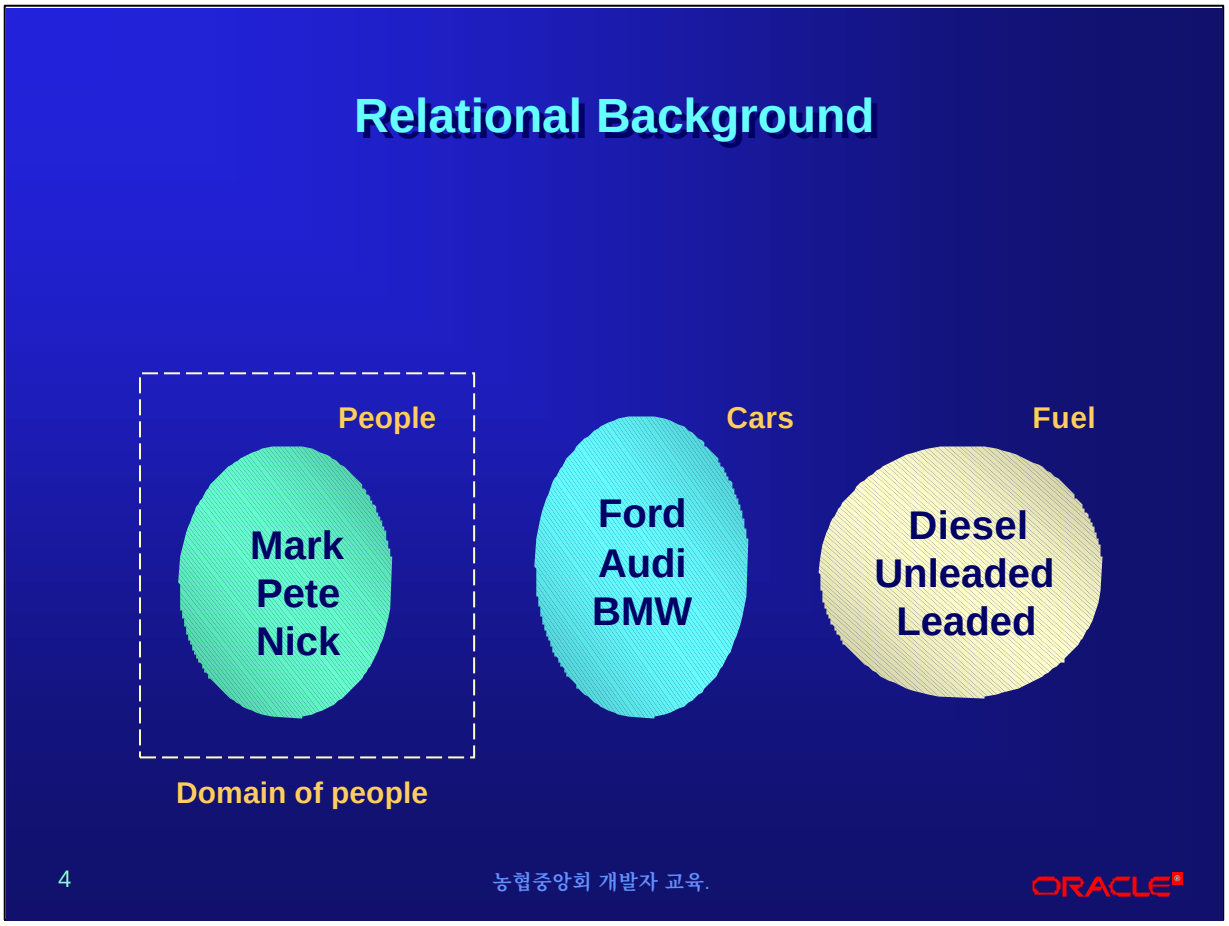
개발자를 위한 Oracle Optimizer 원리 이해 및 SQL Tuning (Oracle 9i)

DB Tech. Team
한국오라클㈜
2004.12

목 차

1

Oracle Relational Concepts



Relational Background

Each ellipse represents a set of objects of a certain type; people, fuel, and cars. In data modelling terms you would call each set an *entity*.

Each set contains three elements. This is the *cardinality* of the set; cardinality is the number of elements of a set.

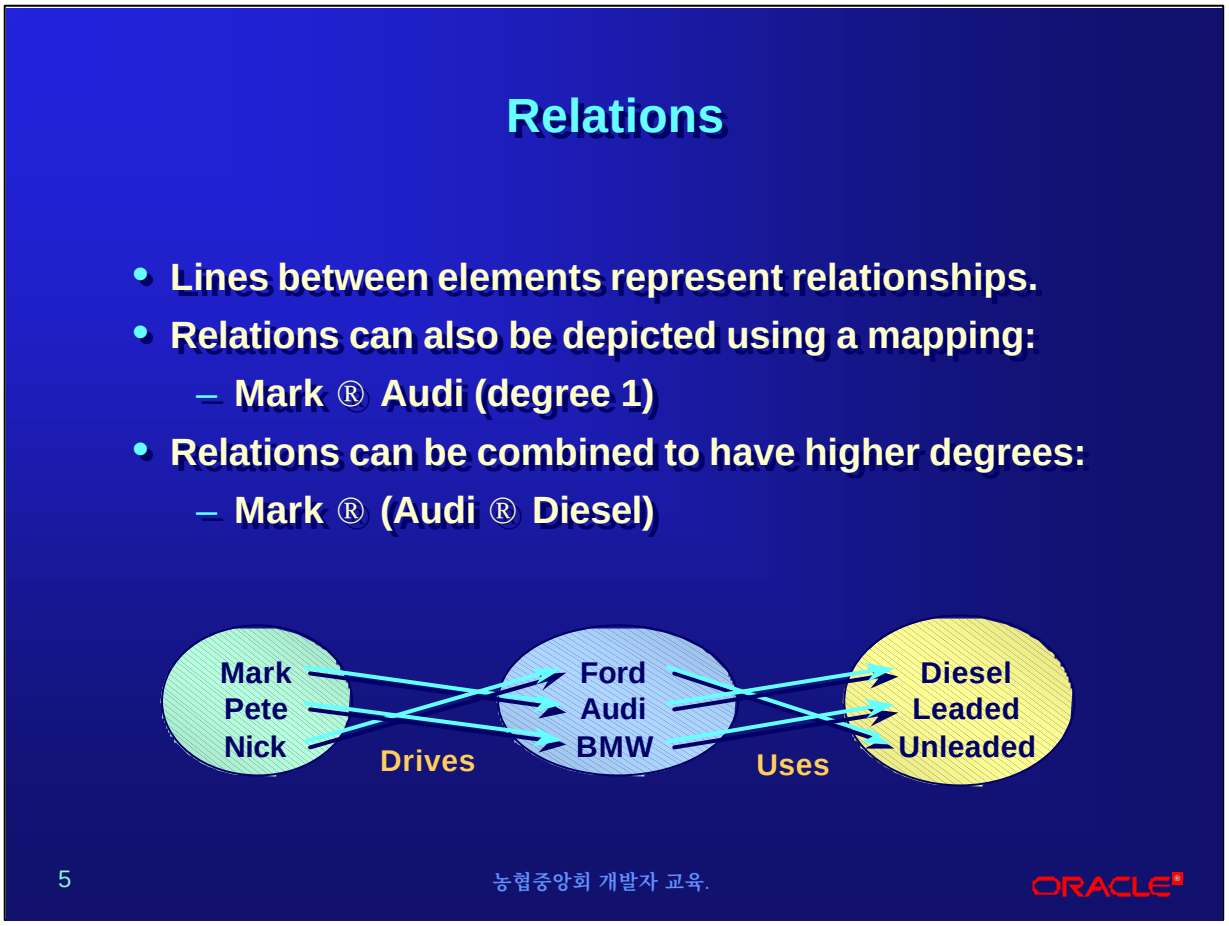
Each set is a subset of a larger domain; the domain is the set from which another set draws its values.

Relational databases do not really represent domains fully. What they do have is a limited set of data types. A domain can be thought of as a very restrictive data type. For example, data in a character column has to meet the type characteristics of the column. Similarly, the people set has to be within the domain of all people.

Domains have implications for joins. The Oracle server allows ENAME to be joined to DNAME, but in terms of domains this makes no sense as employee names are joined to department names. However, the Oracle server allows the join to proceed as long as the data types match (or can be converted.)

Note: The relational model is concerned with the relationships between sets of data. The diagram shows three unrelated sets of data.

There are relationships between the data sets that are not shown.



