOR NULL)

## 11g plan (3rd Child branch has been removed)

| Id  | Operation                 | Name | Starts | A-Rows | Buffers |
|-----|---------------------------|------|--------|--------|---------|
| * 1 | CONNECT BY WITH FILTERING |      | 1      | 14     | 35      |
| * 2 | TABLE ACCESS FULL         | EMP  | 1      | 1      | 7       |
| * 3 | HASH JOIN                 |      | 4      | 13     | 28      |
| 4   | CONNECT BY PUMP           |      | 4      | 14     | 0       |
| 5   | TABLE ACCESS FULL         | EMP  | 4      | 56     | 28      |

-----

1 - access("MGR"=PRIOR NULL)  
2 - filter("ENAME"='KING')  
3 - access("MGR"=PRIOR NULL)

## With hint /\*+ NO\_CONNECT\_BY\_FILTERING \*/

When using no-filtering, connect by just fetches all the "base" (Id = 3) rows into the sort operation (Id = 1)

- Id = 2 -> START clause
- Id = 1 For each row returned from Id = 2, do CONNECT-BY

| Id  | Operation                    | Nam | Starts | A-Rows | Buff |
|-----|------------------------------|-----|--------|--------|------|
| * 1 | CONNECT BY WITHOUT FILTERING |     | 1      | 14     | 14   |
| * 2 | START WITH clause            | EMP | 1      | 1      | 7    |
| 3   | TABLE ACCESS FULL            | EMP | 1      | 14     | 7    |

-----

1 - access("MGR"=PRIOR NULL)  
2 - access("ENAME"='KING')

| Id | Operation | Nam | Starts | A-Rows | Buff |
|----|-----------|-----|--------|--------|------|
|----|-----------|-----|--------|--------|------|

-----

|   |   |  |                              |  |     |  |    |  |    |  |
|---|---|--|------------------------------|--|-----|--|----|--|----|--|
| * | 1 |  | CONNECT BY WITHOUT FILTERING |  | 1   |  | 14 |  | 14 |  |
| * | 2 |  | TABLE ACCESS FULL            |  | EMP |  | 1  |  | 7  |  |
|   | 3 |  | TABLE ACCESS FULL            |  | EMP |  | 14 |  | 7  |  |

- 
- 1 - access(MGR=PRIOR NULL)
  - 2 - filter(ENAME='KING')

***With fix from Bug 5061079, plan becomes***

|       |    |                                           |      |        |               |
|-------|----|-------------------------------------------|------|--------|---------------|
| ----- |    |                                           |      |        |               |
| -     |    |                                           |      |        |               |
|       | Id | Operation                                 | Name | Starts | A-Rows   Buff |
|       |    |                                           |      |        |               |
| ----- |    |                                           |      |        |               |
| -     |    |                                           |      |        |               |
|       | *  | 1 CONNECT BY NO FILTERING WITH START-WITH |      | 1      | 14   7        |
|       |    |                                           |      |        |               |
|       |    | 2  TABLE ACCESS FULL                      | EMP  | 1      | 14   7        |
|       |    |                                           |      |        |               |

- 
- 1 - access("MGR"=PRIOR NULL)
  - filter(ENAME='KING')

***With hint /\*+ NO\_FILTERING NO\_CONNECT\_BY\_COST\_BASED \*/***

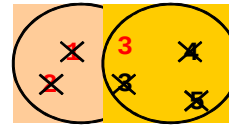
| Id |   | Operation                    | Name | Starts | A-Rows |
|----|---|------------------------------|------|--------|--------|
|    |   |                              |      |        |        |
| *  | 1 | CONNECT BY WITHOUT FILTERING |      | 1      | 14     |
| *  | 2 | FILTER                       |      | 1      | 1      |
|    | 3 | TABLE ACCESS FULL            | EMP  | 1      | 14     |
|    | 4 | TABLE ACCESS FULL            | EMP  | 1      | 14     |

- 1 -access("MGR"=PRIOR NULL)
- 2 - filter("ENAME"='KING')

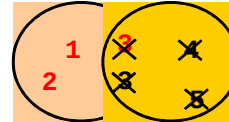
same as previous case with Cost Based Transformation, except START clause which is applied as a FILTER.

