

HINTS

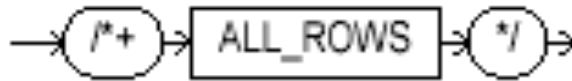
- Hints provide a mechanism to direct the optimizer to choose
 - a certain query execution plan.
- Hints (except for the RULE hint) invoke the cost-based optimizer(CBO).
- The syntax below shows hints contained in both styles of comments that Oracle supports within a statement block.
- ```
{DELETE | INSERT | SELECT | UPDATE} /*+ hint [text]
[hint[text]]... */
```
- where:
  - **DELETE, SELECT, UPDATE** : –Is a keyword that begins a statement block. Comments containing hints can appear only after these keywords.
  - **+ :-** Causes Oracle to interpret the comment as a list of hints. The plus sign must immediately follow the comment delimiter (no space is permitted).
  - **hint** : –Is one of the hints discussed in this section. If the comment contains multiple hints, then each hints must be separated by at least one space.
  - **text** : – Is other commenting text that can be interspersed with the hints.

# Using Optimizer Hints

- The hints described in this section allow you to choose between the cost-based and the rule-based optimization approaches. With the cost-based approach, this also includes the goal of best throughput or best response time.
  1. ALL\_ROWS
  2. FIRST\_ROWS(n)
  3. CHOOSE

# ALL\_ROWS

- The ALL\_ROWS hint explicitly chooses the cost-based approach to optimize a statement for best throughput ie. to minimize the time it takes to execute the query by the query
- all\_rows\_hint



- For example, the optimizer uses the cost-based approach to optimize this statement
- for best throughput:
- `SELECT /*+ ALL_ROWS */ empno, ename, sal, job`
- `FROM emp`
- `WHERE empno = 7566;`

# FIRST\_ROWS( n)

- `FIRST_ROWS(n)` affords greater precision, because it instructs Oracle to choose the plan that returns the first rows most efficiently.
- ```
SELECT /*+ FIRST_ROWS(10) */ empno, ename,  
sal, job
```
- ```
FROM emp
```
- ```
WHERE deptno = 20;
```
- In this example each department contains many employees. The user wants the first 10 employees of department #20 to be displayed as quickly as possible.

CHOOSE

- The **CHOOSE** hint causes the optimizer to choose between the rule-based and cost-based approaches for a SQL statement.
- The optimizer bases its selection on the presence of statistics for the tables accessed by the statement.
- If the tables used by the query have been analyzed, the optimizer will use cost-based optimization. If no statistics, rule-based optimization will be used.

- ```
SELECT /*+ CHOOSE */ empno, ename, sal, job FROM emp
```

- ```
WHERE empno = 7566;
```

- ```
SELECT /*+ CHOOSE */ empno, ename, sal, job
```

- ```
FROM emp
```

- ```
WHERE empno = 7566;
```

# Hints for Access Paths

- Each hint described in this section suggests an access path for a table.

## 1. FULL

  ROWID

  CLUSTER

  HASH

  INDEX

  INDEX\_ASC

  INDEX\_COMBINE

  INDEX\_JOIN

  INDEX\_DESC

  INDEX\_FFS

   NO\_INDEX

    AND\_EQUAL

# FULL

- The `FULL` hint explicitly chooses a full table scan for the specified table.
- For example:
  - `SELECT /*+ FULL(A) don't use the index on accno */ accno, bal`
  - `FROM accounts a`
  - `WHERE accno = 7086854;`
  - 
  - Oracle performs a full table scan on the `accounts` table to execute this statement, even if there is an index on the `accno` column that is made available by the condition in the `WHERE` clause.



# ROWID

- The ROWID hint explicitly chooses a table scan by rowid for the specified table.

rowid\_hint::=

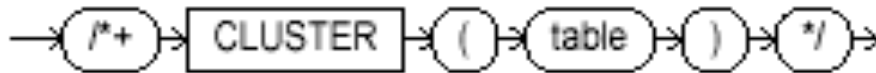


- where `table` specifies the name or alias of the table on which the table access by rowid is to be performed.
- For example:
- `SELECT /*+ROWID(emp)*/ *`
- `FROM emp`
- `WHERE rowid > 'AAAAtkAABAAAFNTAAA' AND empno = 155;`

# CLUSTER

- The **CLUSTER** hint explicitly chooses a cluster scan to access the specified table. It applies only to clustered objects.

cluster\_hint::=



where **table** specifies the name or alias of the table to be accessed by a cluster scan. For example:

```
SELECT --+ CLUSTER
```

```
emp.ename, deptno
```

```
FROM emp, dept
```

```
WHERE deptno = 10
```

```
AND emp.deptno = dept.deptno;
```

# HASH

- The `HASH` hint explicitly chooses a hash scan to access the specified table. It applies only to tables stored in a cluster.

