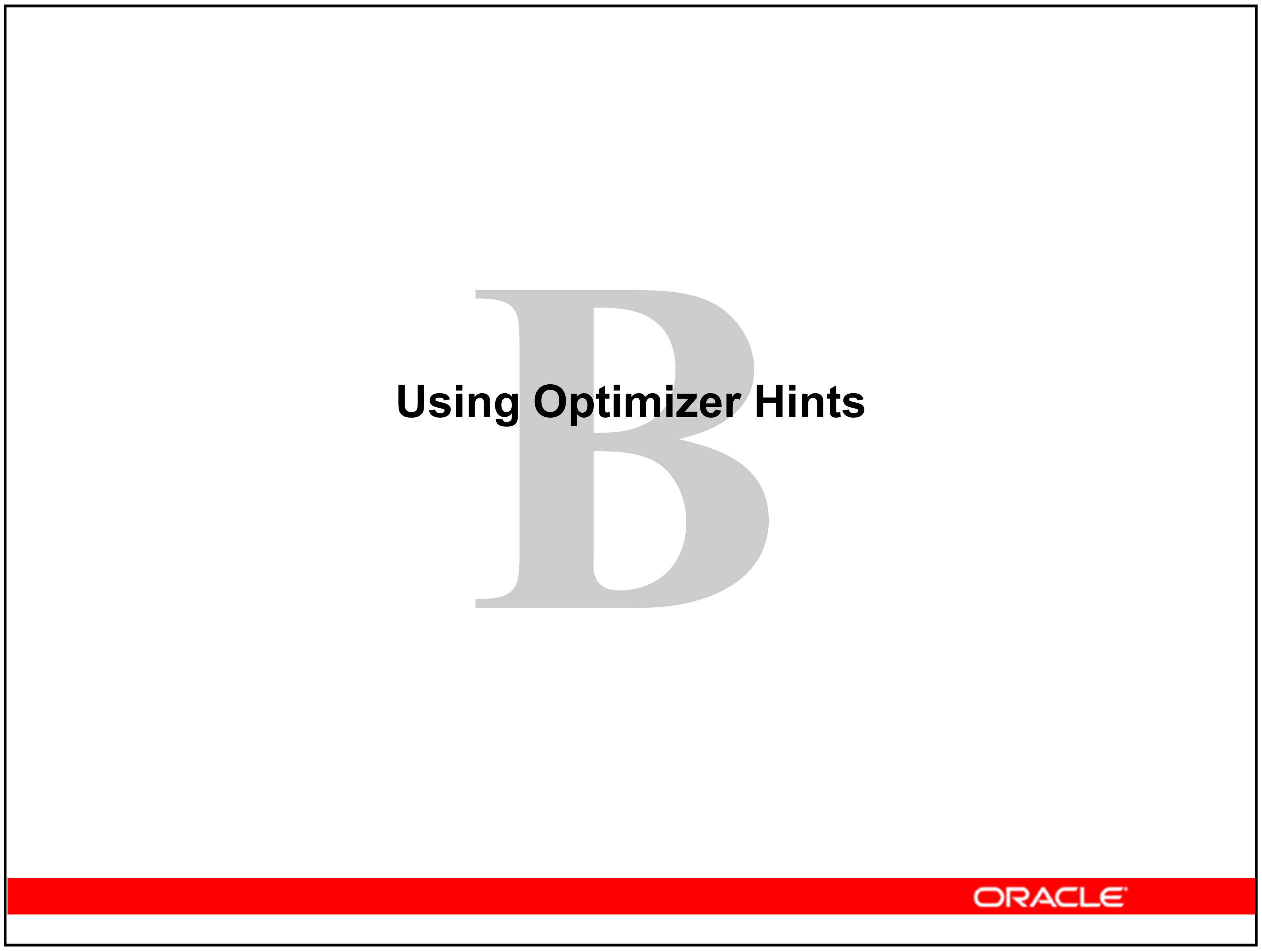
Using Optimizer Hints

Chapter 16

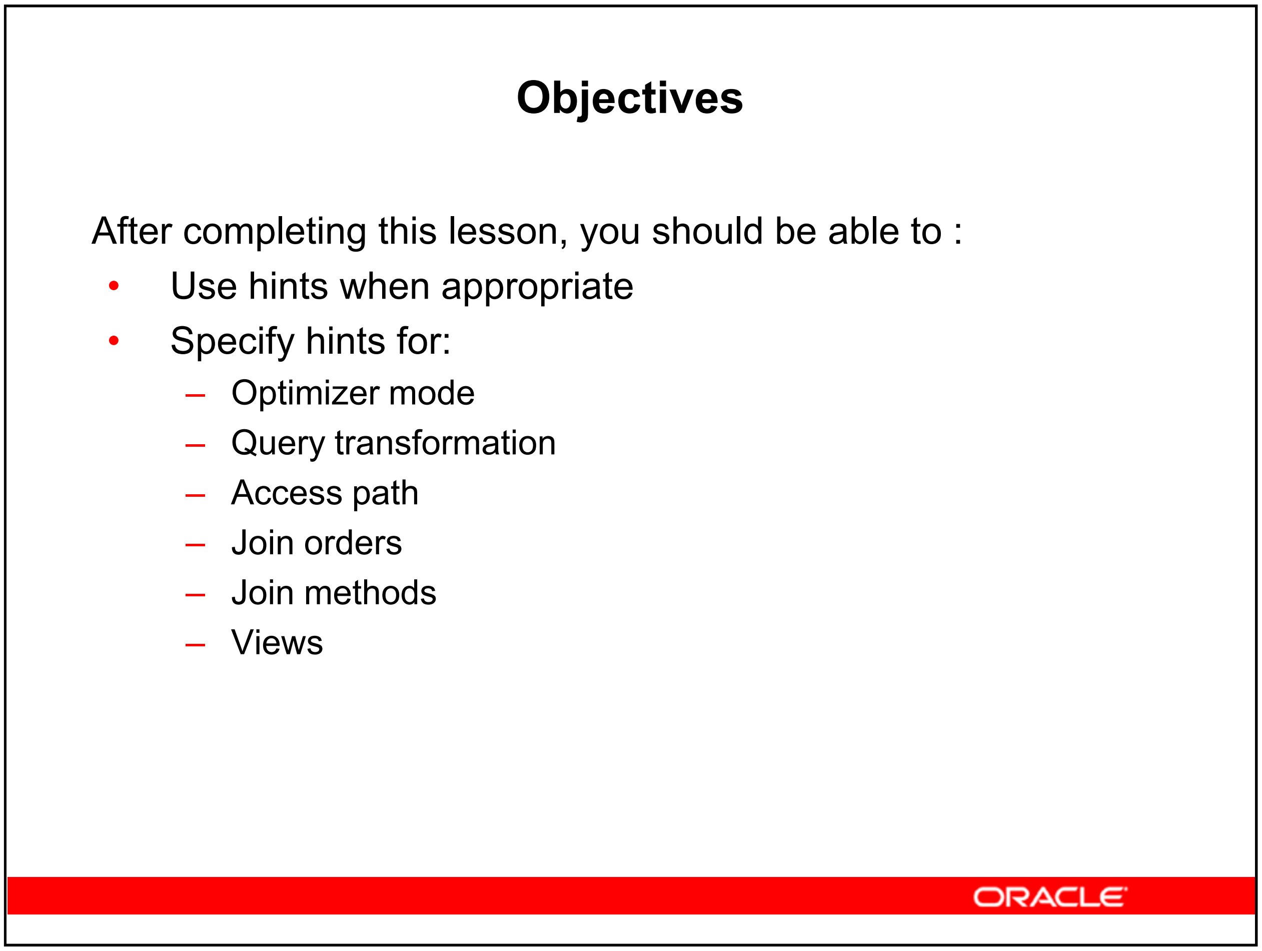


B

Using Optimizer Hints

ORACLE

Objectives

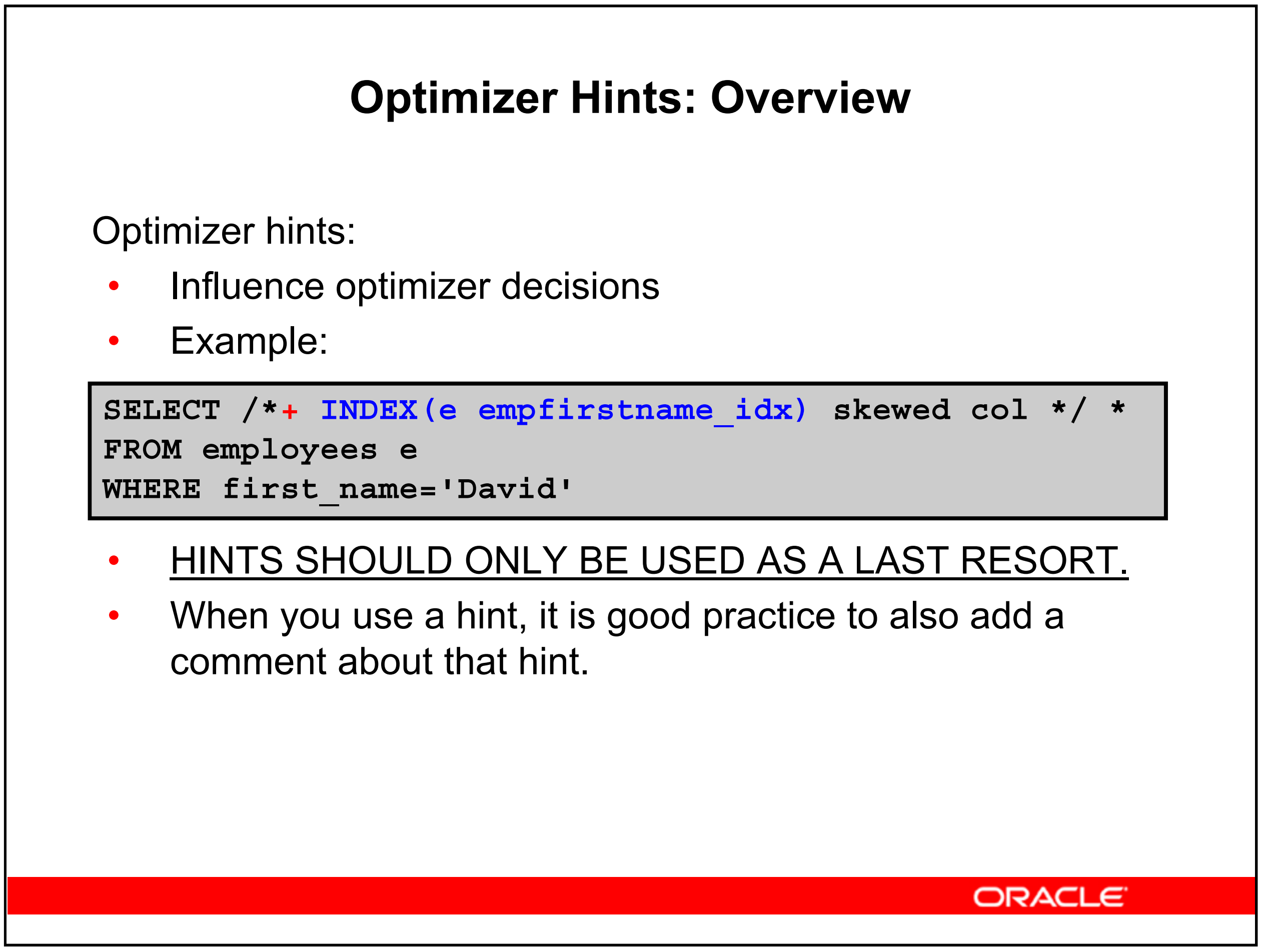


Objectives

After completing this lesson, you should be able to :

- Use hints when appropriate
- Specify hints for:
  - Optimizer mode
  - Query transformation
  - Access path
  - Join orders
  - Join methods
  - Views

Optimizer Hints: Overview



Optimizer Hints: Overview

Optimizer hints:

- Influence optimizer decisions
- Example:

```
SELECT /*+ INDEX(e empfirstname_idx) skewed col */ *
FROM employees e
WHERE first_name='David'
```

- <u>HINTS SHOULD ONLY BE USED AS A LAST RESORT.</u>
- When you use a hint, it is good practice to also add a comment about that hint.

ORACLE

Optimizer Hints: Overview

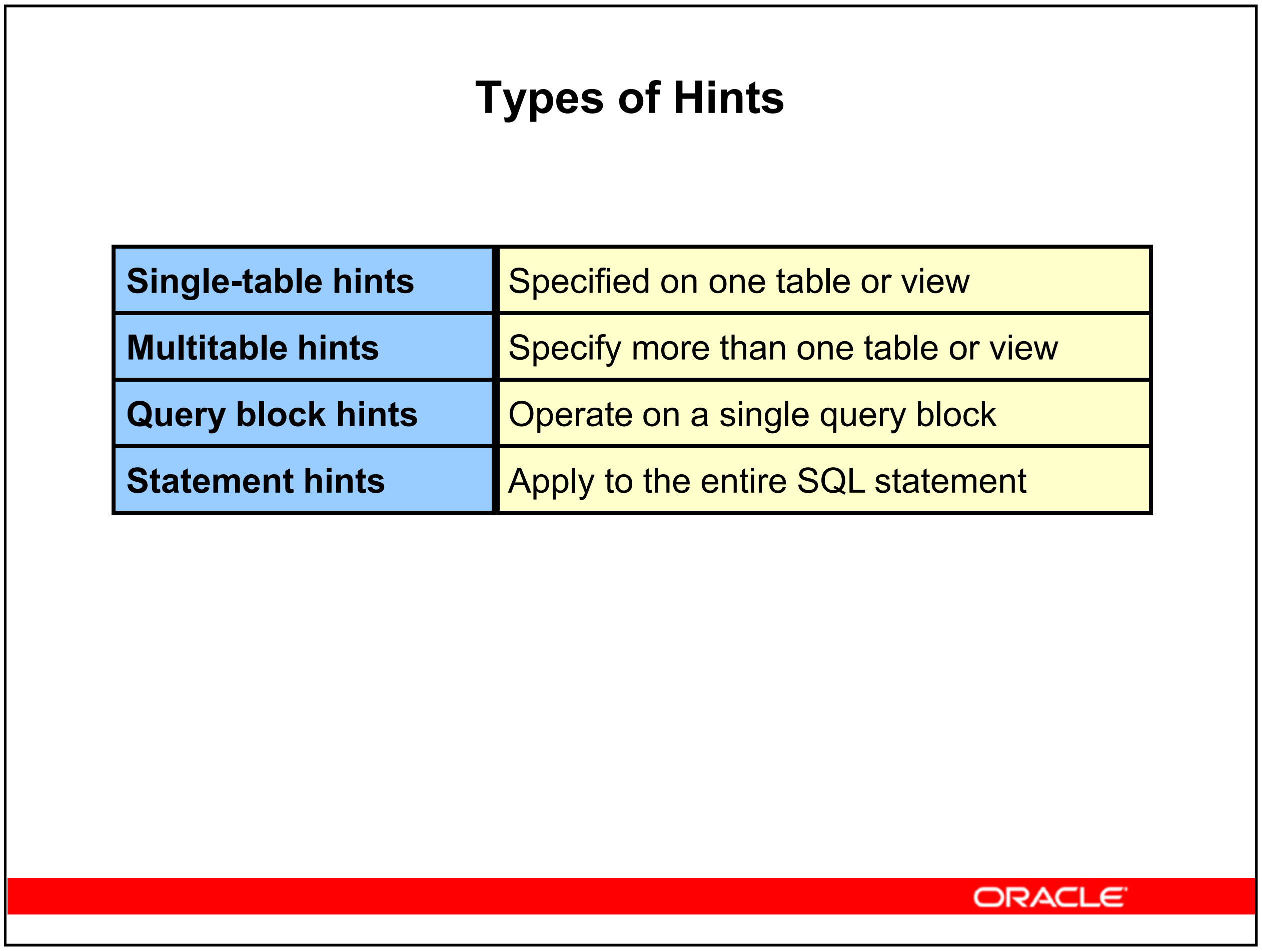
Hints enable you to influence decisions made by the optimizer. Hints provide a mechanism to direct the optimizer to select a certain query execution plan based on the specific criteria.

For example, you might know that a certain index is more selective for certain queries. Based on this information, you might be able to select a more efficient execution plan than the plan that the optimizer recommends. In such a case, use hints to force the optimizer to use the optimal execution plan. This is illustrated in the slide example where you force the optimizer to use the `EMPFIRSTNAME_IDX` index to retrieve the data. As you can see, you can use comments in a SQL statement to pass instructions to the optimizer.