## 4.4 - Intersect - Minus - Union (all)

- **INTERSECT**

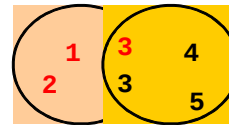


- **MINUS**



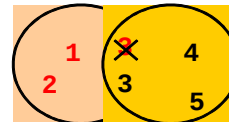
- **UNION ALL**

- Accepts sets of rows
- Applies union rules to determine the output
- Does not remove duplicates



- **UNION**

- Accepts sets of rows
- Applies union rules to determine the output
- removes duplicates



ORACLE

### ***Union (all) - Intersect - Minus***

SQL is a language that is supposed to be based on sets. Dr. Codd even defined the classic set operations as part of his eight basic operators for a relational database. Yet we did not have a full collection of basic set operations until the SQL-92 Standard.

In SQL, there is no pure set theory : SQL tables are multisets (also called bags), which means that, unlike sets, they allow duplicate elements (rows or tuples). Dr. Codd's relational model is stricter and uses only true sets. SQL handles these duplicate rows with an ALL or DISTINCT modifier in different places in the language; ALL preserves duplicates and DISTINCT removes them.

[http://en.wikipedia.org/wiki/Relational\\_algebra](http://en.wikipedia.org/wiki/Relational_algebra)

### ***INTERSECTION***

Operation accepting two sets of rows and returning the intersection of the sets, eliminating duplicates.

- Sub row sources are all executed/optimized individually
- Very similar to Sort-Merge-Join processing

- Full rows are sorted and matched

## **MINUS**

Operation accepting two sets of rows and returning rows appearing in the first set but not in the second, eliminating duplicates.

- Sub row sources are all executed/optimized individually
- Similar to INTERSECT processing
- Instead of match-and-return, match-and-exclude
- 

## **UNION ALL**

Operation accepting two sets of rows and returns the union of the sets, duplicates NOT eliminated.

Result is a 'bag', not a set

- (expensive) Sort-operator not necessary
- Use UNION-ALL if you know the bag is a set. (saving an expensive sort)

```
select e.empno,ename,sal, d.dname from emp e, dept d
 where d.deptno=e.deptno and d.deptno=10
union all
select e.empno,ename,sal, d.dname from emp e, dept d
 where d.deptno=e.deptno and d.deptno=20
```

| Rows | Row Source Operation                                         |
|------|--------------------------------------------------------------|
| 11   | UNION-ALL (cr=17 pr=0 pw=0 time=660 us)                      |
| 2    | NESTED LOOPS (cr=6 pr=0 pw=0 time=188 us)                    |
| 1    | TABLE ACCESS BY INDEX ROWID DEPT (cr=2 pr=0 pw=0 time=57 us) |
| 1    | INDEX UNIQUE SCAN PK_DEPT (cr=1 pr=0 pw=0 time=31 us)        |
| 2    | TABLE ACCESS BY INDEX ROWID EMP (cr=4 pr=0 pw=0 time=97 us)  |
| 2    | INDEX RANGE SCAN I_DEPTNO (cr=2 pr=0 pw=0 time=55 us)        |
| 3    | NESTED LOOPS (cr=4 pr=0 pw=0 time=115 us)                    |
| 1    | TABLE ACCESS BY INDEX ROWID DEPT (cr=2 pr=0 pw=0 time=41 us) |
| 1    | INDEX UNIQUE SCAN PK_DEPT (cr=1 pr=0 pw=0 time=18 us)        |
| 3    | TABLE ACCESS BY INDEX ROWID EMP (cr=2 pr=0 pw=0 time=49 us)  |
| 3    | INDEX RANGE SCAN I_DEPTNO (cr=1 pr=0 pw=0 time=23 us)        |

## ***UNION***

Operation accepting two sets of rows and returns the union of the sets, eliminating duplicates.

Union ALL followed by Sort-Unique

- Sub row sources are all executed/optimized individually
- Rows retrieved are 'concatenated'
- Set theory demands unique elements (Sort)



# Summary

- At the end of the part, you should have learned :
  - The meaning of usual operations
  - Be familiar with very simple execution plan

ORACLE

## ***SUMMARY***

In this lesson you should have learned most usual operations seen in Execution Plan.