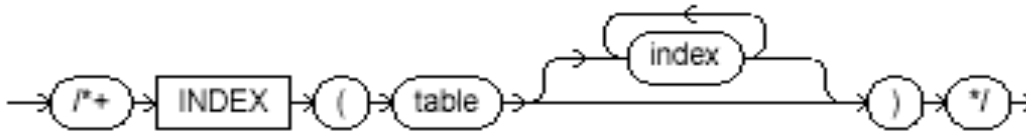
`hash_hint:=`



where `table` specifies the name or alias of the table to be accessed by a hash scan.

# INDEX

- The `INDEX` hint explicitly chooses an index scan for the specified table. You can use the `INDEX` hint for domain, B-tree, bitmap, and bitmap join indexes.
- `index_hint ::=`



where:

**Table**– Specifies the name or alias of the table associated with the index to be scanned.

**Index**– Specifies an index on which an index scan is to be performed.

- If this hint specifies a single available index, then the optimizer performs a scan on this index. The optimizer does not consider a full table scan or a scan on another index on the table.
- If this hint specifies a list of available indexes, then the optimizer considers the cost of a scan on each index in the list and then performs the index scan with the lowest cost.

# INDEX\_ASC

- The `INDEX_ASC` hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, then Oracle scans the index entries in ascending order of their indexed values.
- Because Oracle's default behavior for a range scan is to scan index entries in ascending order of their indexed values,
- However, you might want to use the `INDEX_ASC` hint to specify ascending range scans explicitly should the default behavior change.

# INDEX\_COMBINE

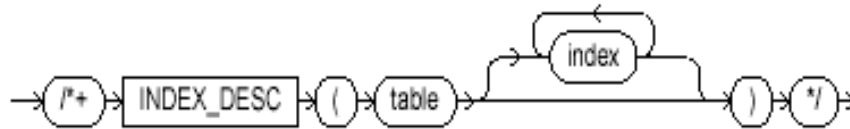
- The `INDEX_COMBINE` hint explicitly chooses a bitmap access path for the table.
- If no indexes are given as arguments for the `INDEX_COMBINE` hint, then the optimizer uses whatever Boolean combination of bitmap indexes has the best cost estimate for the table.
- If certain indexes are given as arguments, then the optimizer tries to use some Boolean combination of those particular bitmap indexes.
- For example:
- ```
SELECT /*+INDEX_COMBINE(emp sal_bmi  
hiredate_bmi)*/ *
```
- ```
FROM emp
```
- ```
WHERE sal < 50000 AND hiredate < '01-JAN-  
1990';
```

INDEX_JOIN

- The `INDEX_JOIN` hint explicitly instructs the optimizer to use an index join as an access path.
- For the hint to have a positive effect, a sufficiently small number of indexes must exist that contain all the columns required to resolve the query.
- For example:
 - `SELECT /*+INDEX_JOIN(emp sal_bmi hiredate_bmi)*/ sal, hiredate`
 - `FROM emp`
 - `WHERE sal < 50000;`

INDEX_DESC

- The `INDEX_DESC` hint explicitly chooses an index scan for the specified table.
- If the statement uses an index range scan, then Oracle scans the index entries in descending order of their indexed values.
- In a partitioned index, the results are in descending order within each partition.



Each parameter serves the same purpose as in the `INDEX` hint.

INDEX_FFS

- The `INDEX_FFS` hint causes a fast full index scan to be performed rather than a full table scan.

index_ffs_hint:=



For example:

```
SELECT /*+INDEX_FFS(emp emp_empno)*/  
empno
```

```
FROM emp
```

```
WHERE empno > 200;
```

NO_INDEX

- The `NO_INDEX` hint explicitly disallows a set of indexes for the specified table.

`no_index_hint::=`



- If this hint specifies a single available index, then the optimizer does not consider a scan on this index. Other indexes not specified are still considered.
- If this hint specifies a list of available indexes, then the optimizer does not consider a scan on any of the specified indexes. Other indexes not specified in the list are still considered.
- If this hint specifies no indexes, then the optimizer does not consider a scan on any index on the table. This behavior is the same as a `NO_INDEX` hint that specifies a list of all available indexes for the table.