The plus sign (+) causes the system to interpret the comment as a list of hints. The plus sign must follow immediately after the comment delimiter. No space is permitted.

Hints should be used sparingly, and only after you have collected statistics on the relevant tables and evaluated the optimizer plan without hints using the `EXPLAIN PLAN` statement. Changing database conditions as well as query performance enhancements in subsequent releases can have a significant impact on how hints in your code affect performance.

In addition, the use of hints involves extra code that must be managed, checked, and controlled.



Types of Hints

Single-table hints	Specified on one table or view
Multitable hints	Specify more than one table or view
Query block hints	Operate on a single query block
Statement hints	Apply to the entire SQL statement

Types of Hints

Single-table: Single-table hints are specified on one table or view. `INDEX` and `USE_NL` are examples of single-table hints.

Multitable: Multitable hints are like single-table hints, except that the hint can specify one or more tables or views. `LEADING` is an example of a multitable hint.

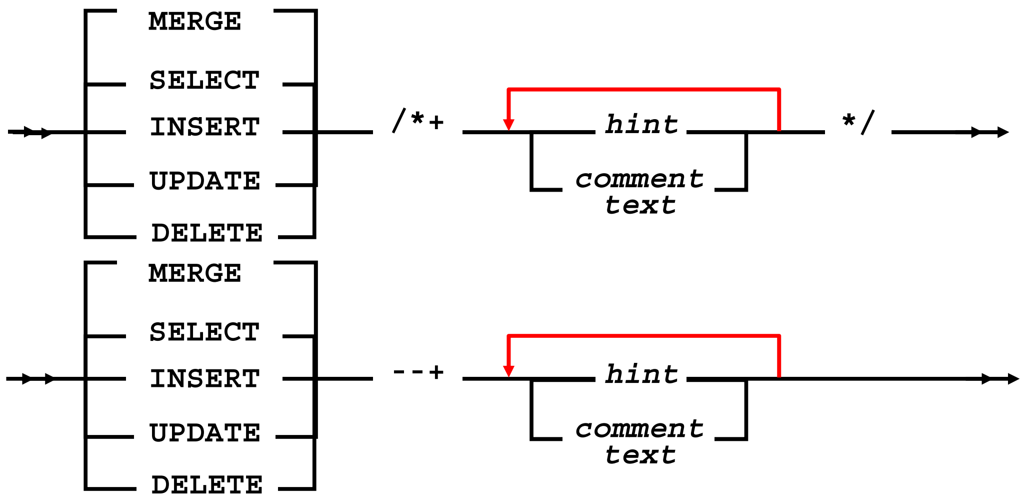
Query block: Query block hints operate on single query blocks. `STAR_TRANSFORMATION` and `UNNEST` are examples of query block hints.

Statement: Statement hints apply to the entire SQL statement. `ALL_ROWS` is an example of a statement hint.

Note: `USE_NL(table1 table2)` is not considered a multitable hint because it is actually a shortcut for `USE_NL(table1)` and `USE_NL(table2)`.

Specifying Hints

Hints apply to the optimization of only one statement block:
- A self-contained DML statement against a table
- A top-level DML or a subquery



ORACLE

Specifying Hints

Hints apply to the optimization of only the block of the statement in which they appear. A statement block is:

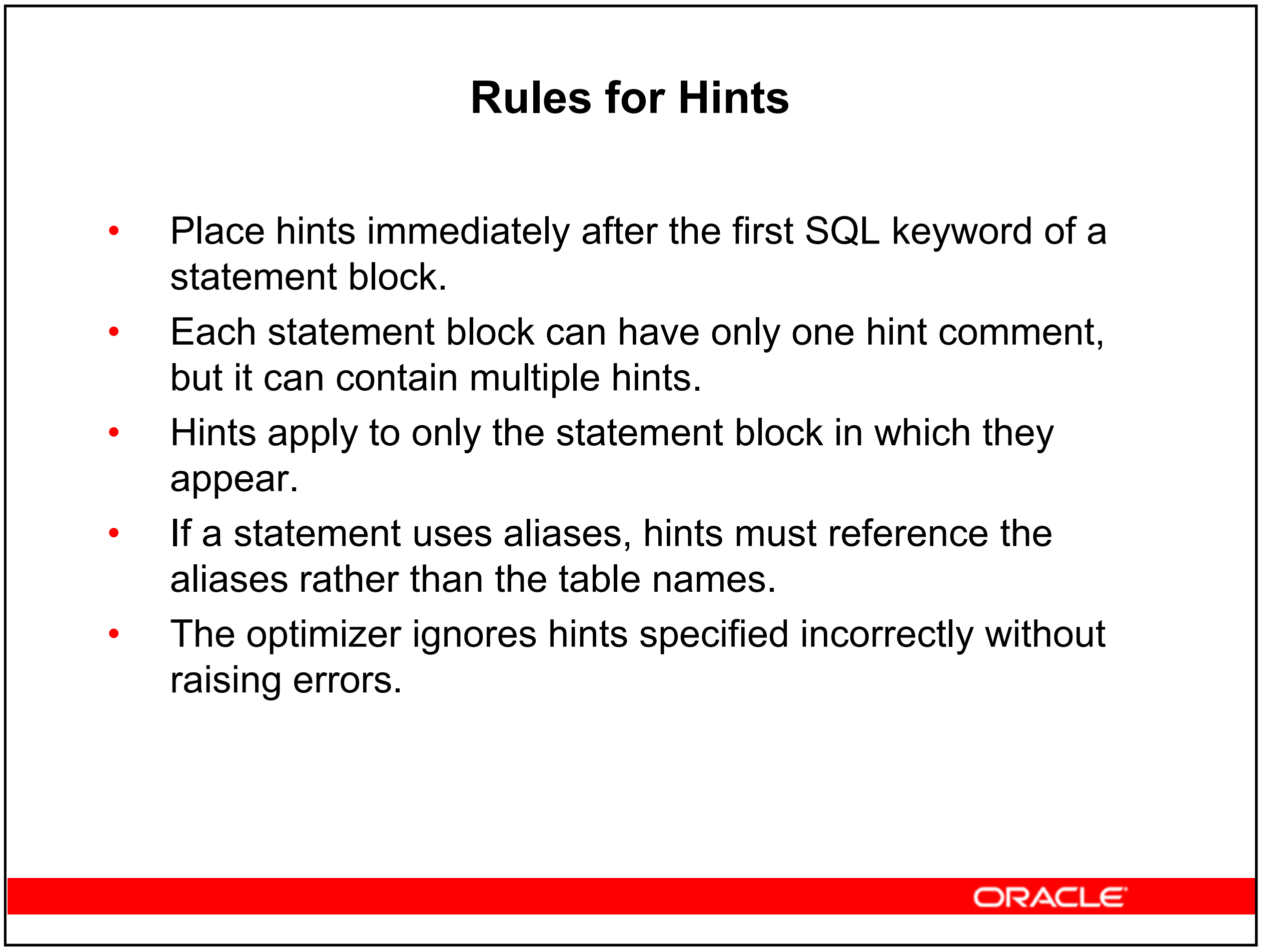
- A simple `MERGE`, `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement
- A parent statement or a subquery of a complex statement
- A part of a compound query using set operators (`UNION`, `MINUS`, `INTERSECT`)

For example, a compound query consisting of two component queries combined by the `UNION` operator has two blocks, one for each component query. For this reason, hints in the first component query apply only to its optimization, not to the optimization of the second component query.

Optimizer Hint Syntax

Enclose hints within the comments of a SQL statement. You can use either style of comment. The hint delimiter (`+`) must come immediately after the comment delimiter. If you separate them by a space, the optimizer does not recognize that the comment contains hints.

Rules for Hints



Rules for Hints

- Place hints immediately after the first SQL keyword of a statement block.
- Each statement block can have only one hint comment, but it can contain multiple hints.
- Hints apply to only the statement block in which they appear.
- If a statement uses aliases, hints must reference the aliases rather than the table names.
- The optimizer ignores hints specified incorrectly without raising errors.

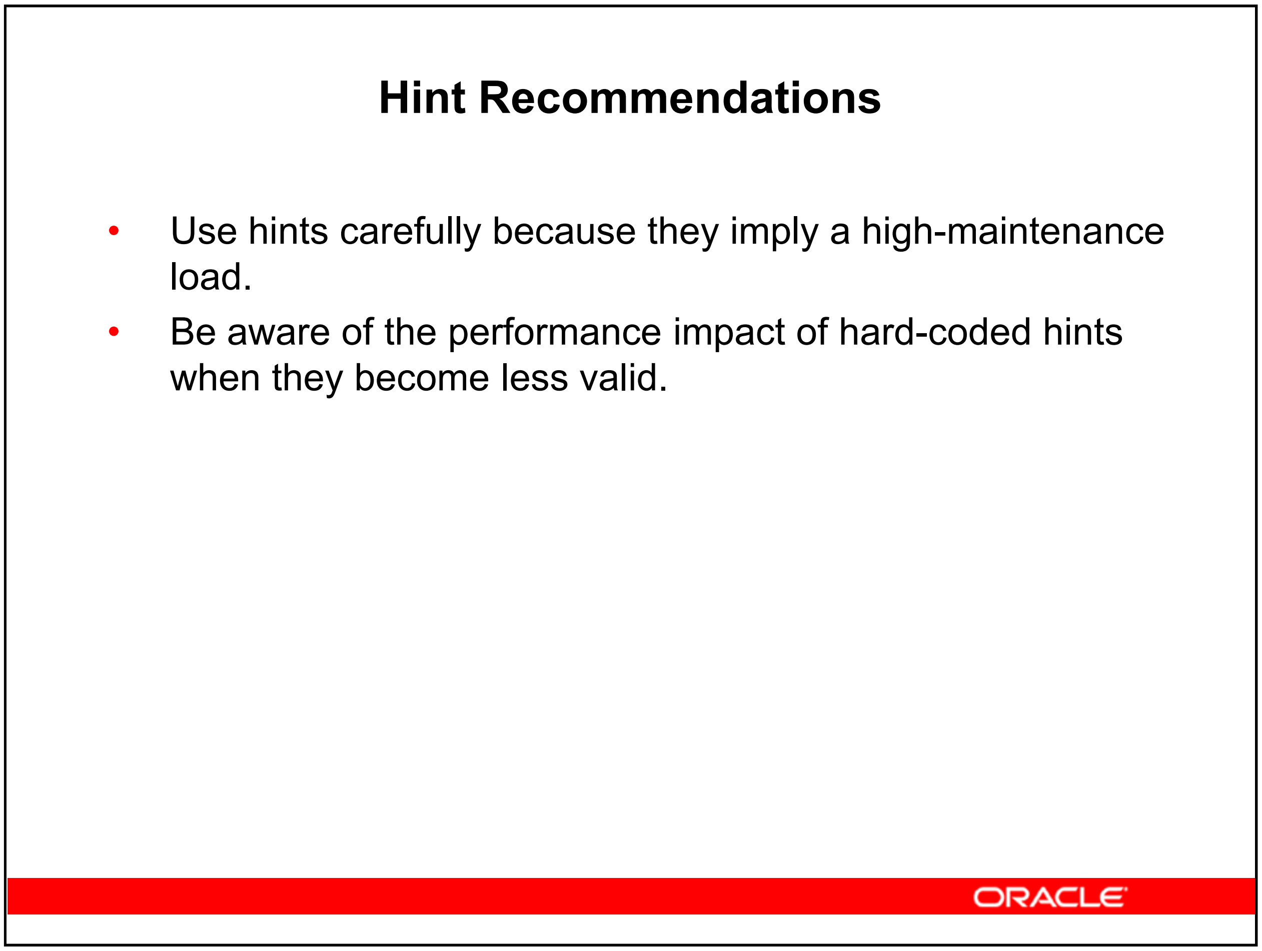
Rules for Hints

- You must place the hint comment immediately after the first keyword (`MERGE`, `SELECT`, `INSERT`, `DELETE`, or `UPDATE`) of a SQL statement block.
- A statement block can have only one comment containing hints, but it can contain many hints inside that comment separated by spaces.
- Hints apply to only the statement block in which they appear and override instance- or session-level parameters.
- If a SQL statement uses aliases, hints must reference the aliases rather than the table names.

The Oracle optimizer ignores incorrectly specified hints. However, be aware of the following situations:

- You never get an error message.
- Other (correctly) specified hints in the same comment are considered.
- The Oracle optimizer also ignores combinations of conflicting hints.

Hint Recommendations



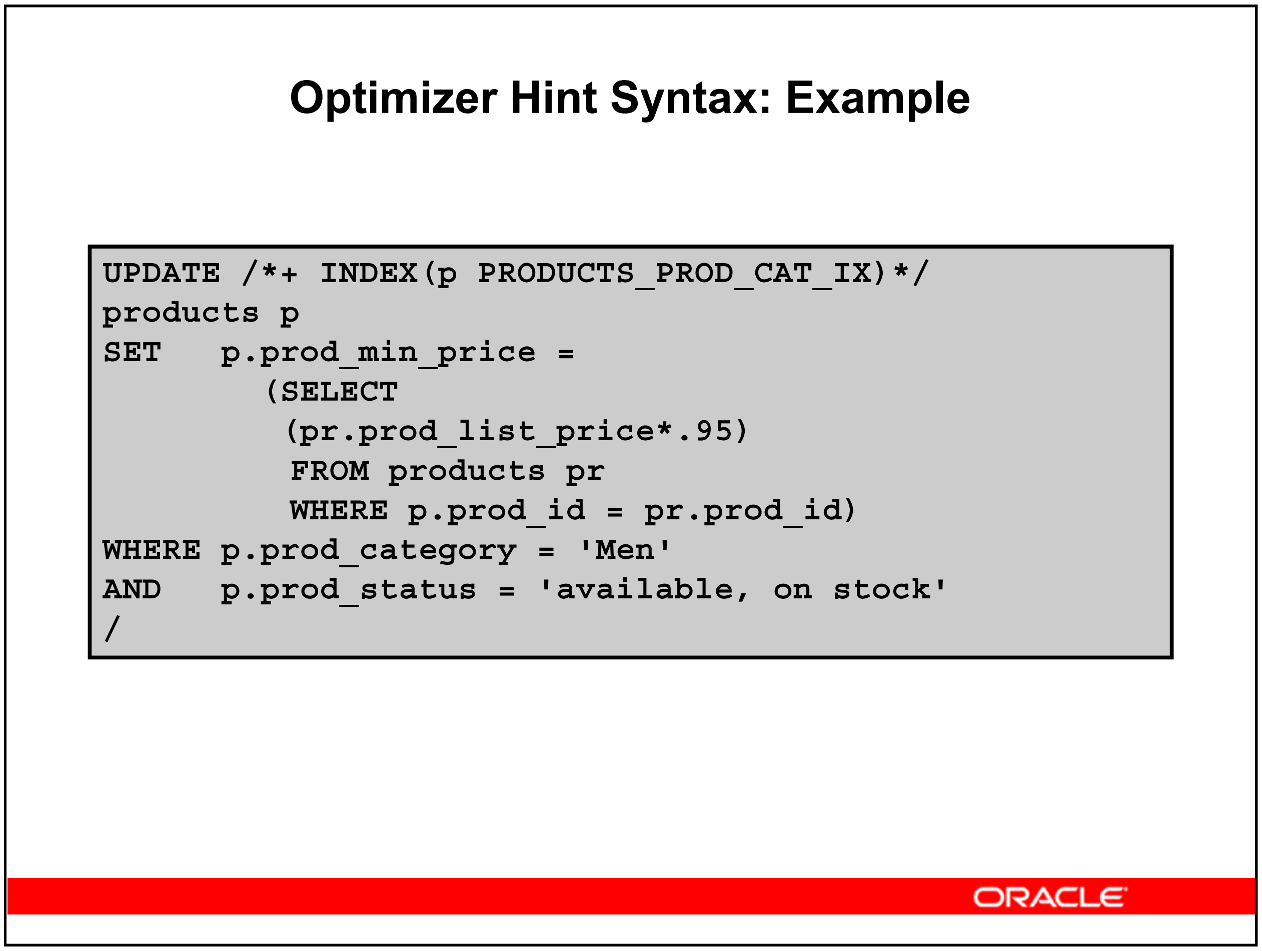
Hint Recommendations

- Use hints carefully because they imply a high-maintenance load.
- Be aware of the performance impact of hard-coded hints when they become less valid.

Hint Recommendations

- Use hints as a last remedy when tuning SQL statements.
- Hints may prevent the optimizer from using better execution plans.
- Hints may become less valid (or even invalid) when the database structure or contents change.