Relations

There is a relationship between the people and cars that can be shown as follows:

Relation: DRIVES

Mark → Audi

Pete → BMW

Nick → Ford

The relation DRIVES shows a relationship between two sets of elements.

Relation: USES

Ford → Unleaded

Audi → Diesel

BMW → Leaded

The relation USES again shows a relationship between two sets of elements. In this representation a relation is known as a *maplet*.

A relationship can be illustrated in a number of ways. So far a diagrammatic and a maplet representation has been presented. Within Oracle Designer these relations are referred to as *one-to-many* entities.

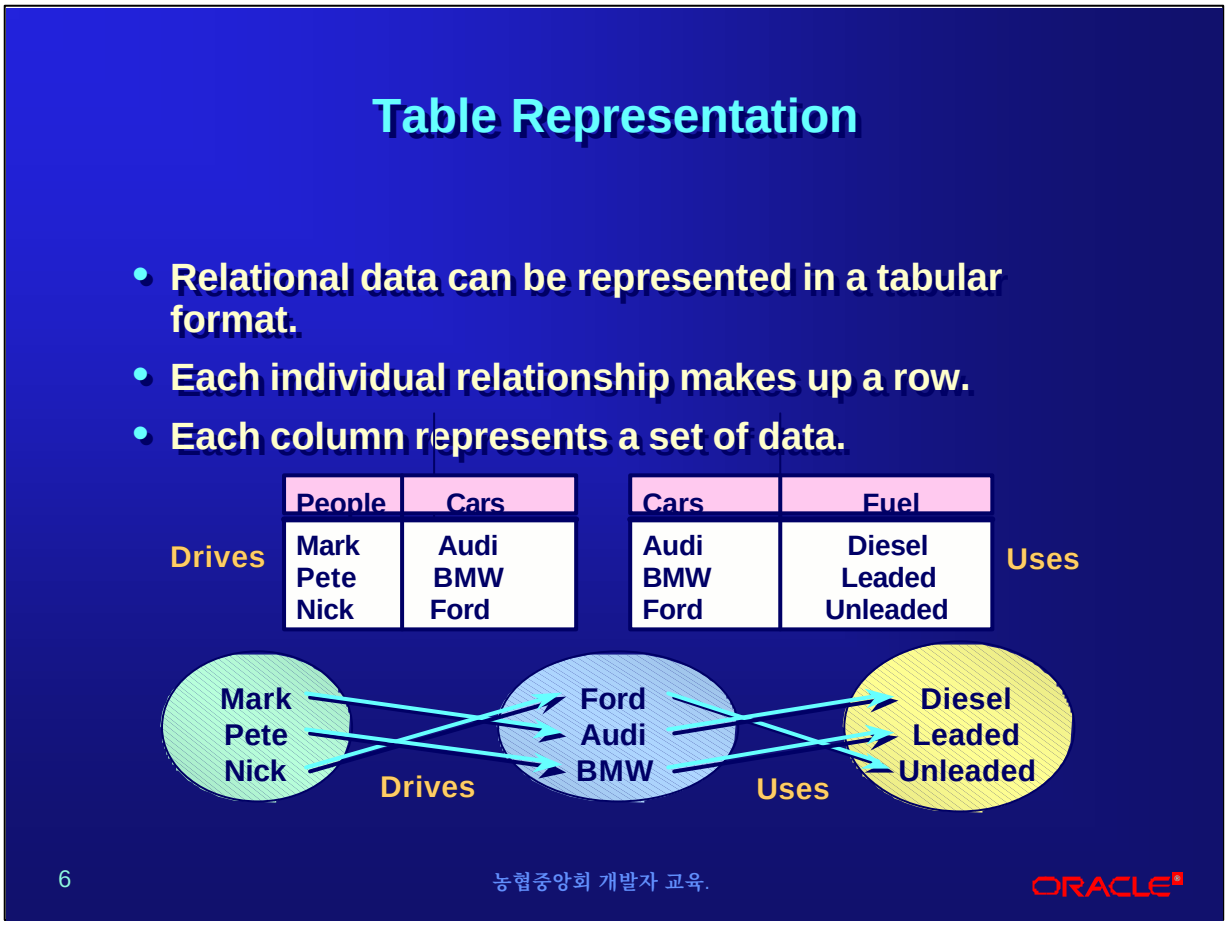


Table Representation

Information can be stored in tabular format:

- A *table* is a representation of a relation.
- A table consists of a number of *rows* or *tuples*.
- A row is a representation of a tuple.

The number of columns in a table minus one represents the *degree* of the relation. In the above tables the degree is 1.

Note: The entire mapping diagram above could be represented as a single table as below. This table has degree 2. In relational terms the move from the two tables DRIVES and USES to a single table is known as *denormalization*.

People	*Cars*	*Fuel*
Mark	**Audi**	**Diesel**
Pete	**BMW**	**Leaded**
Nick	**Ford**	**Unleaded**

Example

- How to find out who uses diesel?

```
SQL> select d.people
  2  from   drives d, uses u
  3  where  d.cars = u.cars
  4  and    u.fuel = 'Diesel';
```

- The relational model says:
  - Take the Cartesian product: DRIVES X USES
  - Apply predicates to eliminate unwanted rows.
  - Remove unwanted columns.

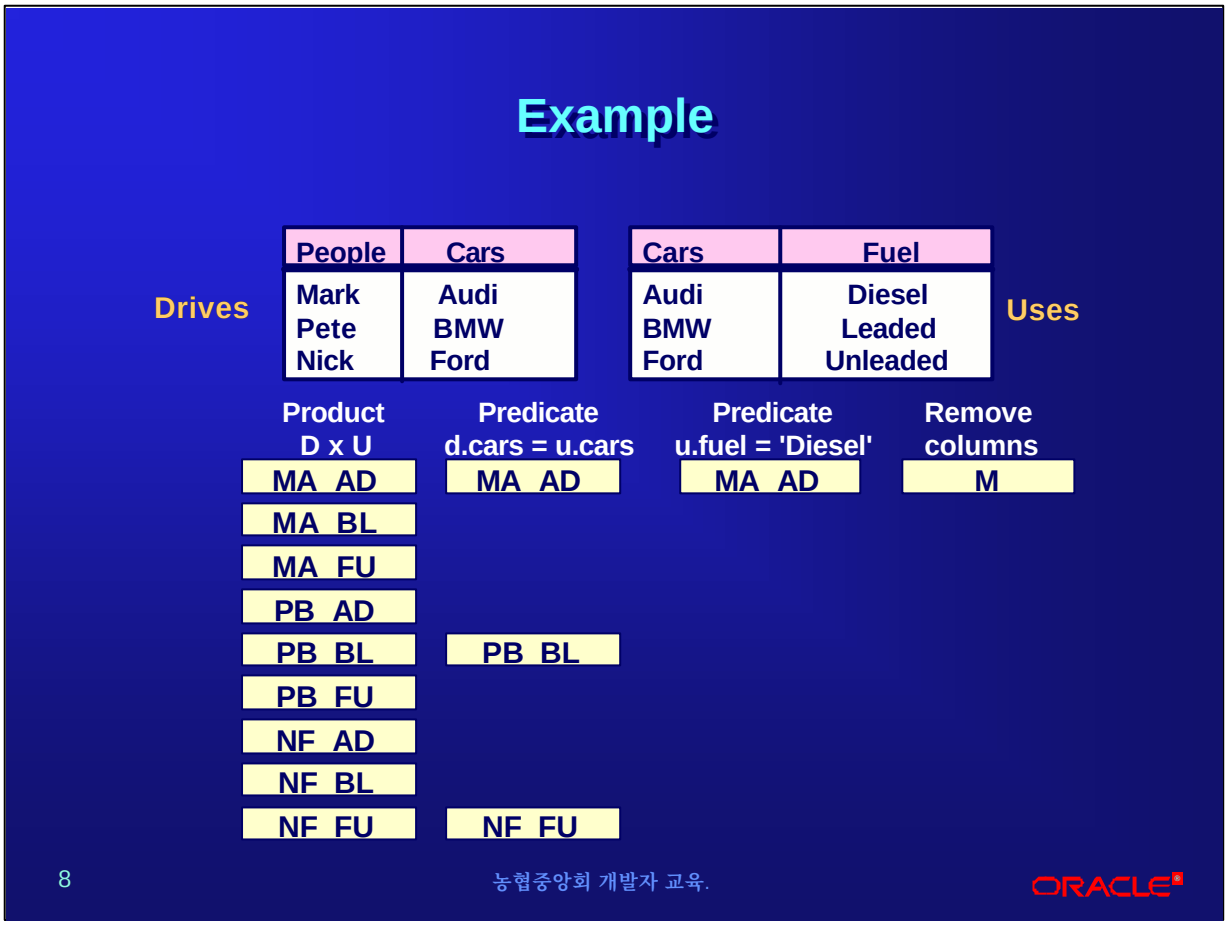
Example

This example illustrates how the relational model combines multiple relations to get information from each relation. This is very similar to table processing within Oracle.