- The `NO_INDEX` hint applies to function-based, B-tree, bitmap, cluster, or domain indexes. If a `NO_INDEX` hint and an index hint (`INDEX`, `INDEX_ASC`, `INDEX_DESC`, `INDEX_COMBINE`, or `INDEX_FFS`) both specify the same indexes, then both the `NO_INDEX` hint and the index hint are ignored for the specified indexes and the optimizer considers the specified indexes.
- For example:
 - `SELECT /*+NO_INDEX(emp emp_empno)*/ empno`
 - `FROM emp`
 - `WHERE empno > 200;`

AND_EQUAL

- The `AND_EQUAL` hint explicitly chooses an execution plan that uses an access path that merges the scans on several single-column indexes.
- `and_equal_hint::=`

`and_equal_hint::=`

where:



Table– Specifies the name or alias of the table associated with the indexes to be merged.

index –Specifies an index on which an index scan is to be performed. You must specify at least two indexes. You cannot specify more than five.

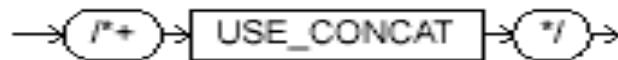
Hints for Query Transformations

- Each hint described in this section suggests a SQL query transformation.
 1. USE_CONCAT
 2. NO_EXPAND
 3. REWRITE
 4. NOREWRITE
 5. MERGE
 6. NO_MERGE
 7. STAR_TRANSFORMATION
 8. FACT
 9. NO_FACT

USE_CONCAT

- **CONCAT** hint forces combined OR conditions in the WHERE clause of a query to be transformed into a compound query using the UNION ALL set operator.
- Generally, this transformation occurs only if the cost of the query using the concatenations is cheaper than the cost without them.
- The **USE_CONCAT** hint turns off IN-list processing and OR-expands all disjunctions, including IN-lists.

use_concat_hint:=



For example:

```
SELECT /*+USE_CONCAT*/ *
```

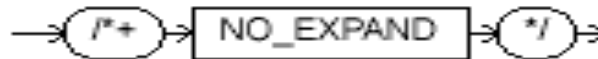
```
FROM emp
```

```
WHERE empno > 50 OR sal < 50000;
```

NO_EXPAND

- The `NO_EXPAND` hint prevents the cost-based optimizer from considering OR-expansion for queries having OR conditions or IN-lists in the WHERE clause.
- Usually, the optimizer considers using OR expansion and uses this method if it decides that the cost is lower than not using it.

no_expand_hint::=



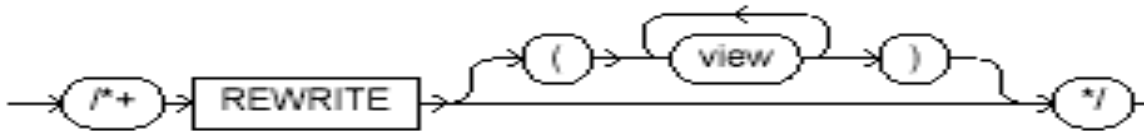
For example:

```
SELECT /*+NO_EXPAND*/ *  
FROM emp  
WHERE empno = 50 OR empno = 100;
```


REWRITE

- The `REWRITE` hint forces the cost-based optimizer to rewrite a query in terms of materialized views, when possible, without cost consideration.
- Use the `REWRITE` hint with or without a view list. If you use `REWRITE` with a view list and the list contains an eligible materialized view, then Oracle uses that view regardless of its cost.
- Oracle does not consider views outside of the list. If you do not specify a view list, then Oracle searches for an eligible materialized view and always uses it regardless of its cost.