Optimizer Hint Syntax: Example



Optimizer Hint Syntax: Example

```
UPDATE /*+ INDEX(p PRODUCTS_PROD_CAT_IX)*/
products p
SET   p.prod_min_price =
        (SELECT
         (pr.prod_list_price*.95)
         FROM products pr
         WHERE p.prod_id = pr.prod_id)
WHERE p.prod_category = 'Men'
AND   p.prod_status = 'available, on stock'
/
```

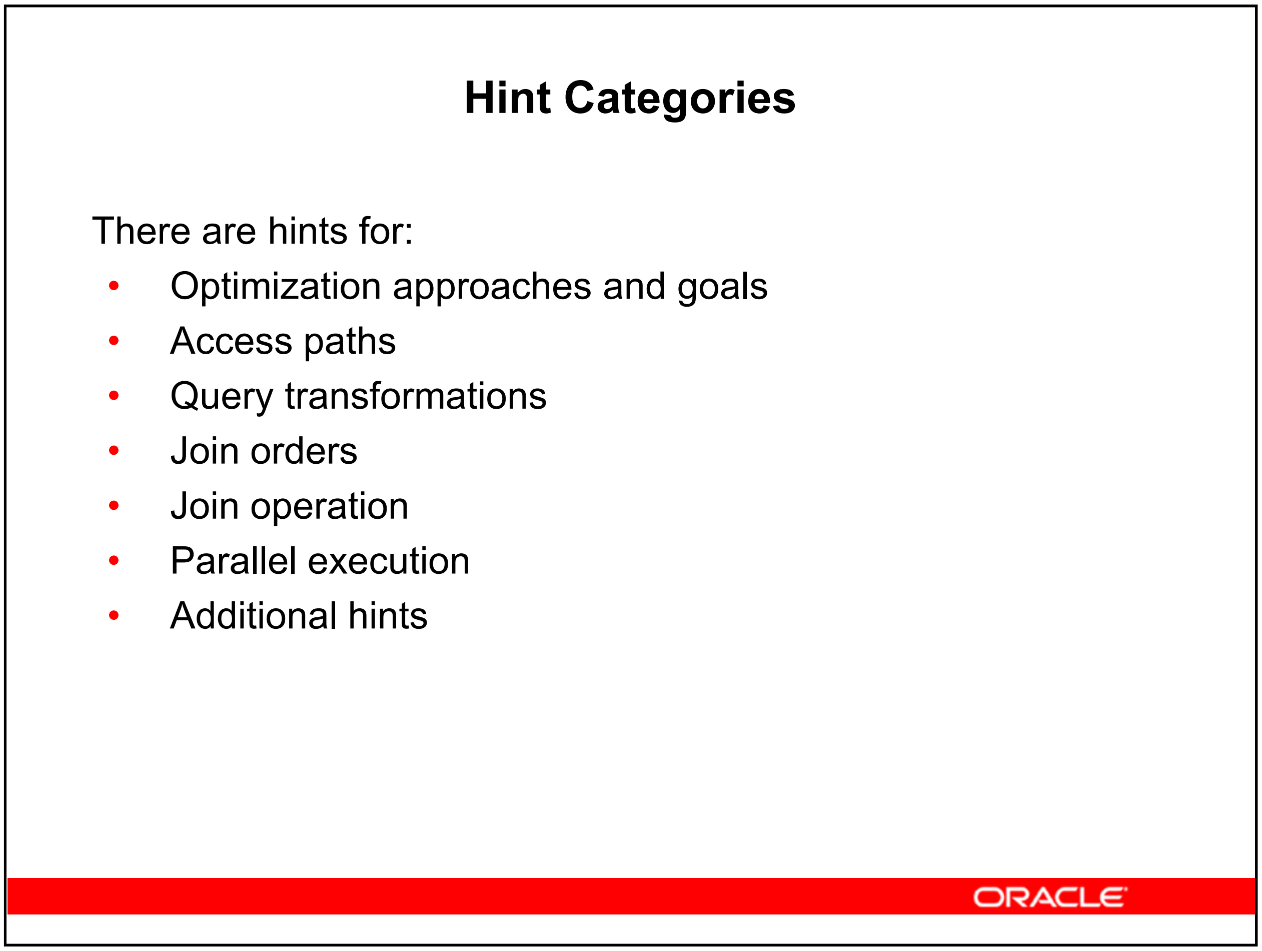
Optimizer Hint Syntax: Example

The slide shows an example with a hint that advises the cost-based optimizer (CBO) to use the index. The execution plan is as follows:

```
Execution Plan
---------------------------------------------------------
   0      UPDATE STATEMENT Optimizer=ALL_ROWS (Cost=3 …)
   1    0   UPDATE OF 'PRODUCTS'
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS' (TABLE) (Cost…)
   3    2      INDEX (RANGE SCAN) OF 'PRODUCTS_PROD_CAT_IX' (INDEX)
                  (cost…)
   4    1     TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS' (TABLE) (Cost…)
   5    4       INDEX (UNIQUE SCAN) OF 'PRODUCTS_PK' (INDEX (UNIQUE))
                 (Cost=0 …)
```

The hint shown in the example works only if an index called `PRODUCTS_PROD_CAT_IX` exists on the `PRODUCTS` table in the `PROD_CATEGORY` column.

Hint Categories



Hint Categories

There are hints for:

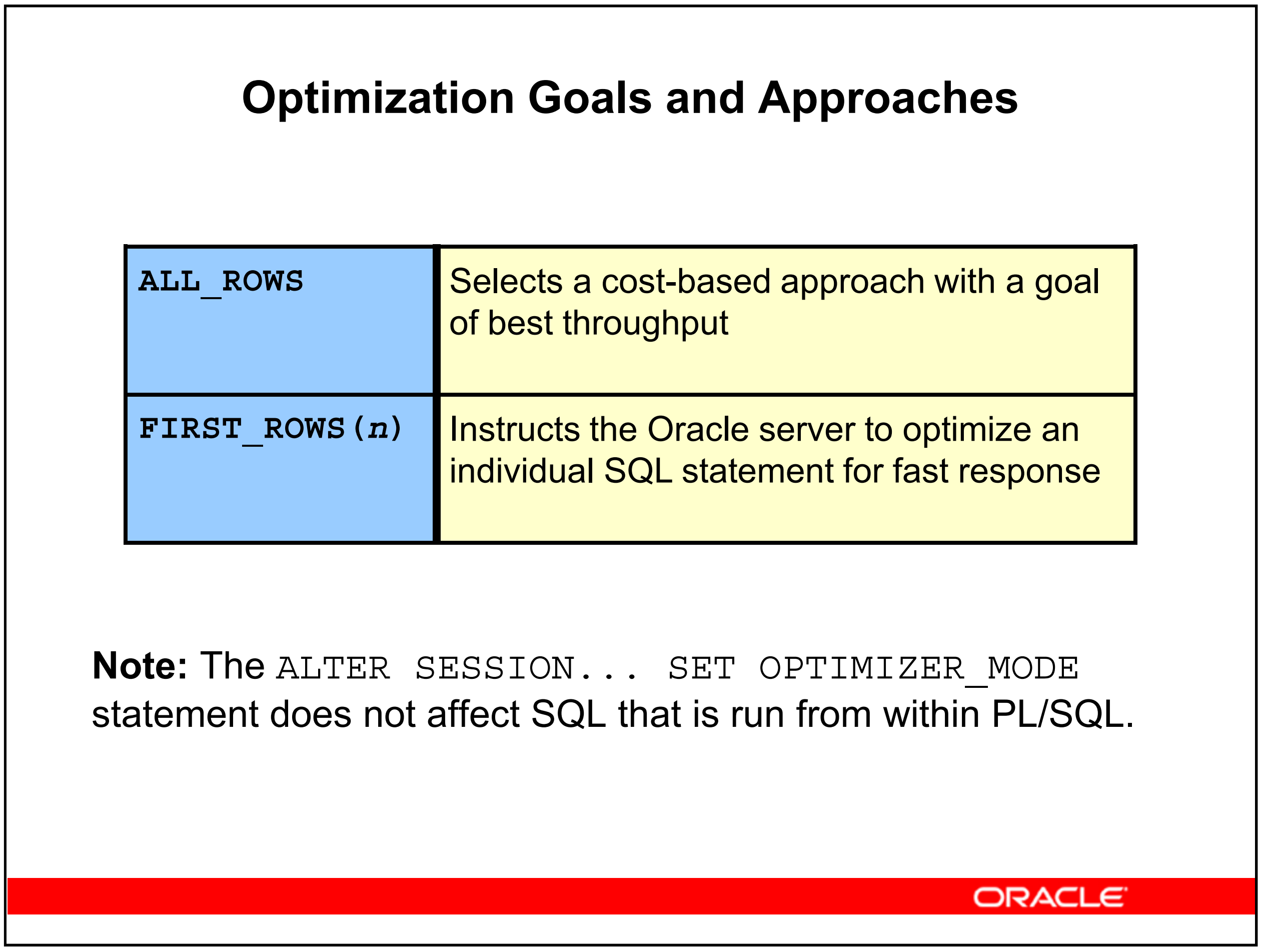
- Optimization approaches and goals
- Access paths
- Query transformations
- Join orders
- Join operation
- Parallel execution
- Additional hints

Hint Categories

Most of these hints are discussed in the following slides. Many of these hints accept the table and index names as arguments.

Note: Hints for parallel execution is not covered in this course.

Optimization Goals and Approaches



Optimization Goals and Approaches

`ALL_ROWS`	Selects a cost-based approach with a goal of best throughput
`FIRST_ROWS(`*`n`*`)`	Instructs the Oracle server to optimize an individual SQL statement for fast response

Note: The `ALTER SESSION... SET OPTIMIZER_MODE` statement does not affect SQL that is run from within PL/SQL.

Optimization Goals and Approaches

`ALL_ROWS`: The `ALL_ROWS` hint explicitly selects the cost-based approach to optimize a statement block with a goal of best throughput. That is, minimum total resource consumption.

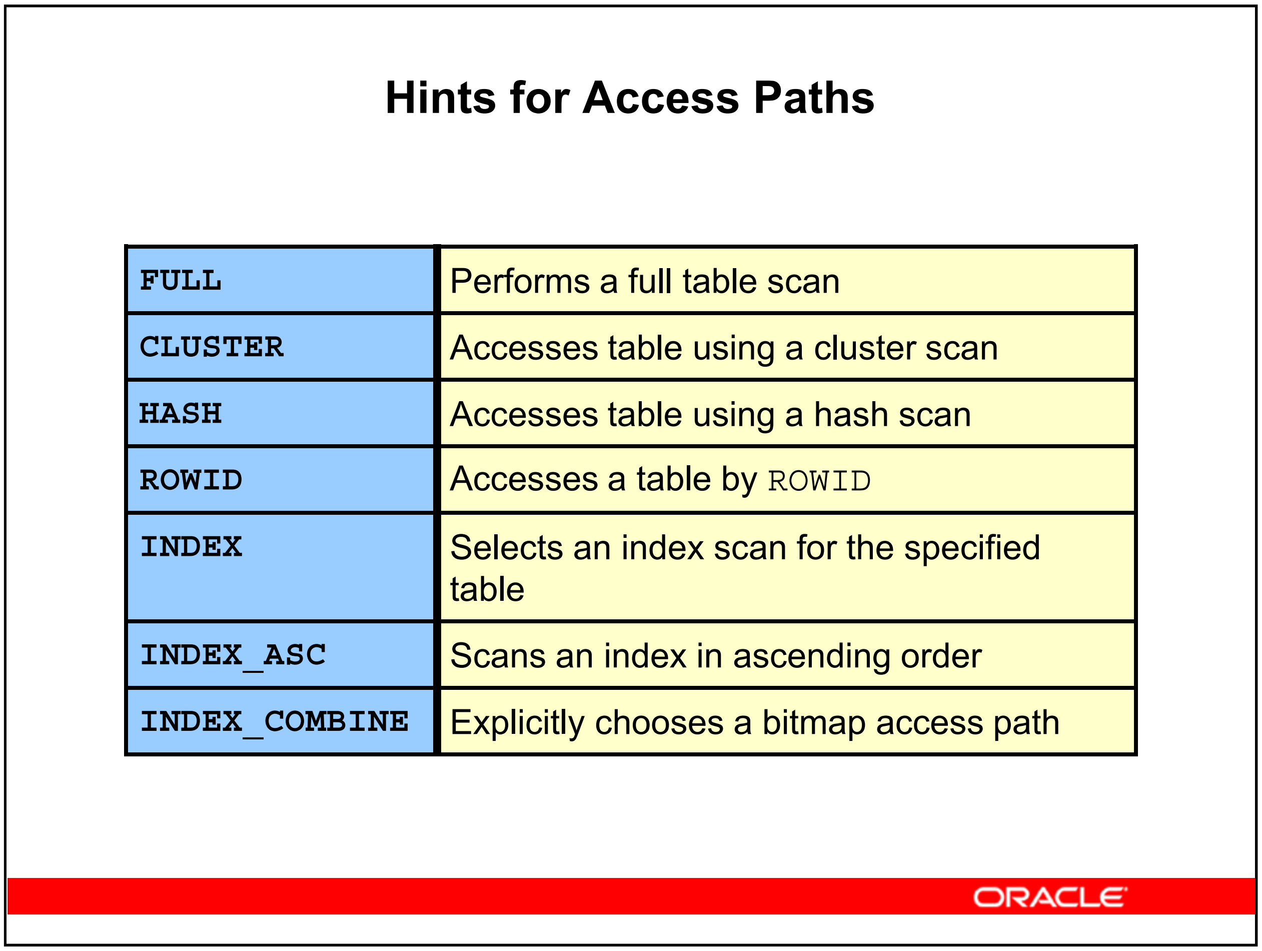
`FIRST_ROWS(`*`n`*`)`: The `FIRST_ROWS(`*`n`*`)` hint (where *n* is any positive integer) instructs the Oracle server to optimize an individual SQL statement for fast response. It instructs the server to select the plan that returns the first *n* rows most efficiently. The `FIRST_ROWS` hint, which optimizes for the best plan to return the first single row, is retained for backward compatibility and plan stability. The optimizer ignores this hint `SELECT` statement blocks that include any blocking operations, such as sorts or groupings. Such statements cannot be optimized for best response time because Oracle Database must retrieve all rows accessed by the statement before returning the first row. If you specify this hint in any such statement, the database optimizes for best throughput.

If you specify either the `ALL_ROWS` or the `FIRST_ROWS(`*`n`*`)` hint in a SQL statement, and if the data dictionary does not have statistics about tables accessed by the statement, then the optimizer uses default statistical values to estimate the missing statistics and to subsequently select an execution plan.

If you specify hints for access paths or join operations along with either the `ALL_ROWS` or `FIRST_ROWS(`*`n`*`)` hint, the optimizer gives precedence to the access paths and join operations specified by the hints.

Note: The `FIRST_ROWS` hints are probably the most useful hints.

Hints for Access Paths



Hints for Access Paths

`FULL`	Performs a full table scan
`CLUSTER`	Accesses table using a cluster scan
`HASH`	Accesses table using a hash scan
`ROWID`	Accesses a table by `ROWID`
`INDEX`	Selects an index scan for the specified table
`INDEX_ASC`	Scans an index in ascending order
`INDEX_COMBINE`	Explicitly chooses a bitmap access path

ORACLE

Hints for Access Paths

Specifying one of these hints causes the optimizer to choose the specified access path only if the access path is available based on the existence of an index and on the syntactic constructs of the SQL statement. If a hint specifies an unavailable access path, the optimizer ignores it. You must specify the table to be accessed exactly as it appears in the statement. If the statement uses an alias for the table, use the alias rather than the table name in the hint. The table name in the hint should not include the schema name if the schema name is present in the statement.

`FULL`: The `FULL` hint explicitly selects a full table scan for the specified table. For example:

```
SELECT /*+ FULL(e) */ employee_id, last_name
FROM hr.employees e WHERE last_name LIKE 'K%';
```

The Oracle server performs a full table scan on the employees table to execute this statement, even if there is an index on the `last_name` column that is made available by the condition in the `WHERE` clause.

`CLUSTER`: The `CLUSTER` hint instructs the optimizer to use a cluster scan to access the specified table. This hint applies only to clustered tables.

`HASH`: The `HASH` hint instructs the optimizer to use a hash scan to access the specified table. This hint applies only to tables stored in a table cluster.

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

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. However, it is better if you use INDEX_COMBINE rather than INDEX for bitmap indexes because it is a more versatile hint. This hint can optionally specify one or more indexes.

If this hint specifies a single available index, 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, the optimizer considers the cost of a scan on each index in the list and then performs the index scan with the lowest cost. The optimizer can also choose to scan multiple indexes from this list and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan or a scan on an index not listed in the hint.