You have two relations, DRIVES and USES, which have the required information.

You can represent your request for information on “Who uses diesel?” by using a SELECT statement.

A Cartesian product joins every row in a table to every row in the other table. Rows and columns are filtered out to give the desired result set. This is shown on the following slide.



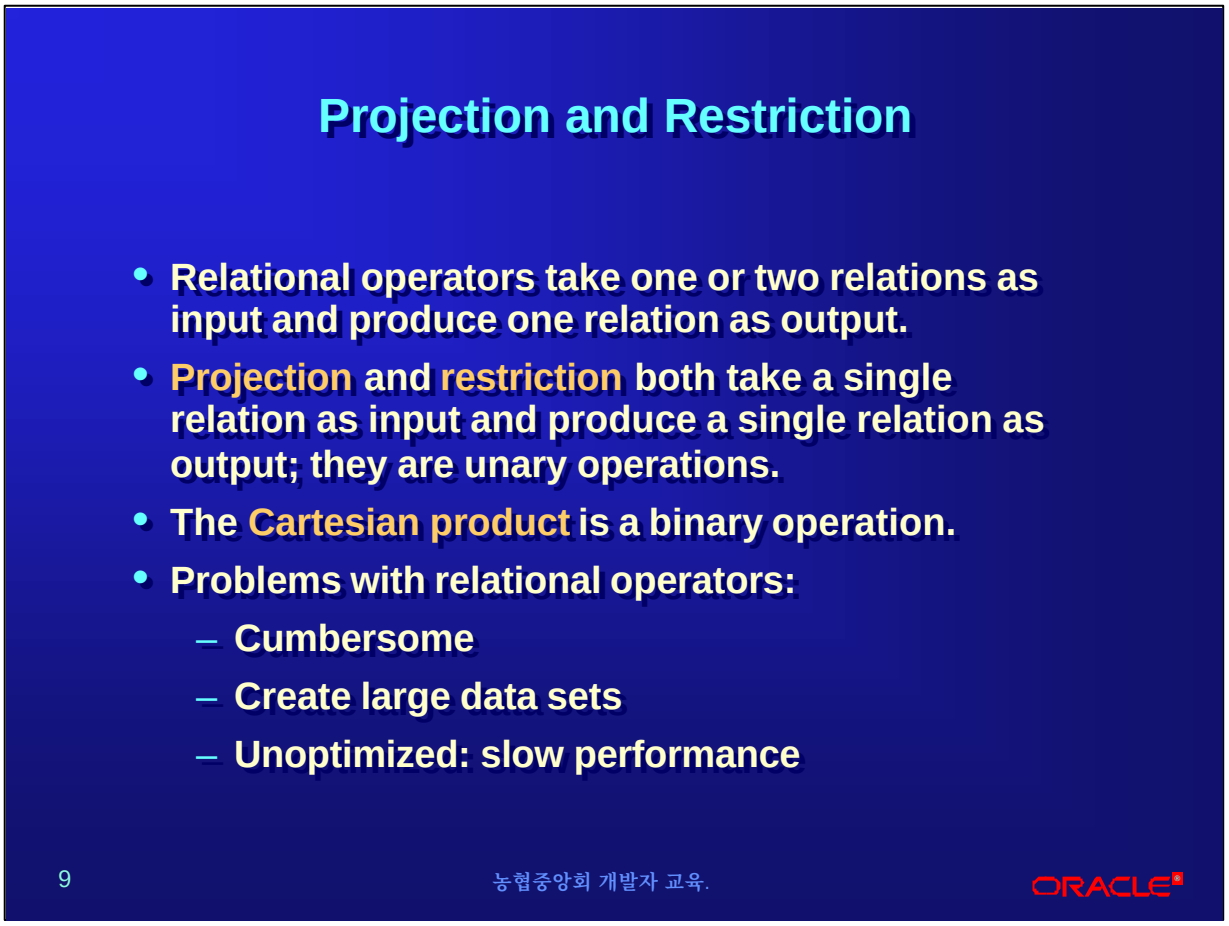
Example (continued)

Step 1 is the (Cartesian) product; this joins every row in DRIVES with every row in USES. The Cartesian product is depicted using the X character.

Step 2 applies the two WHERE clause predicates to remove rows that are not required. Removing rows is known as *restriction*.

Step 3 removes columns from the result set to leave the column in which we are interested. Removing columns is known as *projection*.

Note: To save space in the slide the combinations have been reduced to their initials; for example, "NF FU" stands for "Nick Ford Ford Unleaded."



Projection and Restriction

In relational documentation:

- Restriction is often shown as σ (sigma)
- Projection is shown as Π (pi)
- Cartesian product is shown as X
- Join is shown as |X|

A join is a merge of a Cartesian product and a restriction and is a performance enhancement.

Cartesian product, restriction and projection are valid means of extracting data from a relational database, but are long, often cumbersome and unoptimized procedures. A Cartesian product quickly becomes unmanageable with large numbers of tuples within the relations. Because these steps are independent of the other, there is the potential to miss important optimizations. The same resultant data set can be achieved more efficiently. Oracle achieves this using joins and other methods.

This is the reason why the joining technology implemented in Oracle is discussed in this first lesson.

Join Implementation

- A join is a method of implementing a Cartesian product and a restriction as a single operation.
- Joins are a performance enhancement.
- Join types:
  - Basic (natural) join
  - Outer join
  - Semijoin
  - Antijoin

10 농협중앙회 개발자 교육. ORACLE

Join Implementation

Oracle joins can be executed in parallel (spread over multiple query slaves) or in serial. However, only two row sources can be joined in a single operation.

Example:

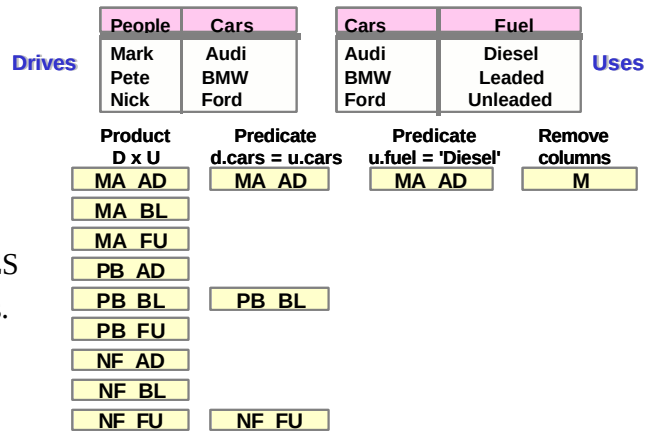
```
SQL> select d.people
  2  from   drives d, uses u
  3  where  d.cars = u.cars
  4  and    u.fuel = 'Diesel';
```

The relational model says:

Take the Cartesian product: DRIVES X USES

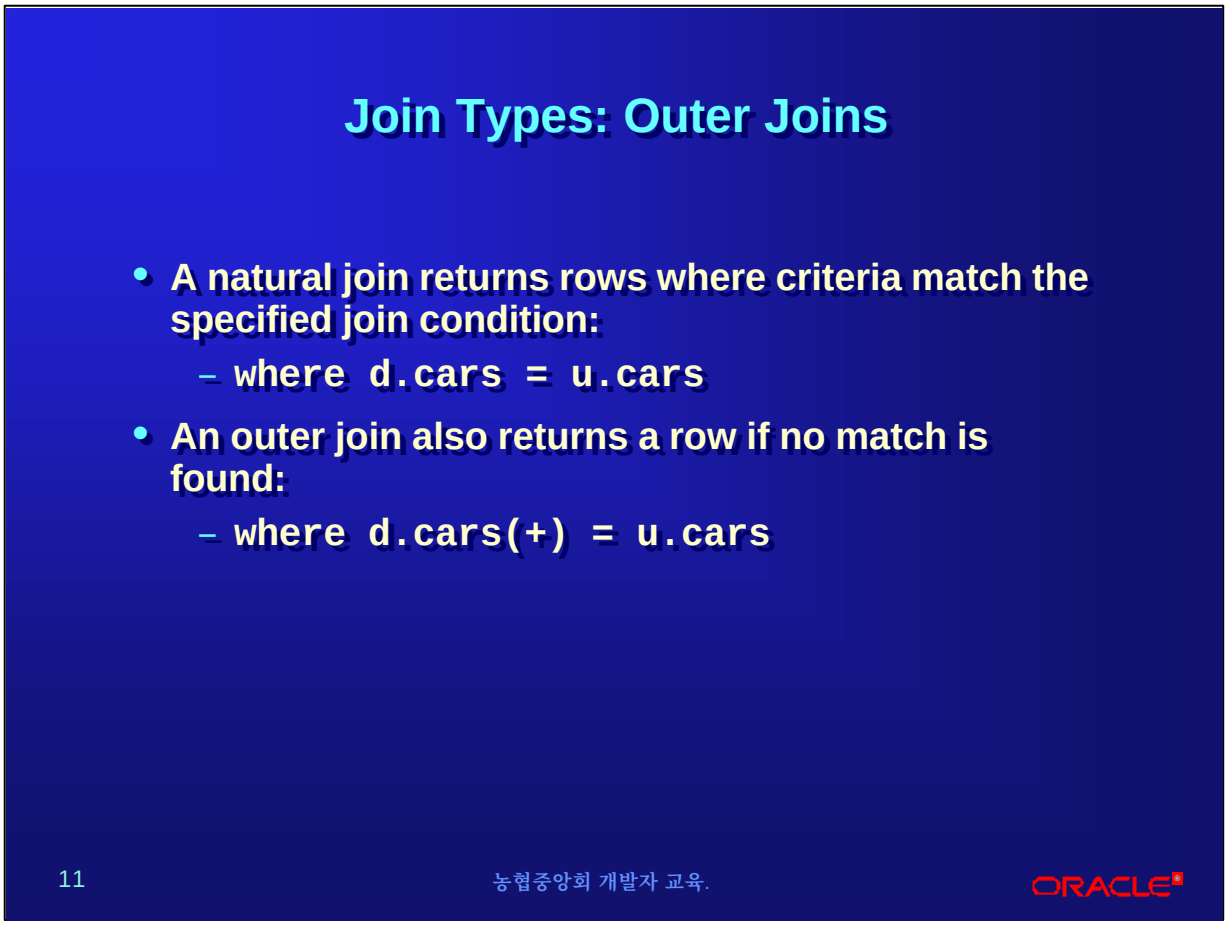
Apply predicates to eliminate unwanted rows. (Restriction)

Remove unwanted columns. (Projection)



Row Sources

In Oracle, the concept of a row source is often used. A row source is a set of data that can be accessed in a query. It can be a table, an index, or even the result set of a join tree consisting of many different objects.



Join Types: Natural joins

A natural join is also called a simple join or a equijoin.

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.

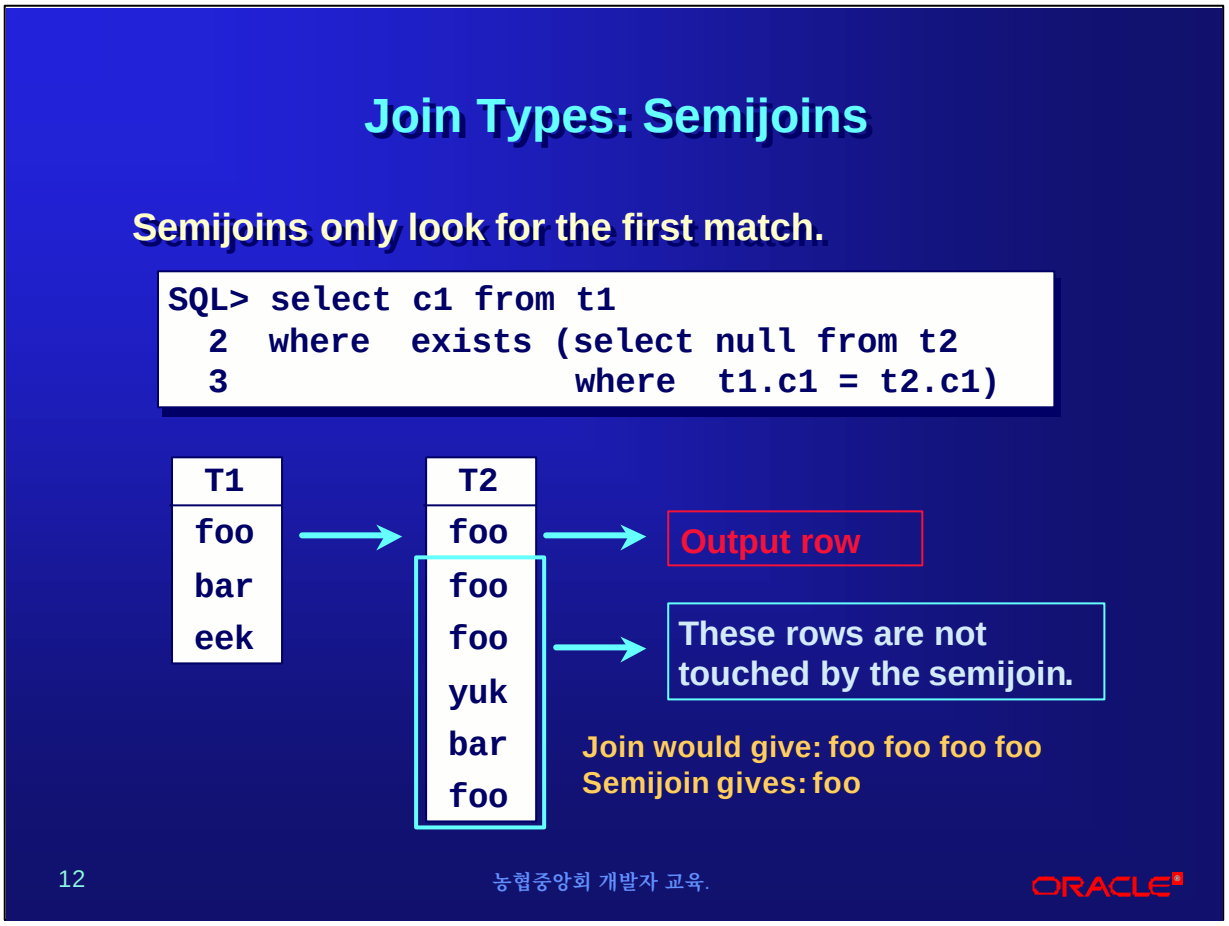
The outer join operator is placed on the deficient side of the query. In other words, it is placed against the table that has the missing join information.

Outer Join Example

Consider EMPLOYEES and COURSES. There may be an employee that has not developed any course. If EMPLOYEES and COURSES are joined together this particular employee would not appear in the output because there is no join for that employee. By using the outer join the missing employee can be displayed.

Notice that the outer join operator (+) is placed on the table that is missing the information.

```
SQL> select e.last_name, c.short_name
  2  from   courses c, employees e
  3  where  c.dev_id(+) = e.emp_id;
```



Join Types: Semijoins

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.

Semijoins are useful for EXISTS subqueries which are transformed otherwise (see later in the course).

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.

The diagram shows only the c1 column.

Before this join type was implemented in Release 8.0.4, the Oracle server transformed EXISTS subqueries so that they could be processed using standard join techniques.

Note: On earlier releases a similar result could be achieved by using a FILTER step and scanning a row source until a match was found, then returning. Often variable numbers of blocks would be read dependent on the position of the constraining rows in the row source. Semijoins were not actually implemented as a real join until Oracle8. Implementation as a join gives more flexibility to push predicates.