`rewrite_hint:=`



NOREWRITE

- The `NOREWRITE` hint disables query rewrite for the query block, overriding the setting of the parameter `QUERY_REWRITE_ENABLED`. Use the `NOREWRITE` hint on any query block of a request.

norewrite_hint::=



MERGE

- The **MERGE** hint lets you merge a view on a per-query basis.
- If a view's query contains a **GROUP BY** clause or **DISTINCT** operator in the list, then the optimizer can merge the view's query into the accessing statement only if complex view merging is enabled.
- Complex merging can also be used to merge an **IN** subquery into the accessing statement if the subquery is uncorrelated.
- Complex merging is not cost-based--that is, the accessing query block must include the **MERGE** hint. Without this hint, the optimizer uses another approach.
- For example:
 - `SELECT /*+MERGE(v)*/ e1.ename, e1.sal, v.avg_sal`
 - `FROM emp e1,`
 - `(SELECT deptno, avg(sal) avg_sal`
 - `FROM emp e2`
 - `GROUP BY deptno) v`
 - `WHERE e1.deptno = v.deptno AND e1.sal > v.avg_sal;`

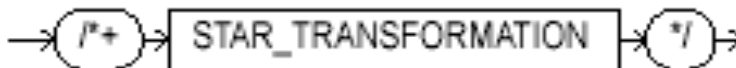
NO_MERGE

- The `NO_MERGE` hint causes Oracle not to merge mergeable views. This hint lets the user have more influence over the way in which the view is accessed.
- For example:
- ```
SELECT /*+NO_MERGE(dallasdept)*/ e1.ename,
dallasdept.dname
```
- ```
FROM emp e1,
```
- ```
(SELECT deptno, dname
```
- ```
FROM dept
```
- ```
WHERE loc = 'DALLAS') dallasdept
```
- ```
WHERE e1.deptno = dallasdept.deptno;
```
- This causes view `dallasdept` not to be merged.
- When the `NO_MERGE` hint is used without an argument, it should be placed in the view query block. When `NO_MERGE` is used with the view name as an argument, it should be placed in the surrounding query.

STAR_TRANSFORMATION

- The `STAR_TRANSFORMATION` hint makes the optimizer use the best plan in which the transformation has been used.
- Without the hint, the optimizer could make a cost-based decision to use the best plan generated without the transformation, instead of the best plan for the transformed query.
- Even if the hint is given, there is no guarantee that the transformation will take place. The optimizer only generates the subqueries if it seems reasonable to do so.
- If no subqueries are generated, then there is no transformed query, and the best plan for the untransformed query is used, regardless of the hint.

`star_transformation_hint:=`



FACT

- The **FACT** hint is used in the context of the star transformation to indicate to the transformation that the hinted table should be considered as a fact table.

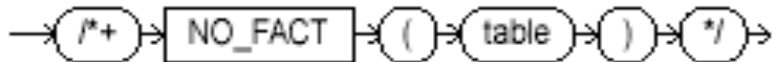
fact_hint ::=



NO_FACT

The **NO_FACT** hint is used in the context of the star transformation to indicate to the transformation that the hinted table should not be considered as a fact table.

no_fact_hint ::=



Hints for Join Orders

- The hints in this section suggest join orders:

1. ORDERED

-  STAR

- ORDERED

- The ORDERED hint causes Oracle to join tables in the order in which they appear in the FROM clause.

- If you omit the ORDERED hint from a SQL statement performing a join, then the optimizer chooses the order in which to join the tables. You might want to use the ORDERED hint to specify a join order if you know something about the number of rows selected from each table that the optimizer does not.

- ```
SELECT /*+ ORDERED */ tab1.col1, tab2.col2,
tab3.col3
```

- ```
FROM tab1, tab2, tab3
```

- ```
WHERE tab1.col1 = tab2.col1
```

- ```
AND tab2.col1 = tab3.col1;
```

STAR

- The `STAR` hint forces a star query plan to be used, if possible.
- A star plan has the largest table in the query last in the join order and joins it with a nested loops join on a concatenated index.
- The `STAR` hint applies when there are at least three tables, the large table's concatenated index has at least three columns, and there are no conflicting access or join method hints.
- The optimizer also considers different permutations of the small tables.
- Usually, if you analyze the tables, then the optimizer selects an efficient star plan.
- You can also use hints to improve the plan. The most precise method is to order the tables in the `FROM` clause in the order of the keys in the index, with the large table last. Then use the following hints:
 - `/*+ ORDERED USE_NL(FACTS) INDEX(facts fact_concat) */`
 - where `facts` is the table and `fact_concat` is the index. A more general method is to use the `STAR` hint.

Hints for Join Operations

- Each hint described in this section suggests a join operation for a table.