If this hint specifies no indexes, the optimizer considers the cost of a scan on each available index on the table and then performs the index scan with the lowest cost. The optimizer can also choose to scan multiple indexes and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan.

INDEX_ASC: The INDEX_ASC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, the Oracle server scans the index entries in ascending order of their indexed values. Because the server's default behavior for a range scan is to scan index entries in the ascending order of their indexed values, this hint does not specify anything more than the INDEX hint. However, you might want to use the INDEX_ASC hint to specify ascending range scans explicitly, should the default behavior change.

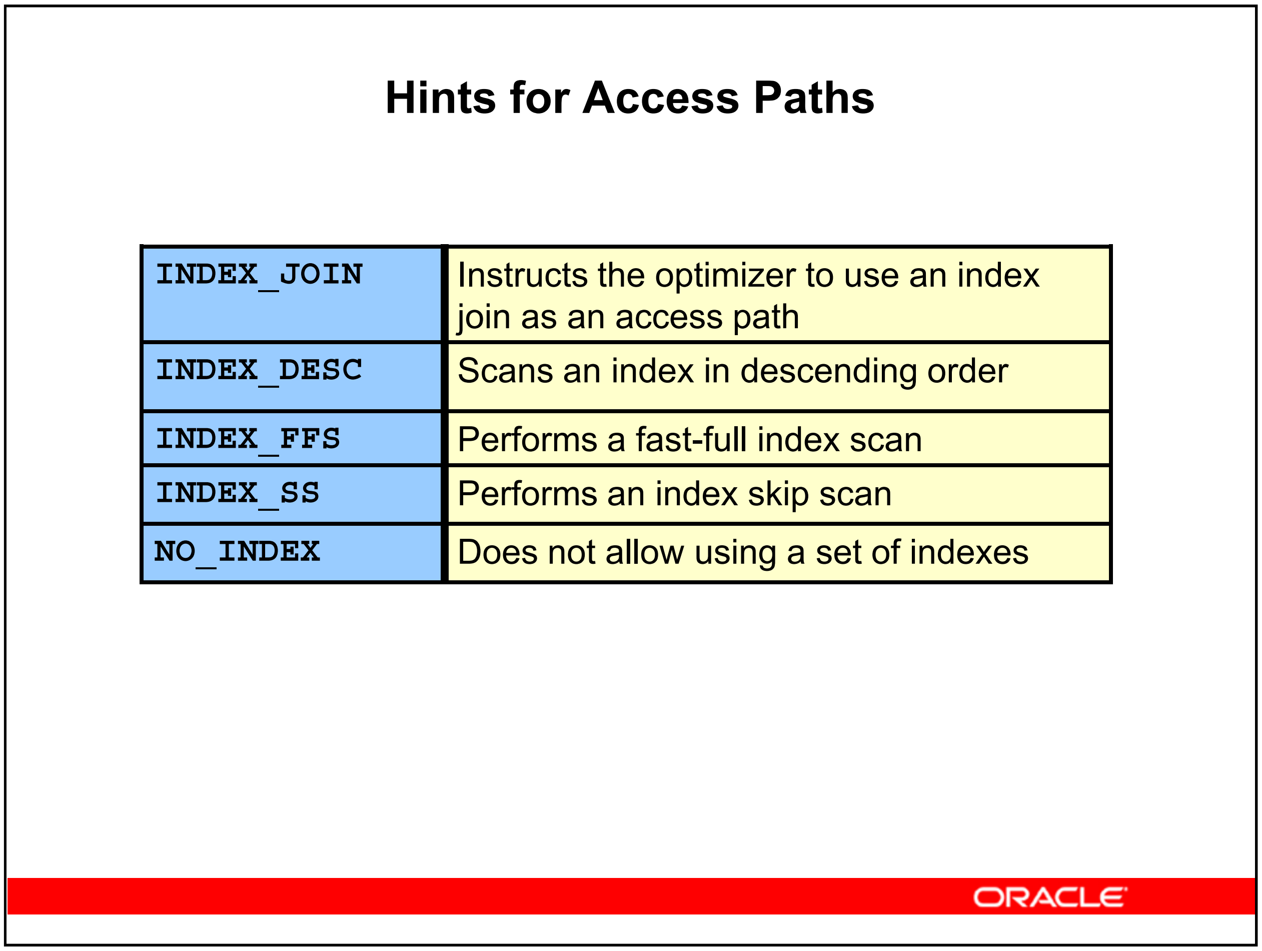
INDEX_COMBINE: The INDEX_COMBINE hint explicitly chooses a bitmap access path for the table. If no indexes are given as arguments for the INDEX_COMBINE hint, the optimizer uses a Boolean combination of bitmap indexes that has the best cost estimate for the table. If certain indexes are given as arguments, the optimizer tries to use some Boolean combination of those particular bitmap indexes.

For example:

```
SELECT /*+INDEX_COMBINE(customers cust_gender_bix cust_yob_bix)*/ *
FROM customers WHERE cust_year_of_birth < 70 AND cust_gender = 'M';
```

Note: For INDEX, INDEX_FFS, and INDEX_SS, there are counter hints, NO_INDEX, NO_INDEX_FFS, and NO_INDEX_SS, respectively to avoid using those paths.

Hints for Access Paths



Hints for Access Paths

`INDEX_JOIN`	Instructs the optimizer to use an index join as an access path
`INDEX_DESC`	Scans an index in descending order
`INDEX_FFS`	Performs a fast-full index scan
`INDEX_SS`	Performs an index skip scan
`NO_INDEX`	Does not allow using a set of indexes

Hints for Access Paths (continued)

`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, the following query uses an index join to access the `employee_id` and `department_id` columns, both of which are indexed in the employees table:

```
SELECT /*+index_join(employees emp_emp_id_pk emp_department_ix)*/
    employee_id, department_id
    FROM hr.employees WHERE department_id > 50;
```

`INDEX_DESC`: The `INDEX_DESC` hint instructs the optimizer to use a descending index scan for the specified table. If the statement uses an index range scan and the index is ascending, the system scans the index entries in the descending order of their indexed values. In a partitioned index, the results are in the descending order within each partition. For a descending index, this hint effectively cancels out the descending order, resulting in a scan of the index entries in the ascending order. The `INDEX_DESC` hint explicitly chooses an index scan for the specified table.

For example:

```
SELECT /*+ INDEX_DESC(a ord_order_date_ix) */ a.order_date,
   a.promotion_id, a.order_id
   FROM oe.orders a WHERE a.order_date < '01-jan-1985';
```

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

For example:

```
SELECT /*+ INDEX_FFS ( o order_pk ) */ COUNT(*)
FROM order_items l, orders o
WHERE l.order_id > 50 AND l.order_id = o.order_id;
```

`INDEX_SS`: The `INDEX_SS` hint instructs the optimizer to perform an index skip scan for the specified indexes of the specified table. If the statement uses an index range scan, the system scans the index entries in the ascending order of their indexed values. In a partitioned index, the results are in the ascending order within each partition. There are also `INDEX_SS_ASC` and `INDEX_SS_DESC` hints.

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

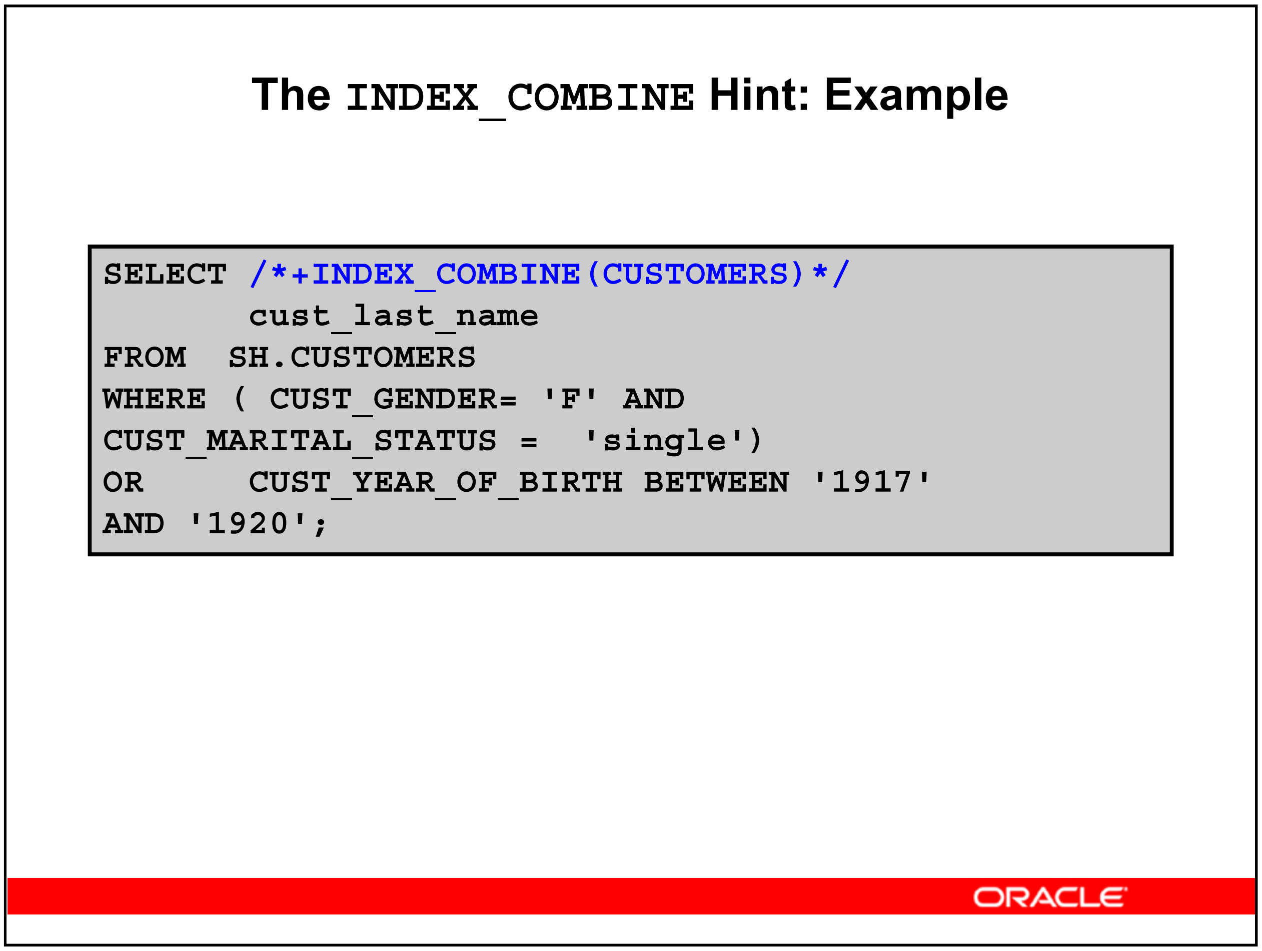
- If this hint specifies a single available index, the optimizer does not consider a scan on this index. Other indexes that are not specified are still considered.
- If this hint specifies a list of available indexes, the optimizer does not consider a scan on any of the specified indexes. Other indexes that are not specified in the list are still considered.
- If this hint specifies no indexes, 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, 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(employees emp_empid)*/ employee_id
FROM employees WHERE employee_id > 200;
```

The INDEX_COMBINE Hint: Example



The INDEX_COMBINE Hint: Example