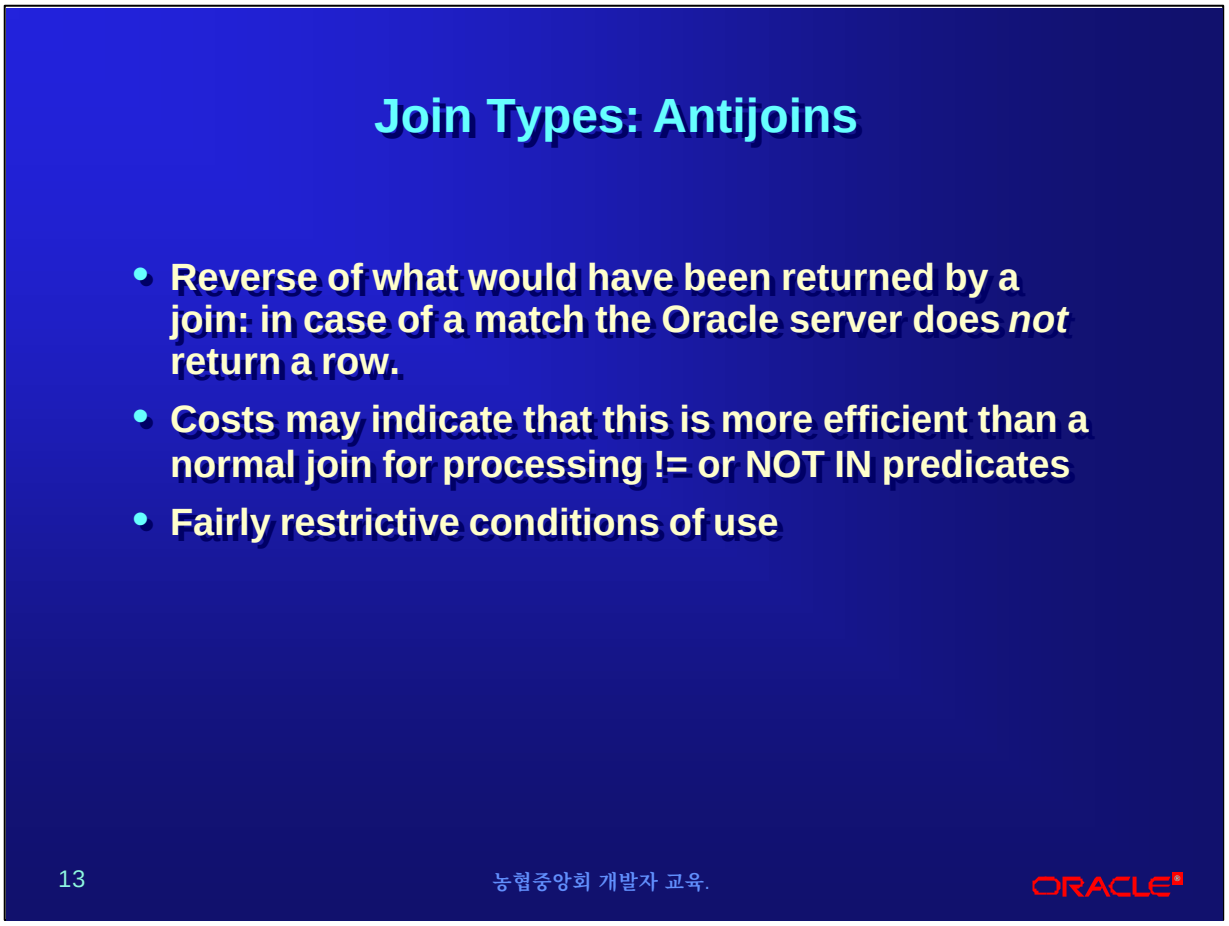
-- EXIST Subquery가 Semijoin으로 풀리기위한 조건

-- 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.



Join Types: Antijoins

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.

Join Type Choice

While the join types may have some limitations of usage, the cost-based optimizer (CBO) chooses an appropriate join type based on calculated costs. This consideration is mostly transparent to the end-user, but can be influenced; see later lesson on hints.

```
-- 'NOT IN' SubQuery의 ANTIJOIN 또는 FILTER 처리
--
-- 1. Subquery에서 참조되는 모든 Column이 not Null이어야 한다.
-- 2. Subquery가 Not Correlated이어야 한다.
-- 3. Outer query block에 어떠한 'OR'조건도 없어야 한다.

-- session level의 always_anti_join변경은 8.1.7이후 부터 가능
-- Analyze 정보가 없어도 Hint로 강제적으로 Anti Join Type을 정할 수 있다.
select * from dept
where deptno IS NOT NULL
and deptno not in
(select /*+ HASH_AJ */ deptno from emp
  where deptno IS NOT NULL and ename = 'SCOTT');
Execution Plan
----------------------------------------------------------
  0     SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=19 Bytes=950)
  1   0   HASH JOIN (ANTI) (Cost=5 Card=19 Bytes=950)
  2   1     TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=20 Bytes=600)
  3   1     TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=1 Bytes=20)
```

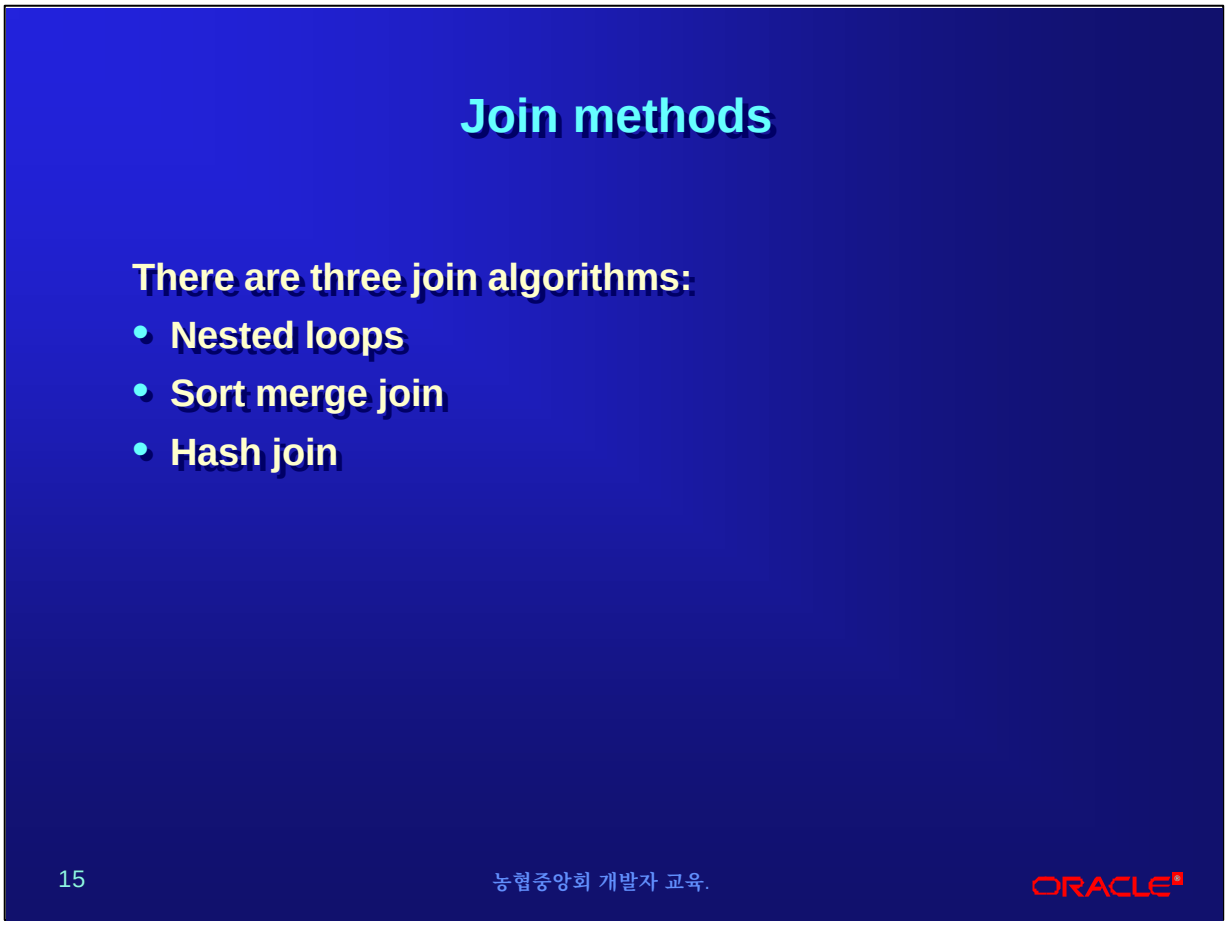
Cartesian Product

Notice that there is no join predicate in the following statement:

```
SQL> explain plan for
  2  select emp.deptno, dept.deptno
  3  from   emp, dept
Query Plan
--------------------------------
SELECT STATEMENT [CHOOSE] Cost=5
  MERGE JOIN CARTESIAN
    TABLE ACCESS FULL DEPT
    SORT JOIN
      TABLE ACCESS FULL EMP
```