  USE_NL

  USE_MERGE

3. USE_HASH

4. DRIVING_SITE

  LEADING

  HASH_AJ, MERGE_AJ, and NL_AJ

7. HASH_SJ, MERGE_SJ, and NL_SJ

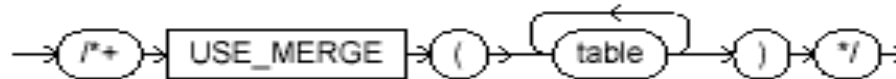
USE_NL

- The `USE_NL` hint causes Oracle to join each specified table to another row source with a nested loops join using the specified table as the inner table.
- `SELECT /*+ ORDERED USE_NL(customers) to get first row faster */`
- `accounts.balance, customers.last_name, customers.first_name`
- `FROM accounts, customers`
- `WHERE accounts.custno = customers.custno;`
- In many cases, a nested loops join returns the first row faster than a sort merge join.
- A nested loops join can return the first row after reading the first selected row from one table and the first matching row from the other and combining them, while a sort merge join cannot return the first row until after reading and sorting all selected rows of both tables and then combining the first rows of each sorted row source.

USE_MERGE

- The `USE_MERGE` hint causes Oracle to join each specified table with another row source with a sort-merge join.

`use_merge_hint::=`



where `table` is a table to be joined to the row source resulting from joining the previous tables in the join order using a sort merge join.

For example:

```
SELECT /*+USE_MERGE(emp dept)*/ *  
FROM emp, dept  
WHERE emp.deptno = dept.deptno;
```

USE_HASH

- The `USE_HASH` hint causes Oracle to join each specified table with another row source with a hash join.

`use_hash_hint::=`



where `table` is a table to be joined to the row source resulting from joining the previous tables in the join order using a hash join.

For example:

```
SELECT /*+use_hash(emp dept)*/ *
```

```
FROM emp, dept
```

```
WHERE emp.deptno = dept.deptno;
```

DRIVING_SITE

- The `DRIVING_SITE` hint forces query execution to be done at a different site than that selected by Oracle. This hint can be used with either rule-based or cost-based optimization.

For example:

```
SELECT /*+DRIVING_SITE(dept)*/ *  
FROM emp, dept@rsite  
WHERE emp.deptno = dept.deptno;
```

- If this query is executed without the hint, then rows from `dept` are sent to the localsite, and the join is executed there. With the hint, the rows from `emp` are sent to the remote site, and the query is executed there, returning the result to the local site.
- This hint is useful if you are using distributed query optimization.

LEADING

- The **LEADING** hint causes Oracle to use the specified table as the first table in the join order.
- If you specify two or more **LEADING** hints on different tables, then all of them are ignored. If you specify the **ORDERED** hint, then it overrides all **LEADING** hints.

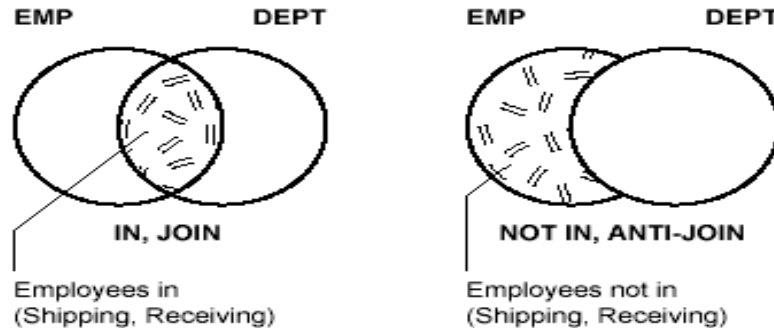
leading_hint::=



where *table* is the name or alias of a table to be used as the first table in the join order.

HASH_AJ, MERGE_AJ, and NL_AJ

- For a specific query, place the MERGE_AJ, HASH_AJ, or NL_AJ hint into the NOT IN subquery. MERGE_AJ uses a sort-merge anti-join, HASH_AJ uses a hash anti-join, and NL_AJ uses a nested loop anti-join. In the figure, the SQL IN predicate can be evaluated using a join to intersect two sets. Thus, emp.deptno can be joined to dept.deptno to yield a list of employees in a set of departments.



- Alternatively, the SQL NOT IN predicate can be evaluated using an anti-join to subtract two sets. Thus, emp.deptno can be anti-joined to dept.deptno to select all employees who are not in a set of departments, and you can get a list of all employees who are not in the shipping or receiving departments.