```
SELECT /*+INDEX_COMBINE(CUSTOMERS)*/
       cust_last_name
FROM   SH.CUSTOMERS
WHERE ( CUST_GENDER= 'F' AND
CUST_MARITAL_STATUS =  'single')
OR      CUST_YEAR_OF_BIRTH BETWEEN '1917'
AND '1920';
```

ORACLE

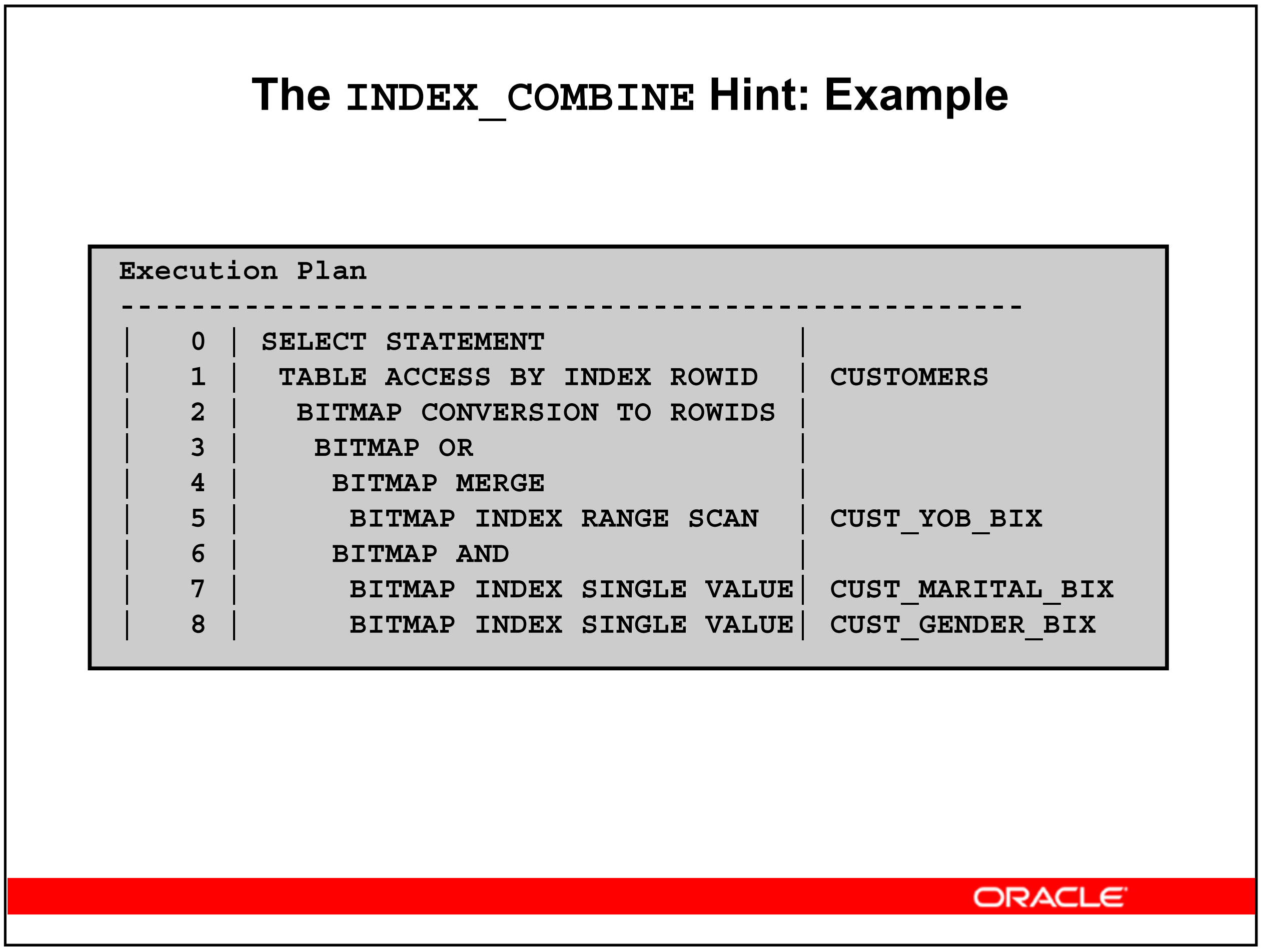
The `INDEX_COMBINE` Hint: Example

The `INDEX_COMBINE` hint is designed for bitmap index operations. Remember the following:

- If certain indexes are given as arguments for the hint, the optimizer tries to use some combination of those particular bitmap indexes.
- If no indexes are named in the hint, all indexes are considered to be hinted.
- The optimizer always tries to use hinted indexes, whether or not it considers them to be cost effective.

In the example in the slide, suppose that all the three columns that are referenced in the `WHERE` predicate of the statement in the slide (`CUST_MARITAL_STATUS`, `CUST_GENDER`, and `CUST_YEAR_OF_BIRTH`) have a bitmap index. When you enable `AUTOTRACE`, the execution plan of the statement might appear as shown in the next slide.

The INDEX_COMBINE Hint: Example



The `INDEX_COMBINE` Hint: Example

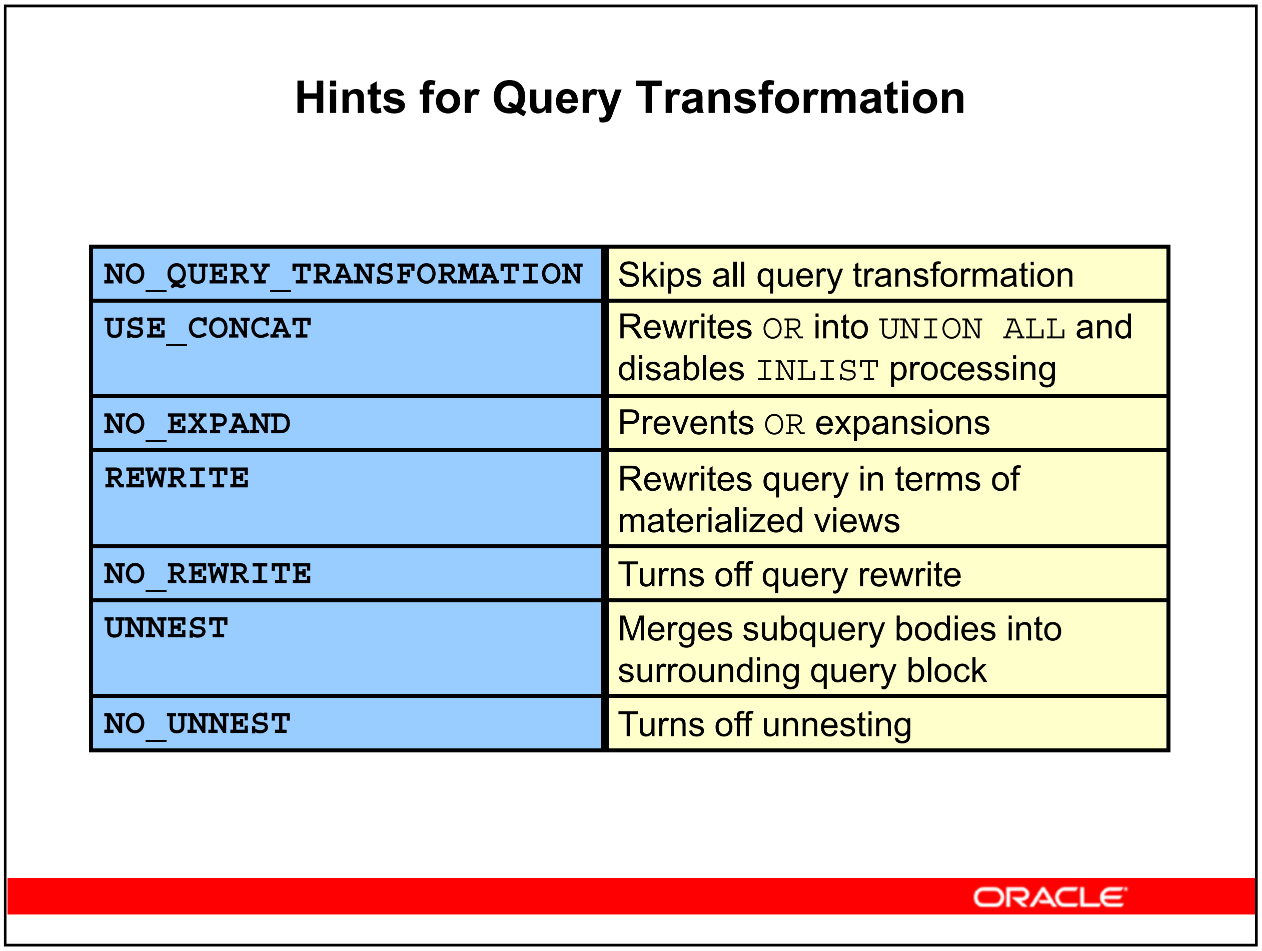
```
Execution Plan
---------------------------------------------------
|   0 | SELECT STATEMENT                |
|   1 |  TABLE ACCESS BY INDEX ROWID    | CUSTOMERS
|   2 |   BITMAP CONVERSION TO ROWIDS   |
|   3 |    BITMAP OR                    |
|   4 |     BITMAP MERGE                |
|   5 |      BITMAP INDEX RANGE SCAN    | CUST_YOB_BIX
|   6 |     BITMAP AND                  |
|   7 |      BITMAP INDEX SINGLE VALUE| CUST_MARITAL_BIX
|   8 |      BITMAP INDEX SINGLE VALUE| CUST_GENDER_BIX
```

The `INDEX_COMBINE` Hint: Example (continued)

In the example in the slide, the following bitmap row sources are used:

- **BITMAP CONVERSION TO ROWIDS:** Converts bitmaps into ROWIDs to access a table
- **COUNT:** Returns the number of entries if the actual values are not needed
- **BITMAP OR:** Computes the bitwise OR of two bitmaps
- **BITMAP AND:** Computes the bitwise AND of two bitmaps
- **BITMAP INDEX SINGLE VALUE:** Looks up the bitmap for a single key
- **BITMAP INDEX RANGE SCAN:** Retrieves bitmaps for a value range
- **BITMAP MERGE:** Merges several bitmaps resulting from a range scan into one (using a bitwise AND operator)

Hints for Query Transformation



Hints for Query Transformation

`NO_QUERY_TRANSFORMATION`	Skips all query transformation
`USE_CONCAT`	Rewrites `OR` into `UNION ALL` and disables `INLIST` processing
`NO_EXPAND`	Prevents `OR` expansions
`REWRITE`	Rewrites query in terms of materialized views
`NO_REWRITE`	Turns off query rewrite
`UNNEST`	Merges subquery bodies into surrounding query block
`NO_UNNEST`	Turns off unnesting

Hints for Query Transformation

`NO_QUERY_TRANSFORMATION`: The `NO_QUERY_TRANSFORMATION` hint instructs the optimizer to skip all query transformations, including but not limited to OR-expansion, view merging, subquery unnesting, star transformation, and materialized view rewrite.

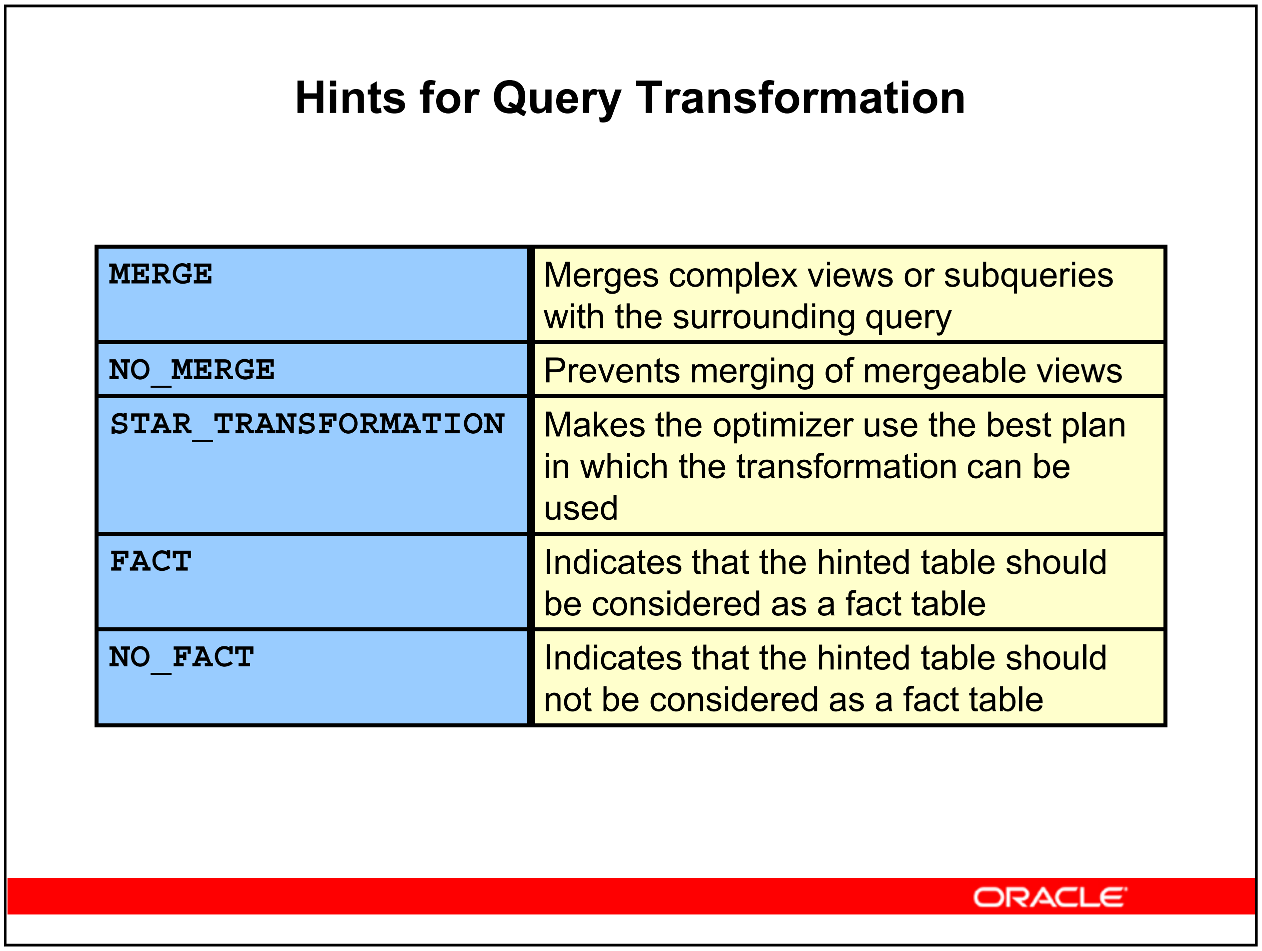
`USE_CONCAT`: The `USE_CONCAT` hint forces combined `OR` conditions in the `WHERE` clause of a query to be transformed into a compound query using the `UNION ALL` set operator. Generally, this transformation occurs only if the cost of the query using the concatenations is cheaper than the cost without them. The `USE_CONCAT` hint disables `IN`-list processing.

`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.

`REWRITE`: The `REWRITE` hint instructs the 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. This course does not deal with Materialized Views.

`UNNEST`: The `UNNEST` hint instructs the optimizer to unnest and merge the body of the subquery into the body of the query block that contains it, allowing the optimizer to consider them together when evaluating access paths and joins.

Hints for Query Transformation



Hints for Query Transformation

`MERGE`	Merges complex views or subqueries with the surrounding query
`NO_MERGE`	Prevents merging of mergeable views
`STAR_TRANSFORMATION`	Makes the optimizer use the best plan in which the transformation can be used
`FACT`	Indicates that the hinted table should be considered as a fact table
`NO_FACT`	Indicates that the hinted table should not be considered as a fact table

Hints for Query Transformation (continued)

`MERGE`: The `MERGE` hint lets you merge a view for each query. If a view's query contains a `GROUP BY` clause or a `DISTINCT` operator in the `SELECT` list, then the optimizer can merge the view's query into the accessing statement only if complex view merging is enabled. This is the case by default, but you can disable this mechanism using the `NO_MERGE` hint. Complex merging can also be used to merge an IN subquery into the accessing statement if the subquery is not correlated.

When the `MERGE` hint is used without an argument, it should be placed in the view query block. When `MERGE` is used with the view name as an argument, it should be placed in the surrounding query.

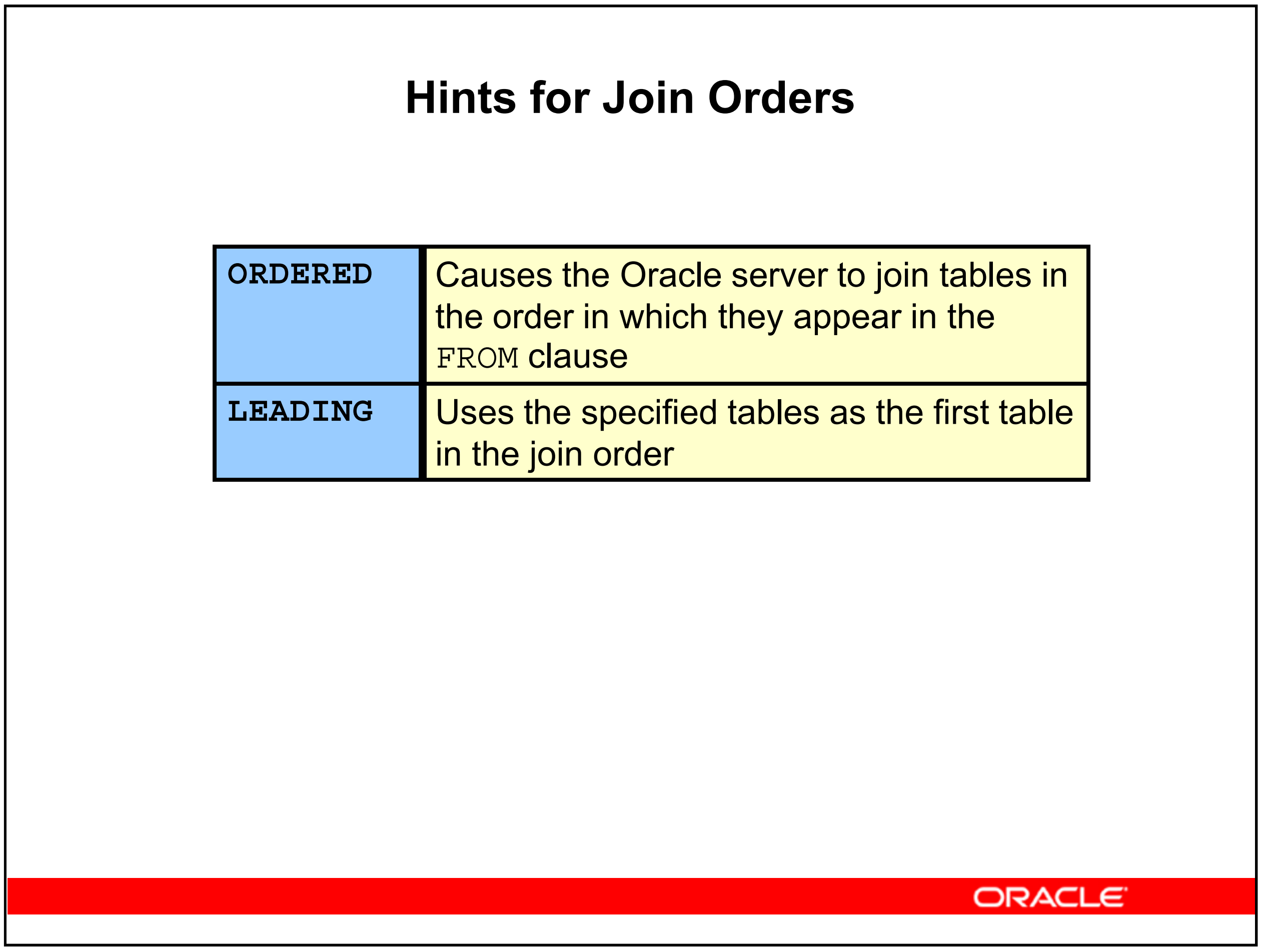
`NO_MERGE`: The `NO_MERGE` hint causes the Oracle server not to merge views that can be merged. This hint gives the user more influence over the way in which the view is accessed. When the `NO_MERGE` hint is used without an argument, it should be placed in the view query block. When `NO_MERGE` is used with the view name as an argument, it should be placed in the surrounding query.

`STAR_TRANSFORMATION`: The `STAR_TRANSFORMATION` hint causes the optimizer to 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 that is 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 generates the subqueries only if it seems reasonable to do so. If no subqueries are generated, there is no transformed query, and the best plan for the untransformed query is used regardless of the 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.

`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.



Hints for Join Orders

`ORDERED`	Causes the Oracle server to join tables in the order in which they appear in the `FROM` clause
`LEADING`	Uses the specified tables as the first table in the join order

Hints for Join Orders

The following hints are used to suggest join orders:

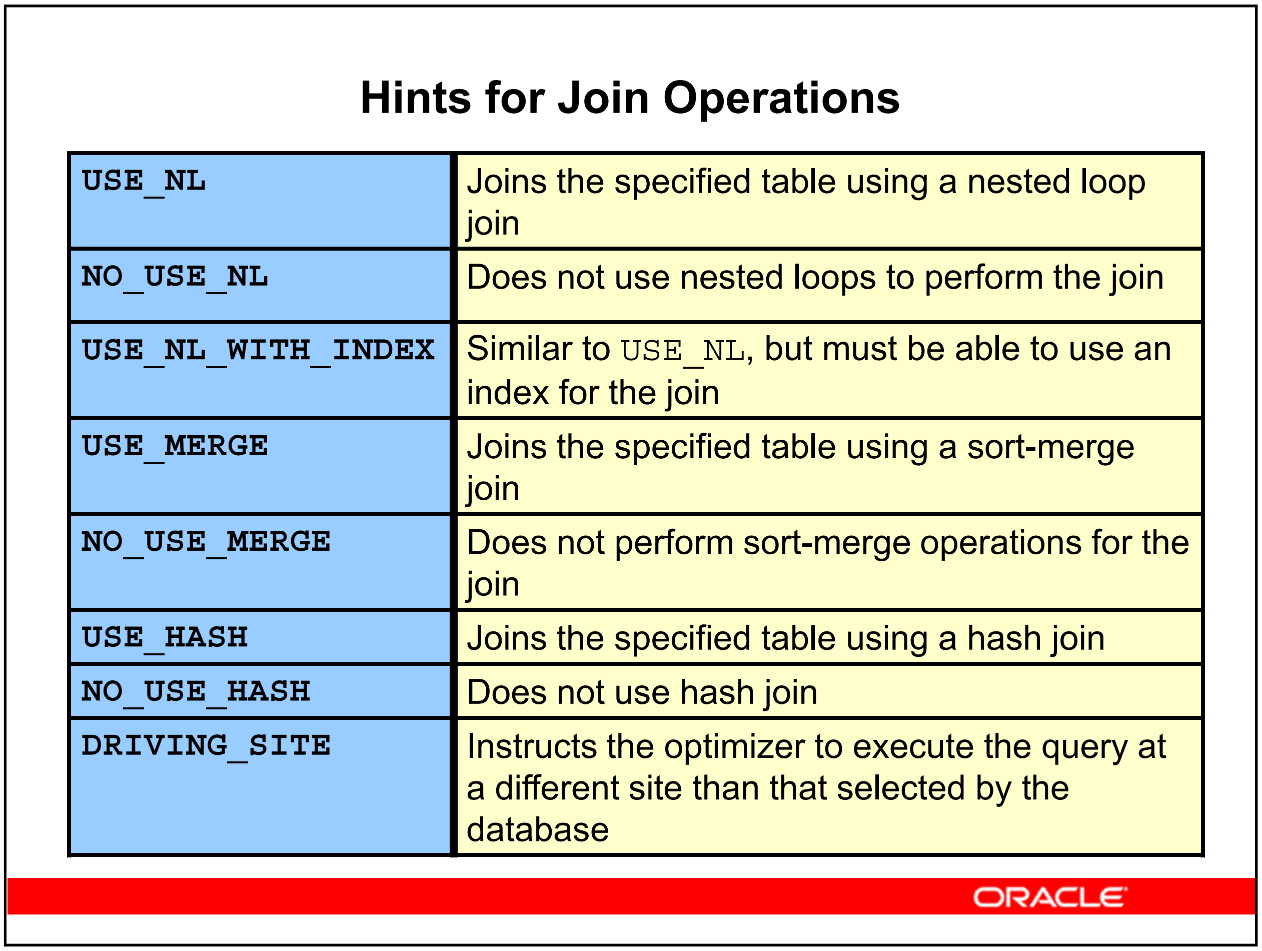
`ORDERED`: The `ORDERED` hint causes the Oracle server to join tables in the order in which they appear in the `FROM` clause. If you omit the `ORDERED` hint from a SQL statement performing a join, the optimizer selects the order in which to join the tables. You might want to use the `ORDERED` hint to specify a join order if you know something that the optimizer does not know about the number of rows that are selected from each table. With a nested loops example, 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 at the end. Then use the following hints:

```
/*+ ORDERED USE_NL(FACTS) INDEX(facts fact_concat) */
```

Here, `facts` is the table and `fact_concat` is the index. A more general method is to use the `STAR` hint.

`LEADING`: The `LEADING` hint instructs the optimizer to use the specified set of tables as the prefix in the execution plan. The `LEADING` hint is ignored if the tables specified cannot be joined first in the order specified because of dependencies in the join graph. If you specify two or more `LEADING` hints on different tables, all the hints are ignored. If you specify the `ORDERED` hint, it overrides all `LEADING` hints.

Hints for Join Operations



Hints for Join Operations

USE_NL	Joins the specified table using a nested loop join
NO_USE_NL	Does not use nested loops to perform the join
USE_NL_WITH_INDEX	Similar to `USE_NL`, but must be able to use an index for the join
USE_MERGE	Joins the specified table using a sort-merge join
NO_USE_MERGE	Does not perform sort-merge operations for the join
USE_HASH	Joins the specified table using a hash join
NO_USE_HASH	Does not use hash join
DRIVING_SITE	Instructs the optimizer to execute the query at a different site than that selected by the database

Hints for Join Operations

Each hint described here suggests a join operation for a table. In the hint, you must specify a table exactly the same way as it appears in the statement. If the statement uses an alias for the table, you must use the alias rather than the table name in the hint. However, the table name in the hint should *not* include the schema name if the schema name is present in the statement. Use of the `USE_NL` and `USE_MERGE` hints is recommended with the `ORDERED` hint. The Oracle server uses these hints when the referenced table is forced to be the inner table of a join; the hints are ignored if the referenced table is the outer table.

USE_NL: The `USE_NL` hint causes the Oracle server to join each specified table to another row source with a nested loops join, using the specified table as the inner table. If you want to optimize the statement for best response time or for the minimal elapsed time that is necessary to return the first row selected by the query, rather than for best throughput, then you can force the optimizer to select a nested loop join by using the `USE_NL` hint.

USE_NL_WITH_INDEX: The `USE_NL_WITH_INDEX` hint is similar to the `USE_NL` hint. However, if no index is specified, the optimizer must be able to use some index with at least one join predicate as the index key. If an index is specified, the optimizer must be able to use that index with at least one join predicate as the index key.

NO_USE_NL: The `NO_USE_NL` hint causes the optimizer to exclude the nested loops join.

However, in some cases tables can only be joined using nested loops. In such cases, the optimizer ignores the hint for those tables.

In many cases, a nested loop join returns the first row faster than a sort-merge join does. A nested loop 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. But 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.

In the following statement in which a nested loop is forced through a hint, `orders` is accessed through a full table scan and the `l.order_id = h.order_id` filter condition is applied to every row. For every row that meets the filter condition, `order_items` is accessed through the index `order_id`.

```
SELECT /*+ USE_NL(l h) */ h.customer_id, l.unit_price * l.quantity
   FROM oe.orders h ,oe.order_items l
   WHERE l.order_id = h.order_id;
```

Adding an `INDEX` hint to the query could avoid the full table scan on `orders`, resulting in an execution plan similar to one that is used on larger systems, even though it might not be particularly efficient here.

`USE_MERGE`: The `USE_MERGE` hint causes the Oracle server to join each specified table with another row source by using a sort-merge join, as in the following example:

```
SELECT /*+USE_MERGE(employees departments)*/ * FROM employees,
   departments WHERE employees.department_id =
   departments.department_id;
```

`NO_USE_MERGE`: The `NO_USE_MERGE` hint causes the optimizer to exclude the sort-merge join to join each specified table to another row source using the specified table as the inner table.

`USE_HASH`: The `USE_HASH` hint causes the Oracle server to join each specified table with another row source using a hash join, as in the following example:

```
SELECT /*+USE_HASH(l l2) */ l.order_date, l.order_id,
      l2.product_id, SUM(l2.unit_price*quantity)
 FROM oe.orders l, oe.order_items l2
 WHERE l.order_id = l2.order_id
 GROUP BY l2.product_id, l.order_date, l.order_id;
```

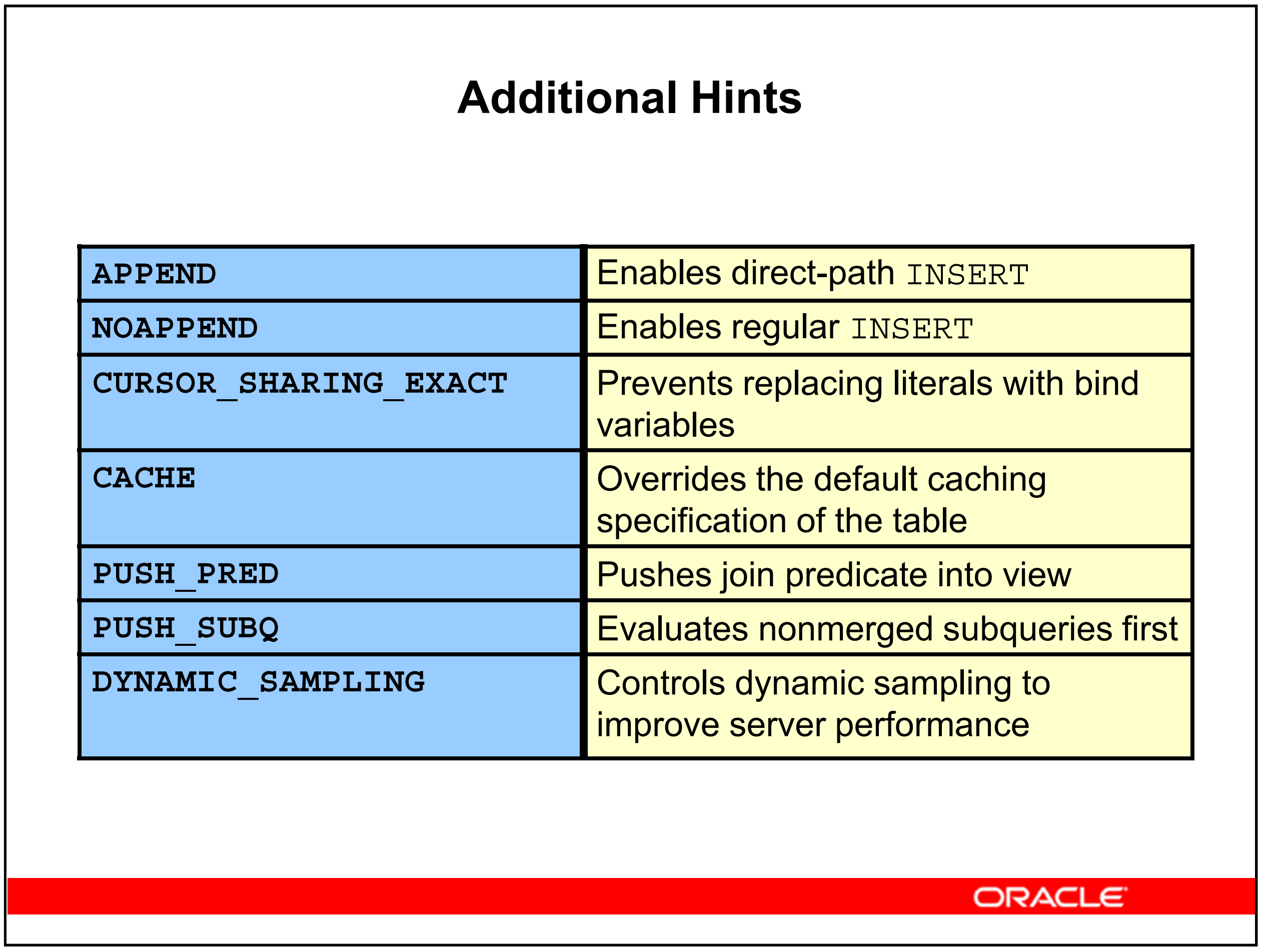
Here is another example:

```
SELECT /*+use_hash(employees departments)*/ *
 FROM hr.employees, hr.departments
 WHERE employees.department_id = departments.department_id;
```

`NO_USE_HASH`: The `NO_USE_HASH` hint causes the optimizer to exclude the hash join to join each specified table to another row source using the specified table as the inner table.

`DRIVING_SITE`: This hint instructs the optimizer to execute the query at a different site than that selected by the database. This hint is useful if you are using distributed query optimization to decide on which site a join should be executed.

Additional Hints



Additional Hints

`APPEND`	Enables direct-path `INSERT`
`NOAPPEND`	Enables regular `INSERT`
`CURSOR_SHARING_EXACT`	Prevents replacing literals with bind variables
`CACHE`	Overrides the default caching specification of the table
`PUSH_PRED`	Pushes join predicate into view
`PUSH_SUBQ`	Evaluates nonmerged subqueries first
`DYNAMIC_SAMPLING`	Controls dynamic sampling to improve server performance

Additional Hints

`APPEND`: The `APPEND` hint lets you enable direct-path `INSERT` if your database runs in serial mode. Your database is in serial mode if you are not using Enterprise Edition. Conventional `INSERT` is the default in serial mode, and direct-path `INSERT` is the default in parallel mode. In direct-path `INSERT`, data is appended to the end of the table rather than using existing space currently allocated to the table. As a result, direct-path `INSERT` can be considerably faster than the conventional `INSERT`.
Note: In Enterprise Edition, a session must be placed in parallel mode for direct-path insert to be the default.

`NOAPPEND`: The `NOAPPEND` hint disables direct-path `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.)

`CURSOR_SHARING_EXACT`: The Oracle server can replace literals in SQL statements with bind variables if it is safe to do so. This is controlled with the `CURSOR_SHARING` startup parameter. The `CURSOR_SHARING_EXACT` hint causes this behavior to be disabled. In other words, the Oracle server executes the SQL statement without any attempt to replace literals with bind variables.

`CACHE`: The `CACHE` hint instructs the optimizer to place the blocks retrieved for the table in the corresponding hot part of the buffer cache when a full table scan is performed. This hint is useful for small lookup tables.

The `CACHE` and `NOCACHE` hints affect system statistics table scans (long tables) and table scans (short tables), as shown in the `V$SYSSTAT` data dictionary view.

`PUSH_PRED`: The `PUSH_PRED` hint instructs the optimizer to push a join predicate into the view.

`PUSH_SUBQ`: The `PUSH_SUBQ` hint instructs the optimizer to evaluate nonmerged subqueries at the earliest possible step 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, evaluating the subquery earlier can improve performance. This hint has no effect if the subquery is applied to a remote table or one that is joined using a merge join.

`DYNAMIC_SAMPLING`: The `DYNAMIC_SAMPLING` hint lets you control dynamic sampling to improve server performance by determining more accurate selectivity and cardinality estimates. You can set the value of `DYNAMIC_SAMPLING` to a value from 0 to 10. The higher the level, the more effort the compiler puts into dynamic sampling and the more broadly it is applied. Sampling defaults to the cursor level unless you specify a table.

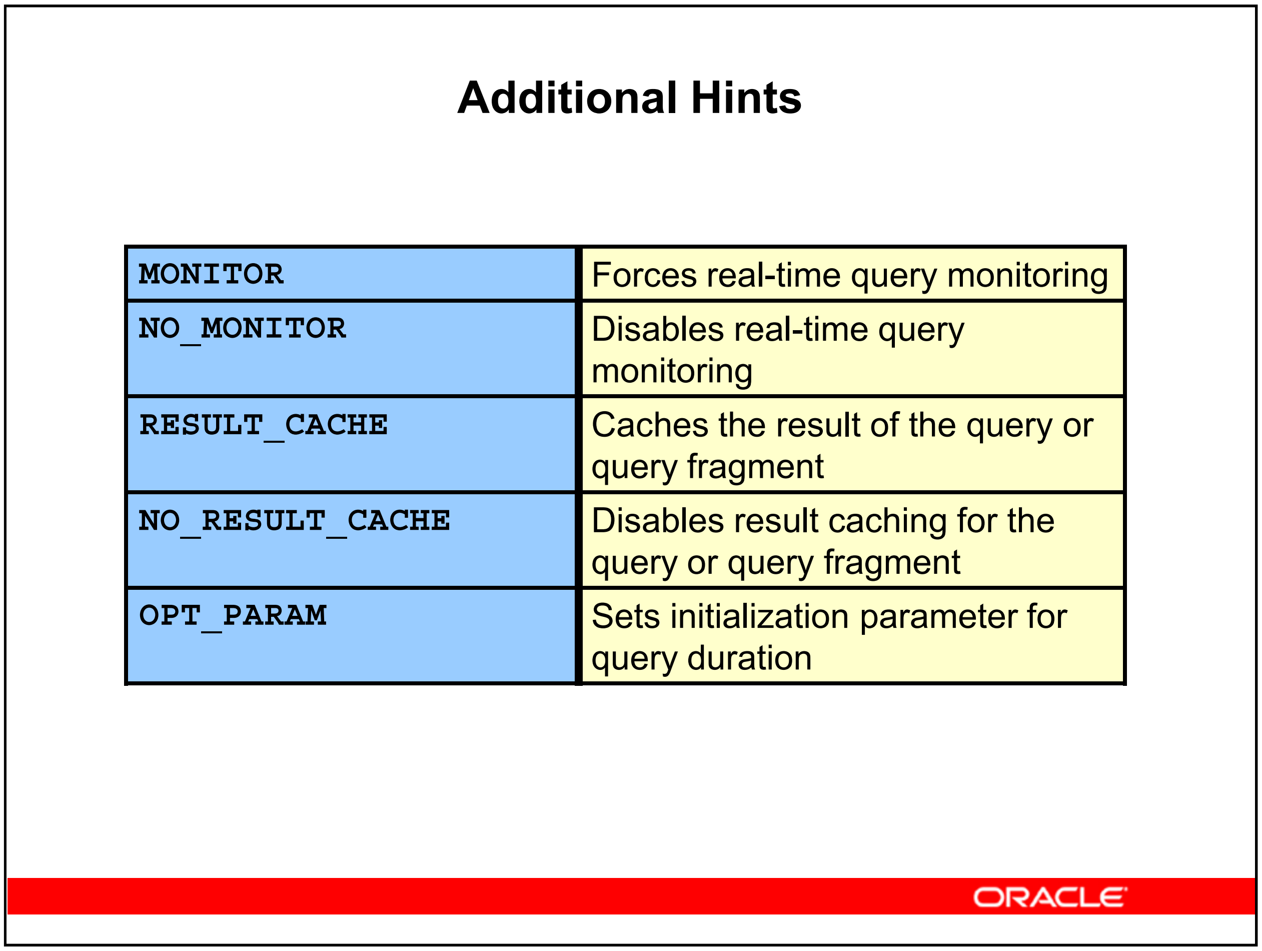
Consider the following example:

```
SELECT /*+ dynamic_sampling(1) */ * FROM ...
```

This example enables dynamic sampling if all the following conditions are true:

- There is more than one table in the query.
- At least one table has not been analyzed and has no indexes.
- The optimizer determines that a relatively expensive table scan is required for the table that has not been analyzed.

Additional Hints



Additional Hints

`MONITOR`	Forces real-time query monitoring
`NO_MONITOR`	Disables real-time query monitoring
`RESULT_CACHE`	Caches the result of the query or query fragment
`NO_RESULT_CACHE`	Disables result caching for the query or query fragment
`OPT_PARAM`	Sets initialization parameter for query duration

Additional Hints (continued)

`MONITOR`: The `MONITOR` hint forces real-time SQL monitoring for the query, even if the statement is not long running. This hint is valid only when the `CONTROL_MANAGEMENT_PACK_ACCESS` parameter is set to `DIAGNOSTIC+TUNING`.

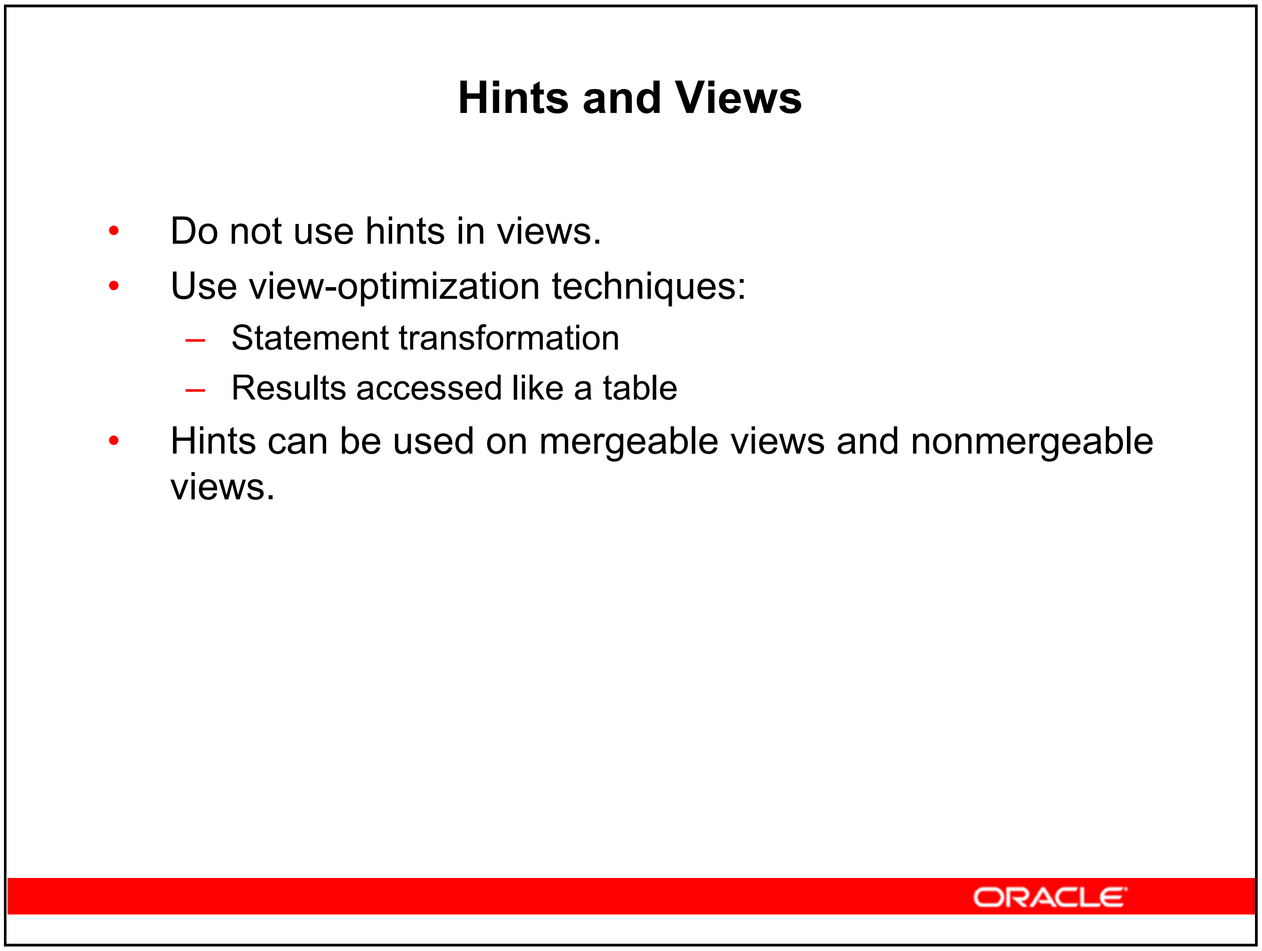
`NO_MONITOR`: The `NO_MONITOR` hint disables real-time SQL monitoring for the query.

`RESULT_CACHE`: The `RESULT_CACHE` hint instructs the database to cache the results of the current query or query fragment in memory and then to use the cached results in future executions of the query or query fragment.

`NO_RESULT_CACHE`: The optimizer caches query results in the result cache if the `RESULT_CACHE_MODE` initialization parameter is set to `FORCE`. In this case, the `NO_RESULT_CACHE` hint disables such caching for the current query.

`OPT_PARAM`: The `OPT_PARAM` hint lets you set an initialization parameter for the duration of the current query only. This hint is valid only for the following parameters: `OPTIMIZER_DYNAMIC_SAMPLING`, `OPTIMIZER_INDEX_CACHING`, `OPTIMIZER_INDEX_COST_ADJ`, `OPTIMIZER_SECURE_VIEW_MERGING`, and `STAR_TRANSFORMATION_ENABLED`

Hints and Views



Hints and Views

- Do not use hints in views.
- Use view-optimization techniques:
  - Statement transformation
  - Results accessed like a table
- Hints can be used on mergeable views and nonmergeable views.

Hints and Views

You should not use hints in or on views because views can be defined in one context and used in another; such hints can result in unexpected plans. In particular, hints in views are handled differently from hints on views depending on whether or not the view is mergeable into the top-level query.

View Optimization

The statement is normally transformed into an equivalent statement that accesses the view's base tables. The optimizer can use one of the following techniques to transform the statement:

- Merge the view's query into the referencing query block in the accessing statement.
- Push the predicate of the referencing query block inside the view.

When these transformations are impossible, the view's query is executed and the result is accessed as if it were a table. This appears as a `VIEW` step in execution plans.

Mergeable Views

The optimizer can merge a view into a referencing query block if the view definition does not contain the following:

- Set operators (`UNION`, `UNION ALL`, `INTERSECT`, `MINUS`)

- The `CONNECT BY` clause
- The `ROWNUM` pseudocolumn
- Group functions (`AVG`, `COUNT`, `MAX`, `MIN`, `SUM`) in the select list

Hints and Mergeable Views

Optimization-approach and goal hints can occur in a top-level query or in views:

- If there is such a hint in the top-level query, that hint is used regardless of any such hints in the views.
- If there is no top-level optimizer-mode hint, 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, all mode hints in the views are discarded and the session mode is used, whether default or user specified.

Access-method and join hints on referenced views are ignored unless the view contains a single table (or references another view with a single table). For such single-table views, an access-method hint or a join hint on the view applies to the table in the view.

Access-method and join hints can also appear in a view definition:

- If the view is a subquery (that is, if it appears in the `FROM` clause of a `SELECT` statement), all access-method and join hints in the view are preserved when the view is merged with the top-level query.
- For views that are not subqueries, access-method and join hints in the view are preserved only if the top-level query references no other tables or views (that is, if the `FROM` clause of the `SELECT` statement contains only the view).

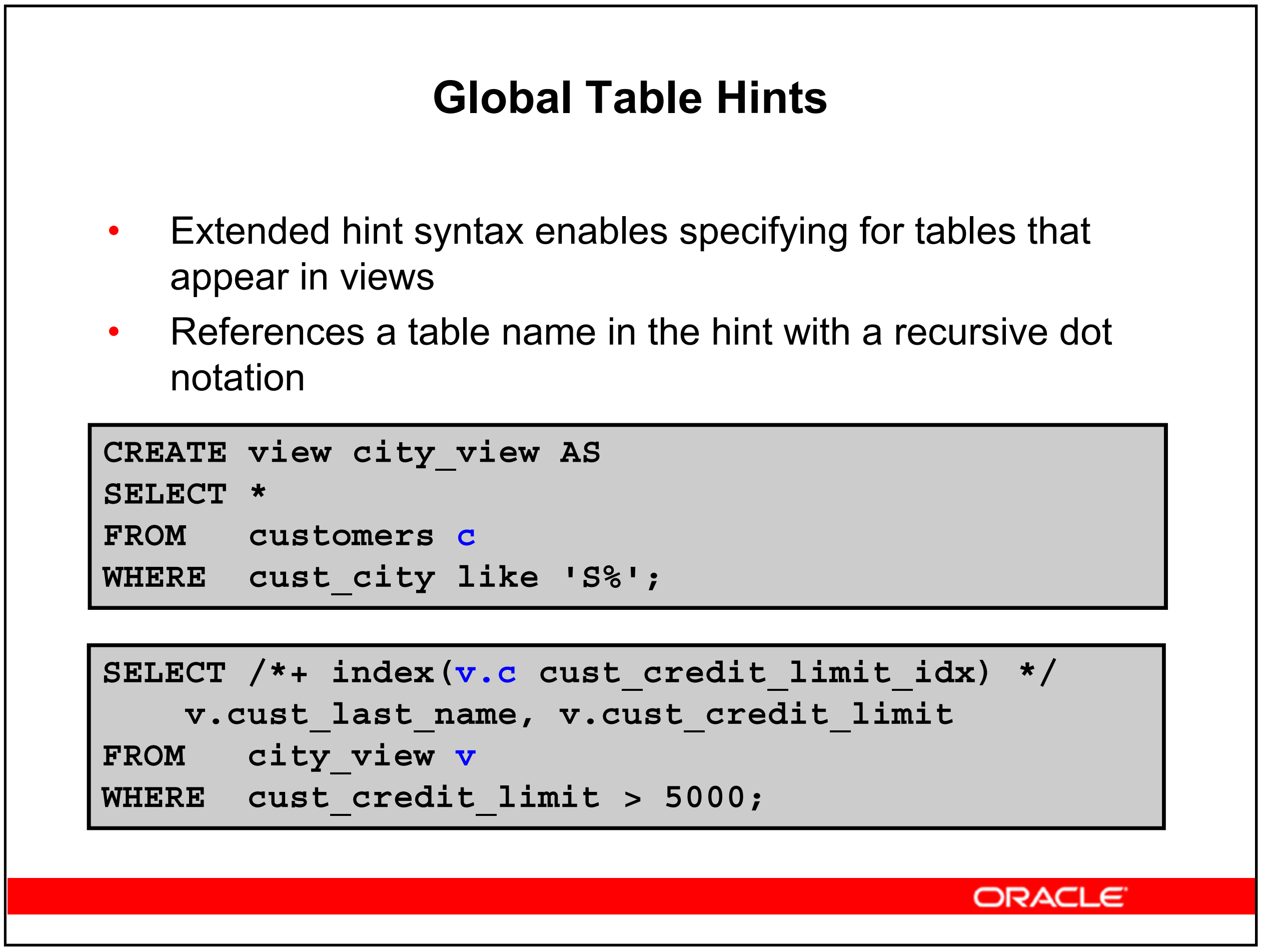
Hints and Nonmergeable Views

With nonmergeable views, optimizer-mode hints in the view are ignored. The top-level query decides the optimization mode.

Because nonmergeable views are optimized separately from the top-level query, access-method and join hints in the view are always preserved. For the same reason, access-method 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 Table Hints



Global Table Hints

- Extended hint syntax enables specifying for tables that appear in views
- References a table name in the hint with a recursive dot notation

```
CREATE view city_view AS
SELECT *
FROM   customers c
WHERE  cust_city like 'S%';
```

```
SELECT /*+ index(v.c cust_credit_limit_idx) */
    v.cust_last_name, v.cust_credit_limit
FROM   city_view v
WHERE  cust_credit_limit > 5000;
```

ORACLE

Global Table Hints

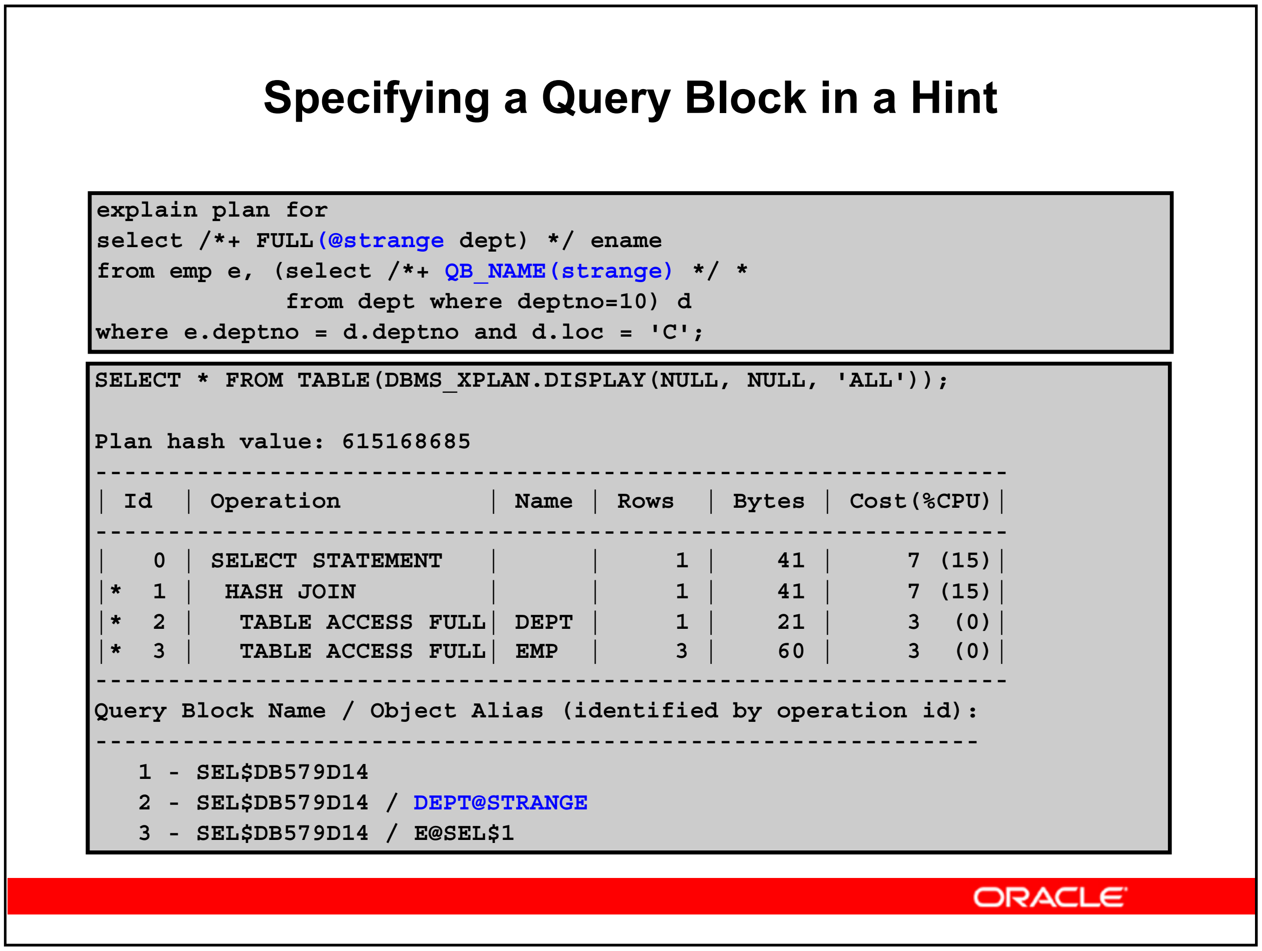
Hints that specify a table generally refer to tables in the `DELETE`, `SELECT`, or `UPDATE` query block in which the hint occurs, rather than to tables inside any views that are referenced by the statement. When you want to specify hints for tables that appear inside views, it is recommended that you use global hints instead of embedding the hint in the view.

The table hints can be transformed into global hints by using an extended table specification syntax that includes view names with the table name as shown in the slide. In addition, an optional query block name can precede the table specification.

For example, by using the global hint structure, you can avoid the modification of a view with the specification of an index hint in the body of view.

Note: If a global hint references a table name or alias that is used twice in the same query (for example, in a `UNION` statement), the hint applies to only the first instance of the table (or alias).

Specifying a Query Block in a Hint



Specifying a Query Block in a Hint