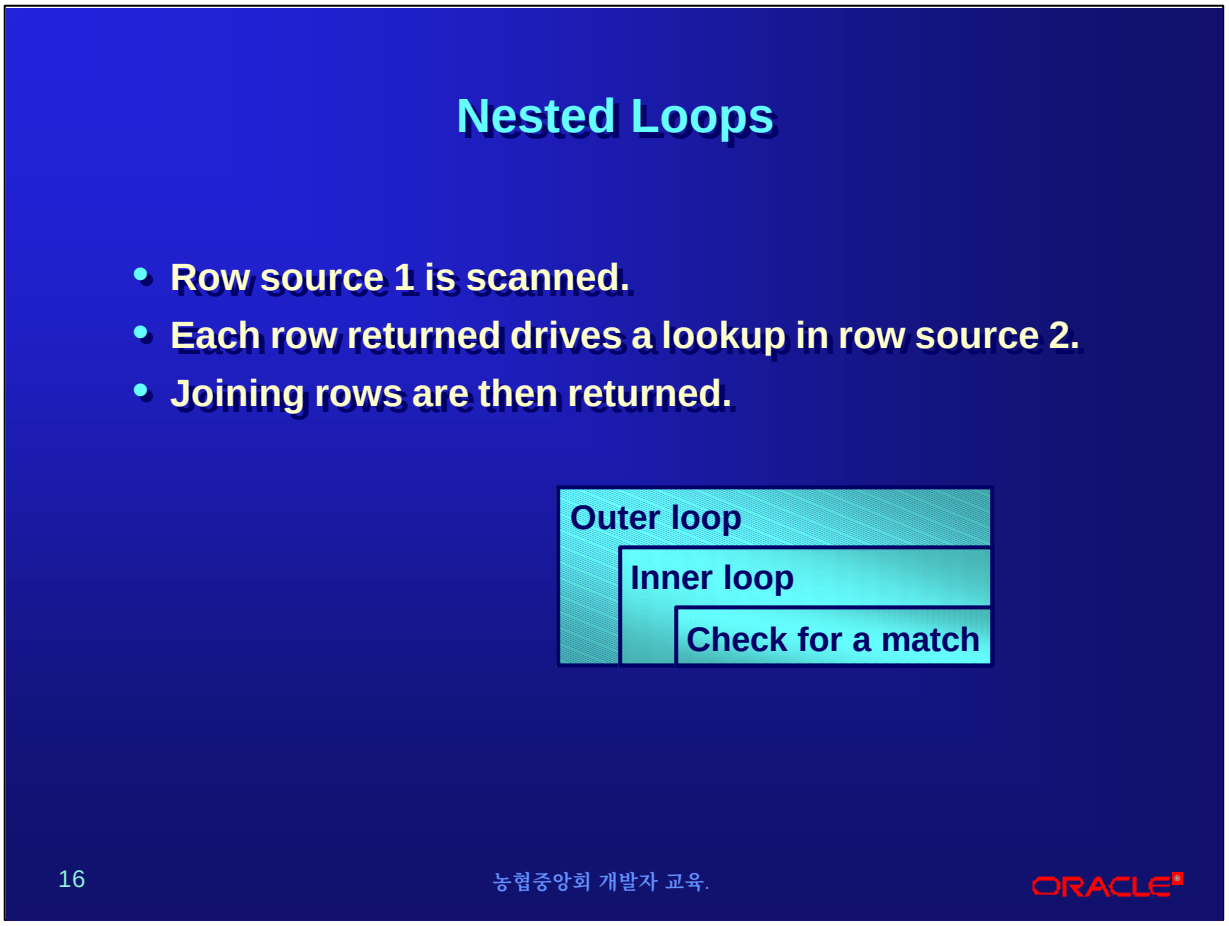
The CARTESIAN keyword indicates that you are doing a Cartesian product.

A star join is not a join as such. It is implemented using a nested loops join with a specific schema layout. Small lookup or fact tables are combined using a Cartesian product. Then a large lookup table is probed using a concatenated index. The Cartesian product is the driving table for the nested loops join. This access method can give performance enhancements with certain data warehouse configurations. Further details are beyond the scope of this lesson.



Join Methods

Once you have your join definitions you need methods for implementing the joins. There are currently three join algorithms available to the Oracle server. There are actually no other valid join methods anyway.



Nested Loops

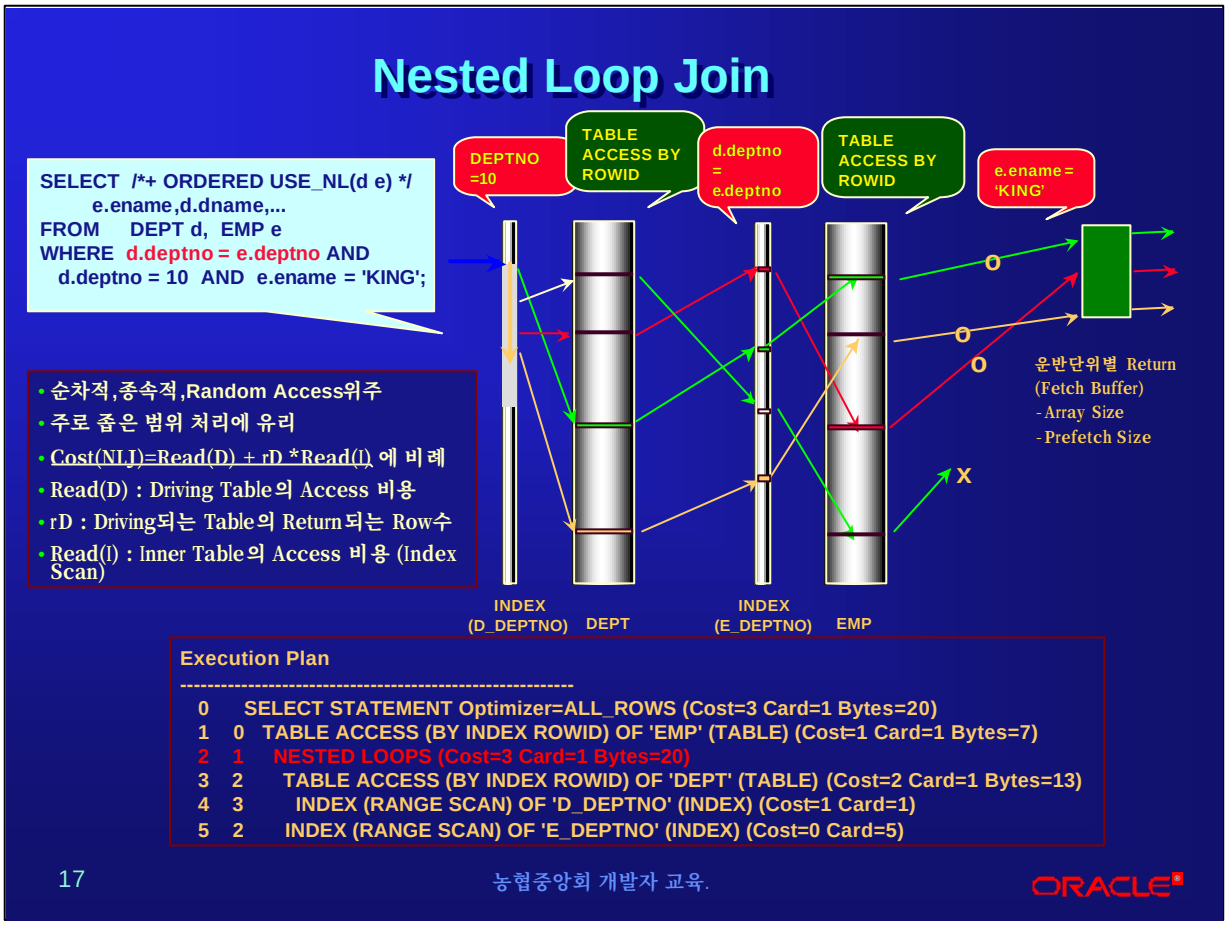
Row source 1 is known as the outer or driving table; row source 2 is known as the inner table. Accessing row source 2 is known as probing the inner table.

The Oracle server fetches a set of rows from the first row source and then probes the second row source using each row from the first row source. Once the set of rows from the first row source is exhausted a new set is fetched until row source 1 is exhausted. The Oracle server does not have to retrieve everything from the first row source before accessing the second row source.