HASH_SJ, MERGE_SJ, and NL_SJ

- For a specific query, place the HASH_SJ, MERGE_SJ, or NL_SJ hint into the EXISTS subquery. HASH_SJ uses a hash semi-join, MERGE_SJ uses a sort merge semi-join, and NL_SJ uses a nested loop semi-join.
- For example:
 - `SELECT * FROM dept`
 - `WHERE exists (SELECT /*+HASH_SJ*/ *`
 - `FROM emp`
 - `WHERE emp.deptno = dept.deptno`
 - `AND sal > 200000);`
- This converts the subquery into a special type of join between t1 and t2 that preserves the semantics of the subquery. That is, even if there is more than one matching row in t2 for a row in t1, the row in t1 is returned only once.
- A subquery is evaluated as a semi-join only with these limitations:
 1. There can only be one table in the subquery.
 2. The outer query block must not itself be a subquery.
 3. The subquery must be correlated with an equality predicate.
 4. The subquery must have no GROUP BY, CONNECT BY, or ROWNUM references.

Hints for Parallel Execution

- The hints described in this section determine how statements are parallelized or not parallelized when using parallel execution.
 1. PARALLEL
 2. NOPARALLEL
 3. PQ_DISTRIBUTE
 4. PARALLEL_INDEX
 5. NOPARALLEL_INDEX

PARALLEL

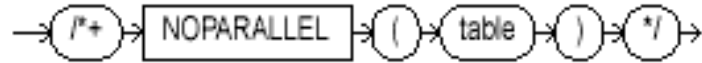
- The `PARALLEL` hint lets you specify the desired number of concurrent servers that can be used for a parallel operation.
- The hint applies to the `INSERT`, `UPDATE`, and `DELETE` portions of a statement as well as to the table scan portion.
- If any parallel restrictions are violated, then the hint is ignored.
- In the following example, the `PARALLEL` hint overrides the degree of parallelism specified in the `emp` table definition:

-
- ```
SELECT /*+ FULL(scott_emp)
PARALLEL(scott_emp, 5) */ ename
```
- ```
FROM scott.emp scott_emp;
```

NOPARALLEL

- The NOPARALLEL hint overrides a PARALLEL specification in the table clause. In general, hints take precedence over table clauses.

noparallel_hint:=



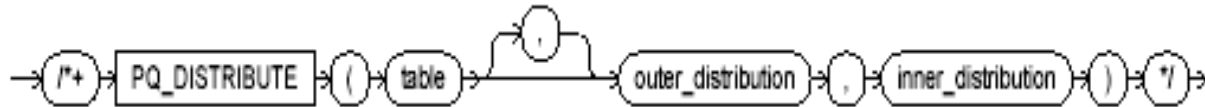
The following example illustrates the NOPARALLEL hint:

```
SELECT /*+ NOPARALLEL(scott_emp) */ ename  
FROM scott.emp scott_emp;
```

PQ_DISTRIBUTE

- The `PQ_DISTRIBUTE` hint improves parallel join operation performance. Do this by specifying how rows of joined tables should be distributed among producer and consumer query servers.
- Using this hint overrides decisions the optimizer would normally make.

`pq_distribute_hint:=`



where:

`table_name`– Name or alias of a table to be used as the inner table of a join.

`outer_distribution` –The distribution for the outer table.

`inner_distribution`– The distribution for the inner table.

- For example: Given two tables, R and S, that are joined using a hash-join, the
- following query contains a hint to use hash distribution:
 - `SELECT <column_list> /*+ORDERED`
 - `PQ_DISTRIBUTE(s HASH, HASH) USE_HASH (s)*/`
 - `FROM r,s`
 - `WHERE r.c=s.c;`
- To broadcast the outer table r, the query is:
 - `SELECT <column list> /*+ORDERED`
 - `PQ_DISTRIBUTE(s BROADCAST, NONE) USE_HASH (s)`
 - `*/`
 - `FROM r,s`
 - `WHERE r.c=s.c;`

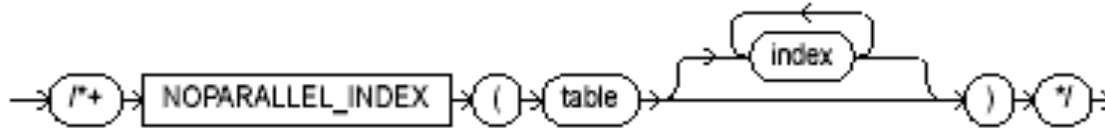
PARALLEL_INDEX

- The `PARALLEL_INDEX` hint specifies the desired number of concurrent servers that can be used to parallelize index range scans for partitioned indexes.
- The hint can take two values separated by commas after the table name.
- The first value specifies the degree of parallelism for the given table. The second value specifies how the table is to be split among the Oracle Real Application Cluster instances.
- For example:
 - `SELECT /*+ PARALLEL_INDEX(table1, index1, 3, 2) */`
 - In this example, there are three parallel execution processes to be used on each of two instances.