```
explain plan for
select /*+ FULL(@strange dept) */ ename
from emp e, (select /*+ QB_NAME(strange) */ *
             from dept where deptno=10) d
where e.deptno = d.deptno and d.loc = 'C';
```

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(NULL, NULL, 'ALL'));

Plan hash value: 615168685
---------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost(%CPU)|
---------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    41 |     7 (15)|
|*  1 |  HASH JOIN         |      |     1 |    41 |     7 (15)|
|*  2 |   TABLE ACCESS FULL| DEPT |     1 |    21 |     3  (0)|
|*  3 |   TABLE ACCESS FULL| EMP  |     3 |    60 |     3  (0)|
---------------------------------------------------------------
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
   1 - SEL$DB579D14
   2 - SEL$DB579D14 / DEPT@STRANGE
   3 - SEL$DB579D14 / E@SEL$1
```

ORACLE

Specifying a Query Block in a Hint

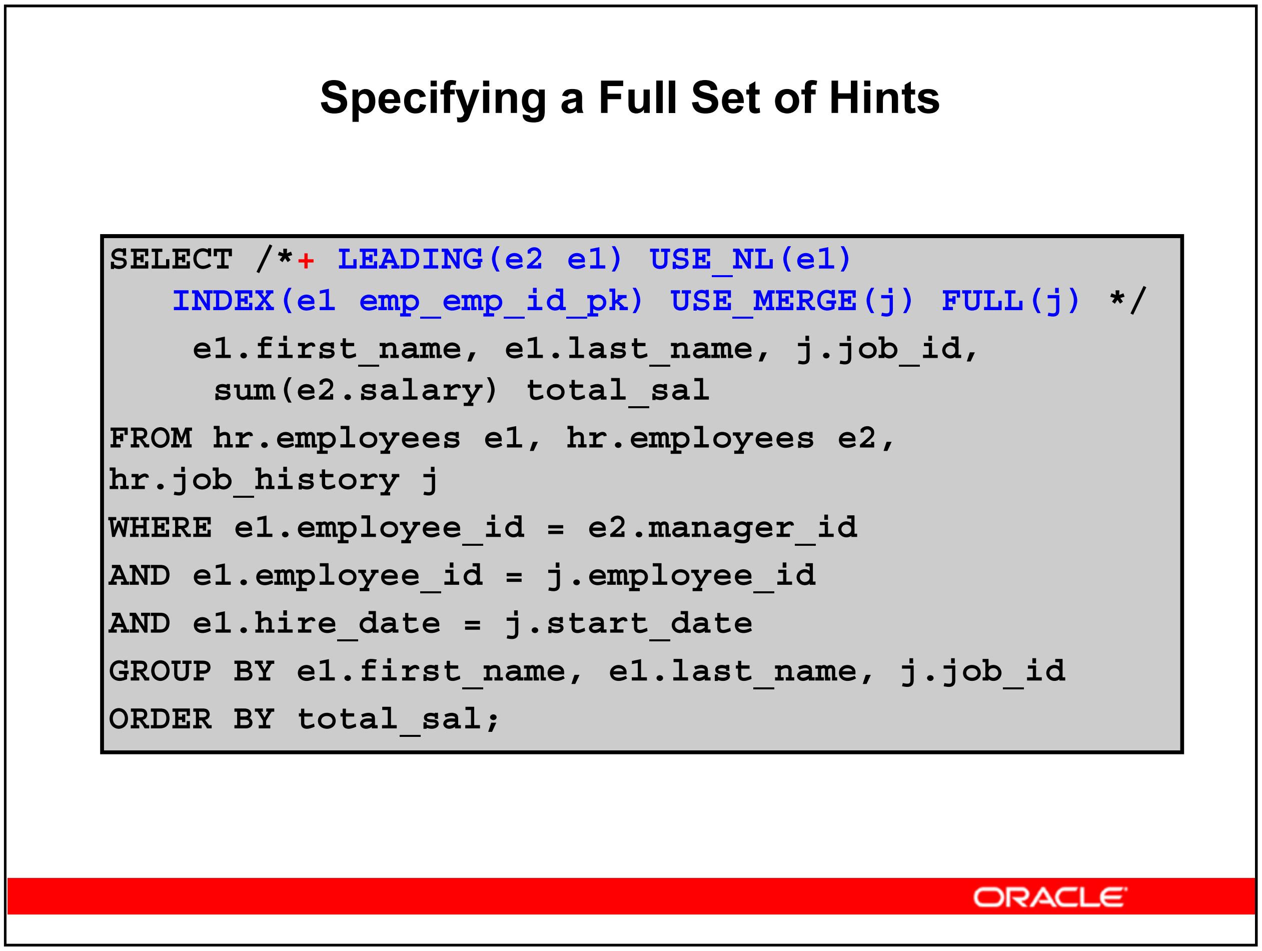
You can specify an optional query block name in many hints to specify the query block to which the hint applies. This syntax lets you specify in the outer query a hint that applies to an inline view.

The syntax of the query block argument is of the `@queryblock` form, where `queryblock` is an identifier that specifies a query block in the query. The `queryblock` identifier can either be system-generated or user-specified. When you specify a hint in the query block itself to which the hint applies, you do not have to specify the `@queryblock` syntax.

The slide gives you an example. You can see that the `SELECT` statement uses an inline view. The corresponding query block is given the name `strange` through the use of the `QB_NAME` hint.

The example assumes that there is an index on the `DEPTNO` column of the `DEPT` table so that the optimizer would normally choose that index to access the `DEPT` table. However, because you specify the `FULL` hint to apply to the `strange` query block in the main query block, the optimizer does not use the index in question. You can see that the execution plan exhibits a full table scan on the `DEPT` table. In addition, the output of the plan clearly shows the system-generated names for each query block in the original query.

Specifying a Full Set of Hints



Specifying a Full Set of Hints

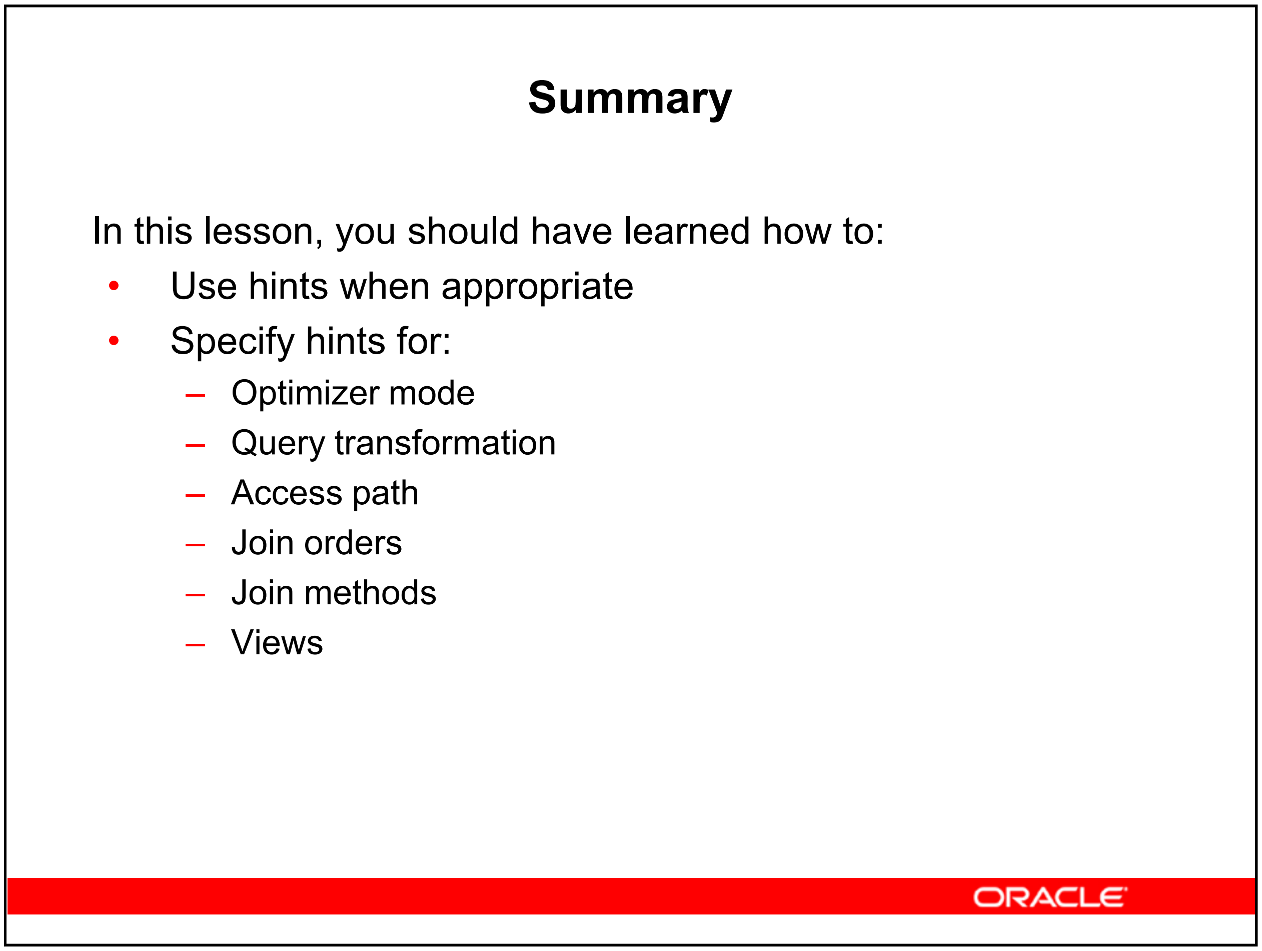
```
SELECT /*+ LEADING(e2 e1) USE_NL(e1)
   INDEX(e1 emp_emp_id_pk) USE_MERGE(j) FULL(j) */
    e1.first_name, e1.last_name, j.job_id,
     sum(e2.salary) total_sal
FROM hr.employees e1, hr.employees e2,
hr.job_history j
WHERE e1.employee_id = e2.manager_id
AND e1.employee_id = j.employee_id
AND e1.hire_date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

Specifying a Full Set of Hints

When using hints, you might sometimes need to specify a full set of hints to ensure the optimal execution plan. For example, if you have a very complex query consisting of many table joins, and if you specify only the `INDEX` hint for a given table, then the optimizer needs to determine the remaining access paths to be used as well as the corresponding join methods. Therefore, even though you gave the `INDEX` hint, the optimizer might not necessarily use that hint because the optimizer might have determined that the requested index cannot be used due to the join methods and access paths that were selected by the optimizer.

In the example, the `LEADING` hint specifies the exact join order to be used. The join methods to be used on the different tables are also specified.

Summary



Summary

In this lesson, you should have learned how to:

- Use hints when appropriate
- Specify hints for:
  - Optimizer mode
  - Query transformation
  - Access path
  - Join orders
  - Join methods
  - Views

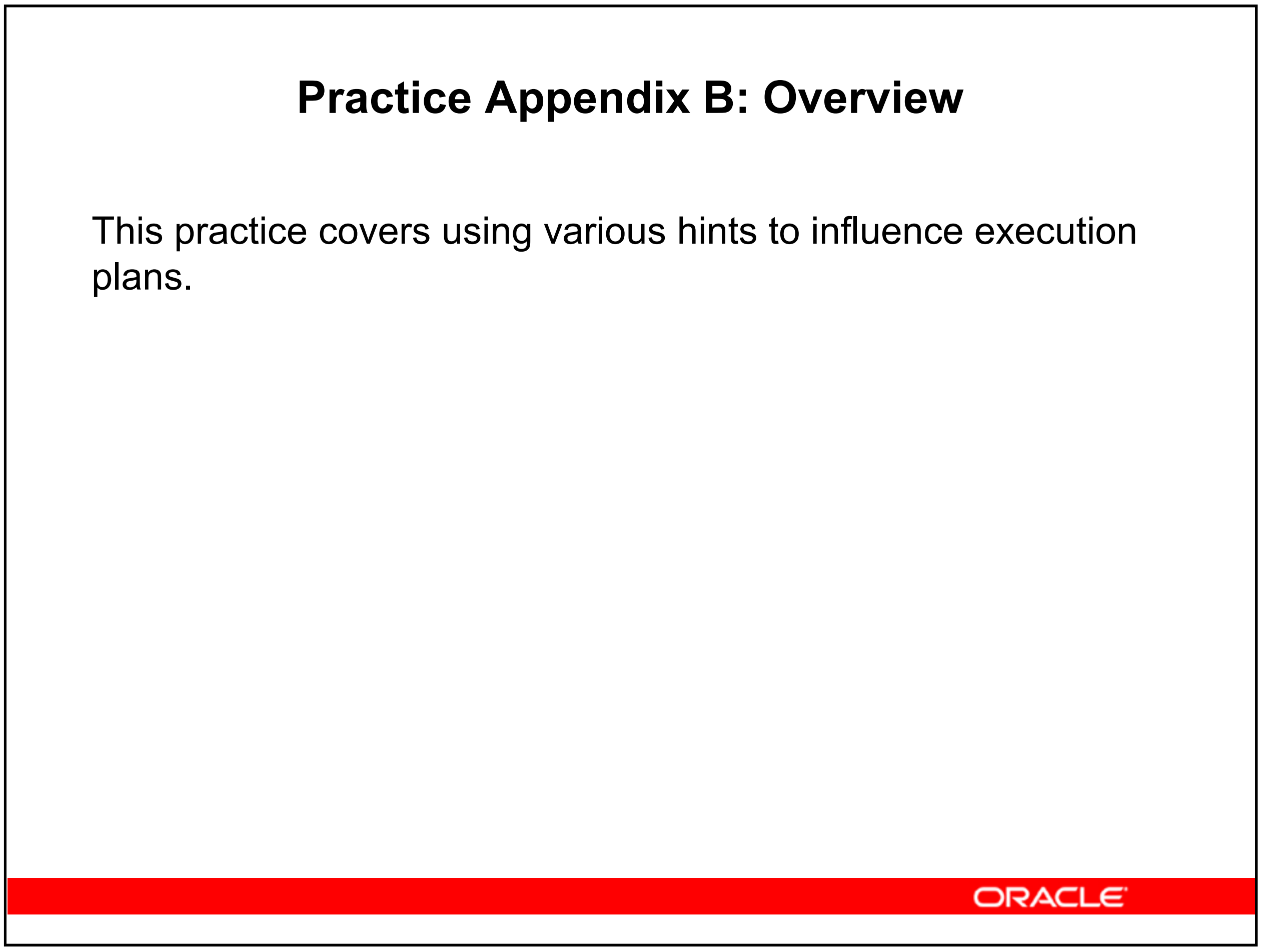
Summary

In this lesson, you should have learned about additional optimizer settings and hints.

By using hints, you can influence the optimizer at the statement level. Use hints as a last remedy when tuning SQL statements. There are several hint categories, one of which includes hints for access-path methods.

To specify a hint, use the hint syntax in the SQL statement.

Practice Appendix B: Overview



Practice Appendix B: Overview

This practice covers using various hints to influence execution plans.