For efficient nested loops it is important that row source 1 returns as few rows as possible, as this directly controls the number of probes of row source 2. Also, it helps if the access method for row source 2 is efficient as this operation is being repeated numerous times. In other words, an index on row source 2 improves the join efficiency.

```
SQL> explain plan for
  2  select /*+ ordered use_nl(c) */
  3         e.last_name, c.short_name
  4  from   employees e, courses c
  5  where  c.dev_id = e.emp_id;
Query Plan
----------------------------------------
SELECT STATEMENT [CHOOSE] Cost=45413
  NESTED LOOPS
    TABLE ACCESS FULL EMPLOYEES [ANALYZED]
    TABLE ACCESS FULL COURSES [ANALYZED]
```

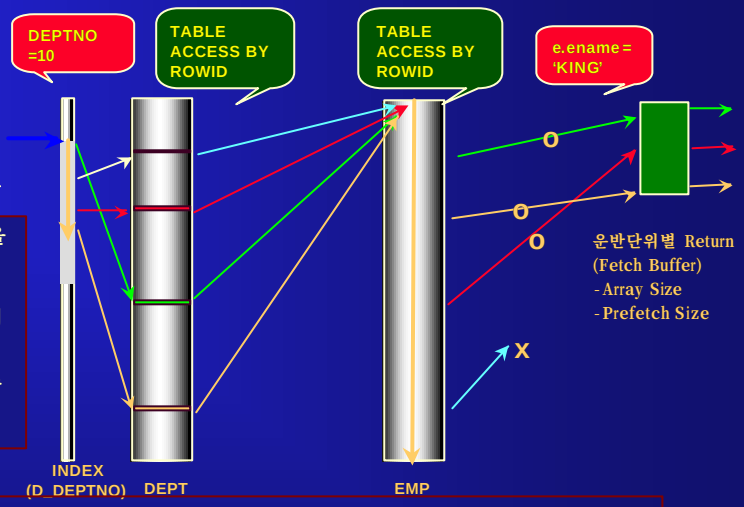
Nested loops efficiency depends on the number of rows in the driving table and the speed of access into the inner table.



1. NLJ는 순차적인 처리로 Fetch의 운반단위(ArraySize,PrefetchSize)마다 결과 Row를 Return받을 수 있다.
2. 첫번째 Row를 받는 시간은 빠르나, 전체 결과 Row를 받는데 까지 걸리는 시간은 느리다. 즉 첫번째 Row를 받을 준비가 되어 있는 단계까지를 실행시간으로 볼 때 실행시간은 빠르나 Fetch시간은 느리다.
3. NLJ는 Memory가 필요 없는 Join이다. 그러므로 추가적인 Memory비용이 필요 없다.
4. NLJ은 Driving Table에서 많은 Row들이 Filtering되어 Inner Table로 찾아 들어가는 부분을 줄여야 하므로 Driving순서가 중요하다.
5. Inner Table은 Driving Table의 Return되는 모든 Row들에 대해서 반복 실행하므로 Access의 효율이 좋아야 한다. 즉 대부분의 경우 Inner Table은 Index가 있어야 한다. 또한 Index의 효율이 좋아야 한다. Inner Table이 작더라도 Access회수가 높다면 Index가 있어야 한다. Index의 효율이 좋지 않아 전체의 Index Range Scan과 같은 경우는 최악의 조건이다.
6. NLJ는 주로 Index위주의 Single Block I/O의 Random I/O위주 이므로 OLTP에서 적은 Data범위 처리에 주로 사용된다. 즉 전체의 15%이상의 경우는 Full Table Scan을 이용한 Sort Merge또는 Hash Join을 이용한다.
7. NLJ도 Driving Table이 Full Table Scan에 Parallel로 처리되면 Inner Table도 Parallel로 종속적으로 처리된다.

Nested Loop Join

```
SELECT  /*+ ORDERED USE_NL(d e) */
        e.ename,d.dname,...
FROM    DEPT d,  EMP e
WHERE  d.deptno = e.deptno AND
   d.deptno = 10  AND  e.ename = 'KING';
```



- Inner Table에 join Key에 대한 index가 없을 경우 Return되는 Row마다 Full Table Scan발생
- Cost(NLJ)=Read(D) + rD * Read(I) 에 비례
- Read(D) : Driving Table의 Access 비용
- rD : Driving되는 Table의 Return되는 Row수
- Read(I) : Inner Table의 Access 비용(Full Table Scan비용)

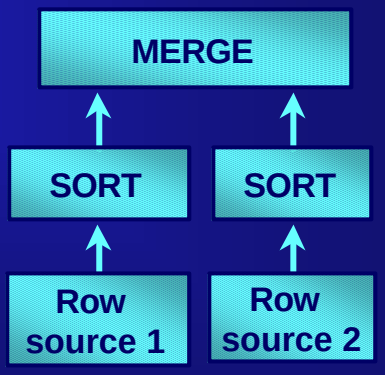
Execution Plan

```
----------------------------------------------------------
  0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=5 Card=1 Bytes=20)
  1   0  NESTED LOOPS (Cost=5 Card=1 Bytes=20)
  2   1    TABLE ACCESS (BY INDEX ROWID) OF 'DEPT' (TABLE) (Cost=2 Card=1 Bytes=13)
  3   2      INDEX (RANGE SCAN) OF 'D_DEPTNO' (INDEX) (Cost=1 Card=1)
  4   1    TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=3 Card=1 Bytes=7)
```

Driving되는 Table의 Return되는 Row수 만큼 반복되므로 Full Table 이 여러 번 반복되면 비효율적이다.

Sort Merge Join

- Rows from row source 1 are sorted.
- Rows from row source 2 are then sorted by the same sort key.
- Row source 1 and 2 are not accessed concurrently.
- Sorted rows from both sides are then merged.



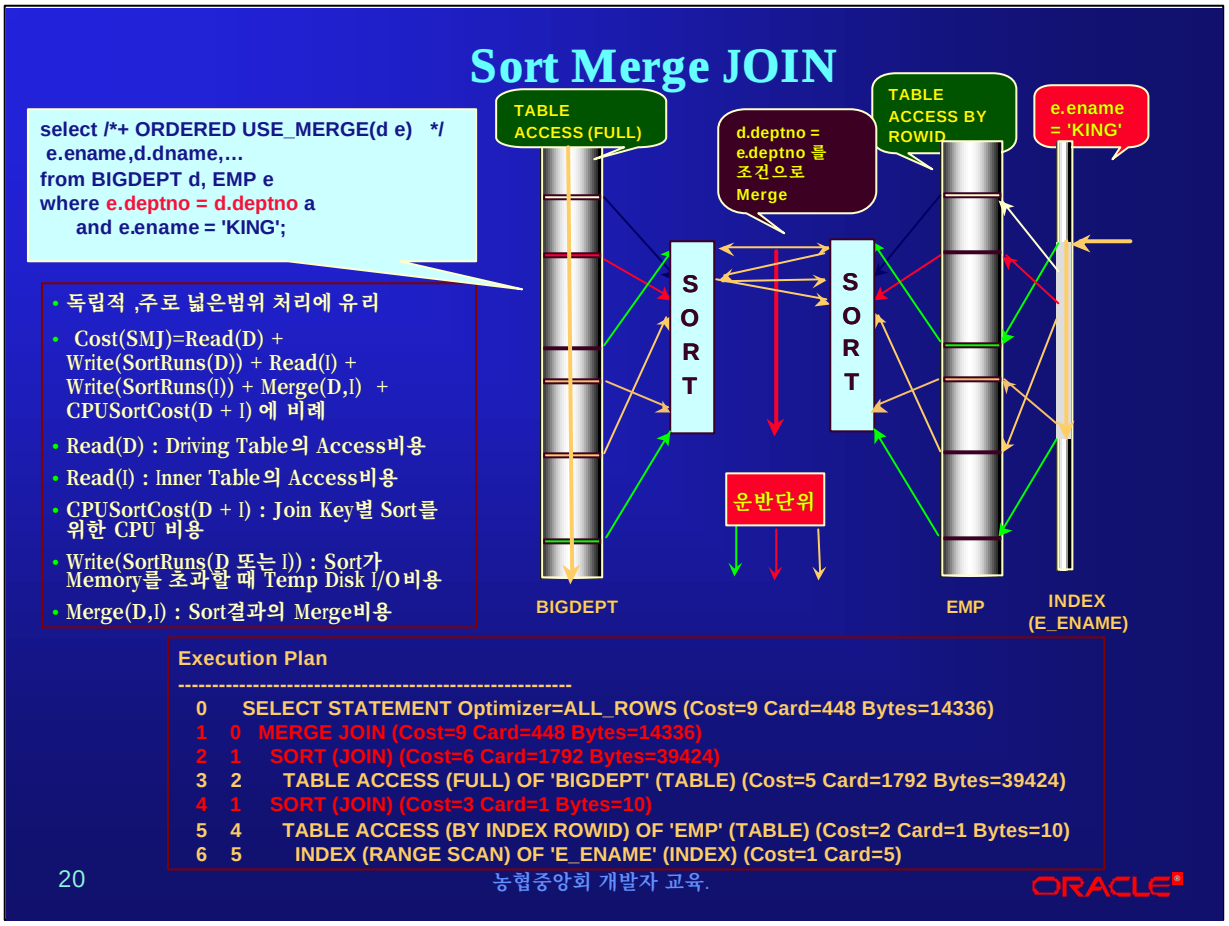
Sort Merge Joins

If the row sources are already (known to be) sorted then the sort operation is unnecessary as long as the sort key is able to be used to merge the two row sources. Presorted row sources include indexed columns and row sources that have already been sorted in earlier steps.