NOPARALLEL_INDEX

- The `NOPARALLEL_INDEX` hint overrides a `PARALLEL` attribute setting on an index to avoid a parallel index scan operation.

`noparallel_index_hint::=`



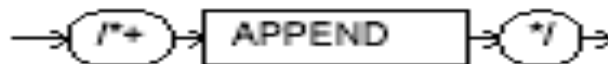
Additional Hints

- Several additional hints are included in this section:
 1. APPEND
 2. NOAPPEND
 3. CACHE
 4. NOCACHE
 5. UNNEST
 6. NO_UNNEST
 7. PUSH_PRED
 - ✗  NO_PUSH_PRED
 9. PUSH_SUBQ

APPEND

- The `APPEND` hint lets you enable direct-path `INSERT` if your database is running in serial mode.
- In direct-path `INSERT`, data is appended to the end of the table, rather than using existing space currently allocated to the table. In addition, direct-path `INSERT` bypasses the buffer cache and ignores integrity constraints.
- As a result, direct-path `INSERT` can be considerably faster than conventional `INSERT`.

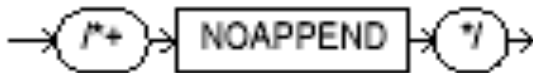
`append_hint::=`



NOAPPEND

- The NOAPPEND hint enables conventional INSERT by disabling parallel mode for the duration of the INSERT statement. (Conventional INSERT is the default in serial mode, and direct-path INSERT is the default in parallel mode).

noappend_hint::=



CACHE

- The `CACHE` hint specifies that the blocks retrieved for the table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed.
- This option is useful for small lookup tables.

cache_hint:=



In the following example, the `CACHE` hint overrides the table's default caching specification:

```
SELECT /*+ FULL (scott_emp) CACHE(scott_emp) */ ename
FROM scott.emp scott_emp;
```

NOCACHE

- The `NOCACHE` hint specifies that the blocks retrieved for the table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed.
- This is the normal behavior of blocks in the buffer cache.

`nocache_hint::=`



For example:

```
SELECT /*+ FULL(scott_emp) NOCACHE(scott_emp) */  
ename  
  
FROM scott.emp scott_emp;
```

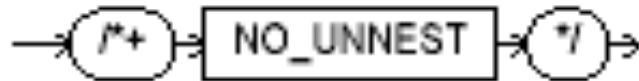
UNNEST

- Setting the `UNNEST_SUBQUERY` session parameter to `TRUE` enables subquery unnesting.
- Subquery unnesting unnests and merges the body of the subquery into the body of the statement that contains it, allowing the optimizer to consider them together when evaluating access paths and joins.
- `UNNEST_SUBQUERY` first verifies if the statement is valid. If the statement is not valid, then subquery unnesting cannot proceed. The statement must then must pass a heuristic test.
- If the `UNNEST_SUBQUERY` parameter is set to true, then the `UNNEST` hint checks the subquery block for validity only. If it is valid, then subquery unnesting is enabled without Oracle checking the heuristics.

NO_UNNEST

- If you enabled subquery unnesting with the UNNEST_SUBQUERY parameter, then the NO_UNNEST hint turns it off for specific subquery blocks.

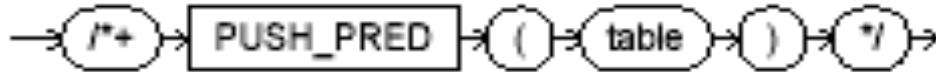
no_unnest_hint:=



PUSH_PRED

- The PUSH_PRED hint forces pushing of a join predicate into the view.

push_pred_hint::=

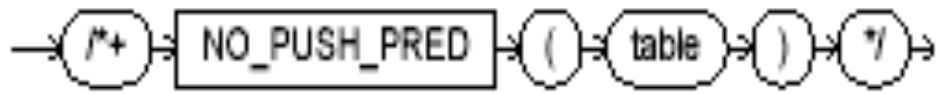


For example:

```
SELECT /*+ PUSH_PRED(v) */ t1.x, v.y
FROM t1
(SELECT t2.x, t3.y
FROM t2, t3
WHERE t2.x = t3.x) v
WHERE t1.x = v.x and t1.y = 1;
```

NO_PUSH_PRED

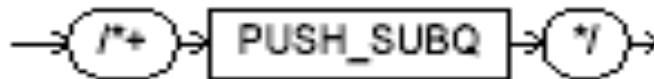
- The NO PUSH PRED hint prevents pushing of a join predicate into the `no_push_pred_hint::=`



PUSH_SUBQ

- The `PUSH_SUBQ` hint causes non-merged subqueries to be evaluated at the earliest possible place in the execution plan.
- Generally, subqueries that are not merged are executed as the last step in the execution plan.
- If the subquery is relatively inexpensive and reduces the number of rows significantly, then it improves performance to evaluate the subquery earlier.
- This hint has no effect if the subquery is applied to a remote table or one that is joined using a merge join.

`push_subq_hint::=`



Using Hints with Views

1. Hints and Nonmergeable Views

2. Hints and Mergeable Views

- Hints and Mergeable Views
- Optimization approach and goal hints can occur in a top-level query or inside views.
- If there is such a hint in the top-level query, then that hint is used regardless of any such hints inside the views.
- If there is no top-level optimizer mode hint, then mode hints in referenced views are used as long as all mode hints in the views are consistent.
- If two or more mode hints in the referenced views conflict, then all mode hints in the views are discarded and the session mode is used, whether default or user-specified.

Hints and Nonmergeable Views

- With nonmergeable views, optimization approach and goal hints inside the view are ignored: the top-level query decides the optimization mode.
- Because nonmergeable views are optimized separately from the top-level query, access path and join hints inside the view are preserved. For the same reason, access path hints on the view in the top-level query are ignored.
- However, join hints on the view in the top-level query are preserved because, in this case, a nonmergeable view is similar to a table.

Global Hints

- If you want to specify a hint for a table in a view or subquery, then the global hint syntax is recommended. You can transform any table hint into a global hint by using an extended syntax for the table name as follows
- Consider the following view definitions and `SELECT` statement:
 - `CREATE VIEW v1 AS`
 - `SELECT *`
 - `FROM emp`
 - `WHERE empno < 100;`
 - `CREATE VIEW v2 AS`
 - `SELECT v1.empno empno, dept.deptno deptno`
 - `FROM v1, dept`
 - `WHERE v1.deptno = dept.deptno;`

- `SELECT /*+ INDEX(v2.v1.emp emp_empno)
FULL(v2.dept) */ *`
- `FROM v2`
- `WHERE deptno = 20;`
- The view V1 retrieves all employees whose employee number is less than 100. The view V2 performs a join between the view V1 and the department table. The `SELECT` statement retrieves rows from the view V2 restricting it to the department whose number is 20.
- There are two global hints in the `SELECT` statement. The first hint specifies an index scan for the employee table referenced in the view V1, which is referenced in the view V2.
- The second hint specifies a full table scan for the department table
- referenced in the view V2.

- A hint such as:
 - `INDEX(emp emp_empno)`
 - in the `SELECT` statement is ignored because the employee table does not appear in the `FROM` clause of the `SELECT` statement.
- The global hint syntax also applies to unmergeable views. Consider the following
 - `SELECT` statement:
 - `SELECT /*+ NO_MERGE(v2) INDEX(v2.v1.emp emp_empno) FULL(v2.dept) */ *`
 - `FROM v2`
 - `WHERE deptno = 20;`
- It causes `V2` not to be merged and specifies access path hints for the employee and department tables. These hints are pushed down into the (nonmerged) view `V2`.

- If a global hint references a UNION or UNION ALL view, then the hint is applied to the first branch that contains the hinted table.
- Consider the INDEX hint in the following SELECT statement:
 - SELECT /*+ INDEX(v.emp emp_empno) */ *
 - FROM (SELECT *FROM emp
 - WHERE empno < 50
 - UNION ALL
 - SELECT *
 - FROM emp
 - WHERE empno > 1000) v
 - WHERE deptno = 20;
 - The INDEX hint applies to the employee table in the first branch of the UNION ALL
 - view v, not to the employee table in the second branch.