The merge of the two row sources is handled serially but the row sources could be accessed in parallel.

```
SQL> explain plan for
  2  select e.last_name, c.short_name
  3  from   courses c, employees e
  4  where  c.dev_id = e.emp_id;

Query Plan
-------------------------------------
SELECT STATEMENT [CHOOSE] Cost=149
  MERGE JOIN
    SORT JOIN
      TABLE ACCESS FULL COURSES [ANALYZED]
    SORT JOIN
      TABLE ACCESS FULL EMPLOYEES [ANALYZED]
```

The main advantage of a merge join over a Cartesian product is the data set worked on is much smaller. The potential problem is the inefficiency of the sort operation.

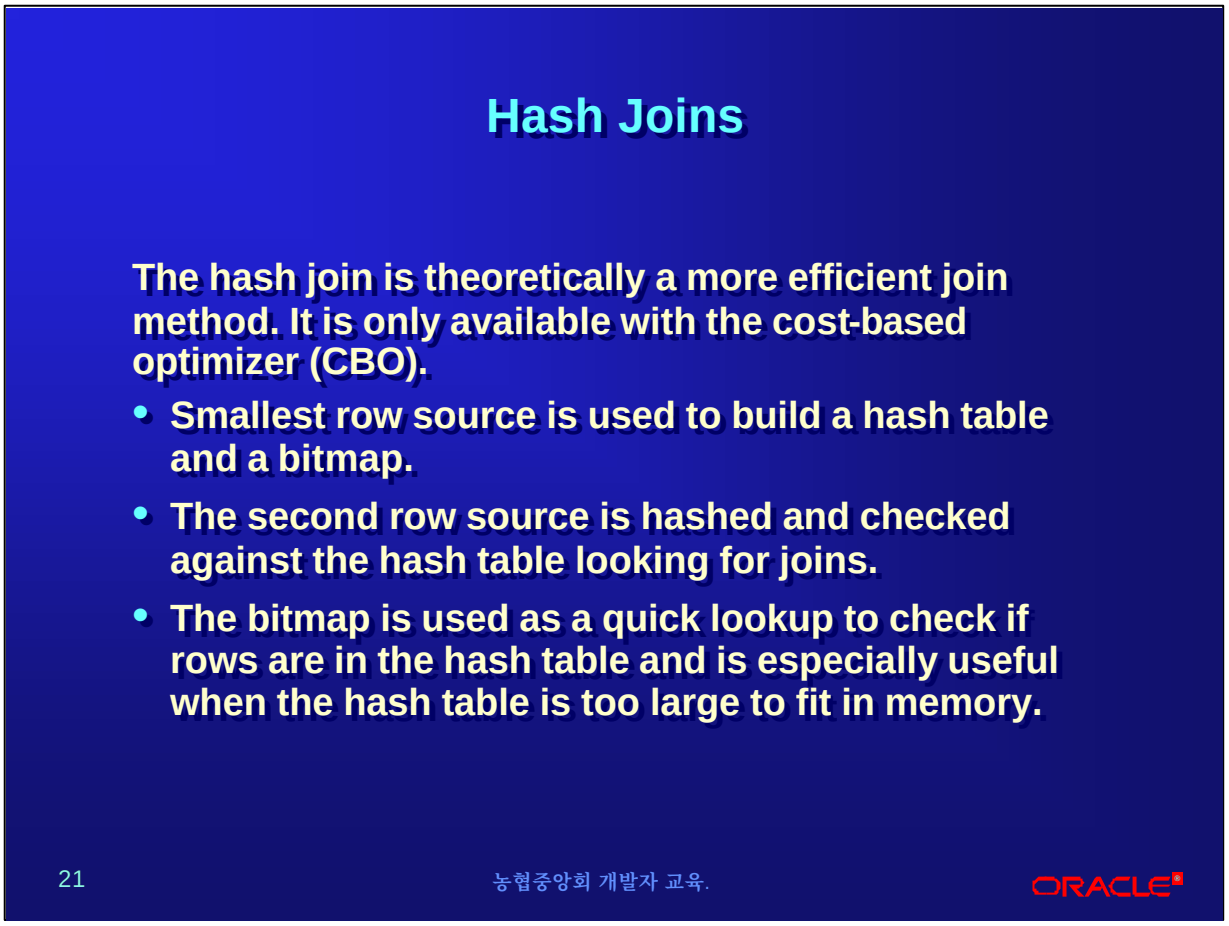


1. 전체 Row를 Return받는 시간이 빠르다. 즉 첫번째 Row를 Return받을 준비까지의 시간은 느리나, 준비가 된 상태에서의 Fetch시간은 빠르다. (Memory에서 Return하므로) 이는 대상 Row(Where조건에 의해 Filter된 Rowe들만 Sorting)들을 가지고 Sort Operation(모든 Rows들을 Join Key로 Sorting)하기 전까지는 어떠한 Row들도 Return할 수 없기 때문이다.
2. NLJ과 같이 Driving Table의 Return되는 Row수와 Inner Table의 Access Pattern에 의에 Access 의 효율이 좌우되지 않으며, Join Table간의 자신의 처리범위로만 처리량을 결정하므로 독립적이다.
3. 추가적인 Sort Memory (SORT_AREA_SIZE)의 비용이 필요하다. Memory가 부족하면 TEMP Tablespace에 Sorting중간단계(Sort Runs)를 기록하게 되므로 추가적인 Disk I/O비용이 발생할 수 있다.
4. Sort대상은 Join key뿐만 아니라 Select List도 포함하므로 불필요한 Select List는 제거해야 한다.
5. Sort의 CPU사용에 대한 Overhead가 있다. 그러므로 많은 Row들과 전체적으로 Select List의 Size의 합이 큰 Table의 Join에는 문제가 있다. 즉 Disk Sort를 피할 수가 없으며, Sort의 CPU 비용이 많이 든다.
6. Disk Sort만 발생하지 않는다면 넓은 범위 처리에 유리하다.
7. Disk Sort를 피할 수 없는 경우라면 SORT_AREA_SIZE , SORT_MULTIBLOCK_READ_COUNT를 SQL마다 Session Level에 할당해서 사용하도록 한다(WORKAREA_SIZE_POLICY가 MANUAL일 경우이거나 9i 이전 Version에서). 또한 TEMP Tablespace의 Extent Size도 충분히 크게 주도록 한다.

   ALTER SESSION SET SORT_AREA_SIZE= 104857600;

   ALTER SESSION SET SORT_MULTIBLOCK_READ_COUNT=128;
8. Sort Memory의 Size는 (= Target rows * (total selected column's bytes) * 2) 이상 설정하되 PGA의 Memory의 한계로 인해 TEST를 통해 PGA Memory Allocation Error가 발생하지 않는 범위 내에서 설정하도록 한다. 필요시 10032 Trace를 이용해 점검한다.

   ALTER SESSION SET EVENTS '10032 TRACE NAME CONTEXT FOREVER;



Hash Joins

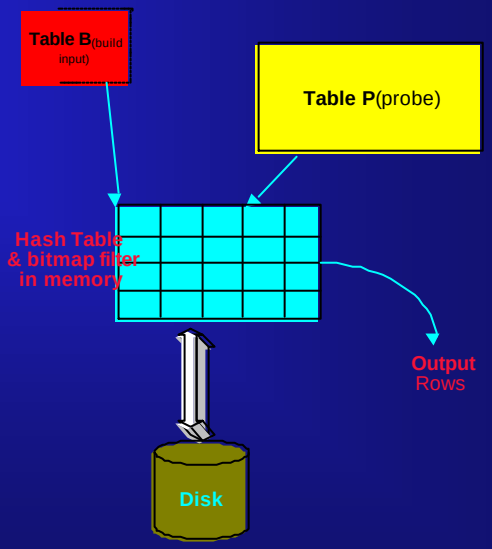
```
SQL> explain plan for
  2  select /*+ use_hash (e) */
  3         e.last_name, c.short_name
  4  from   courses c, employees e
  5  where  c.dev_id = e.emp_id;
Query Plan
----------------------------------
SELECT STATEMENT  [CHOOSE] Cost=3
  HASH JOIN
    TABLE ACCESS FULL COURSES
    TABLE ACCESS FULL EMPLOYEES
```

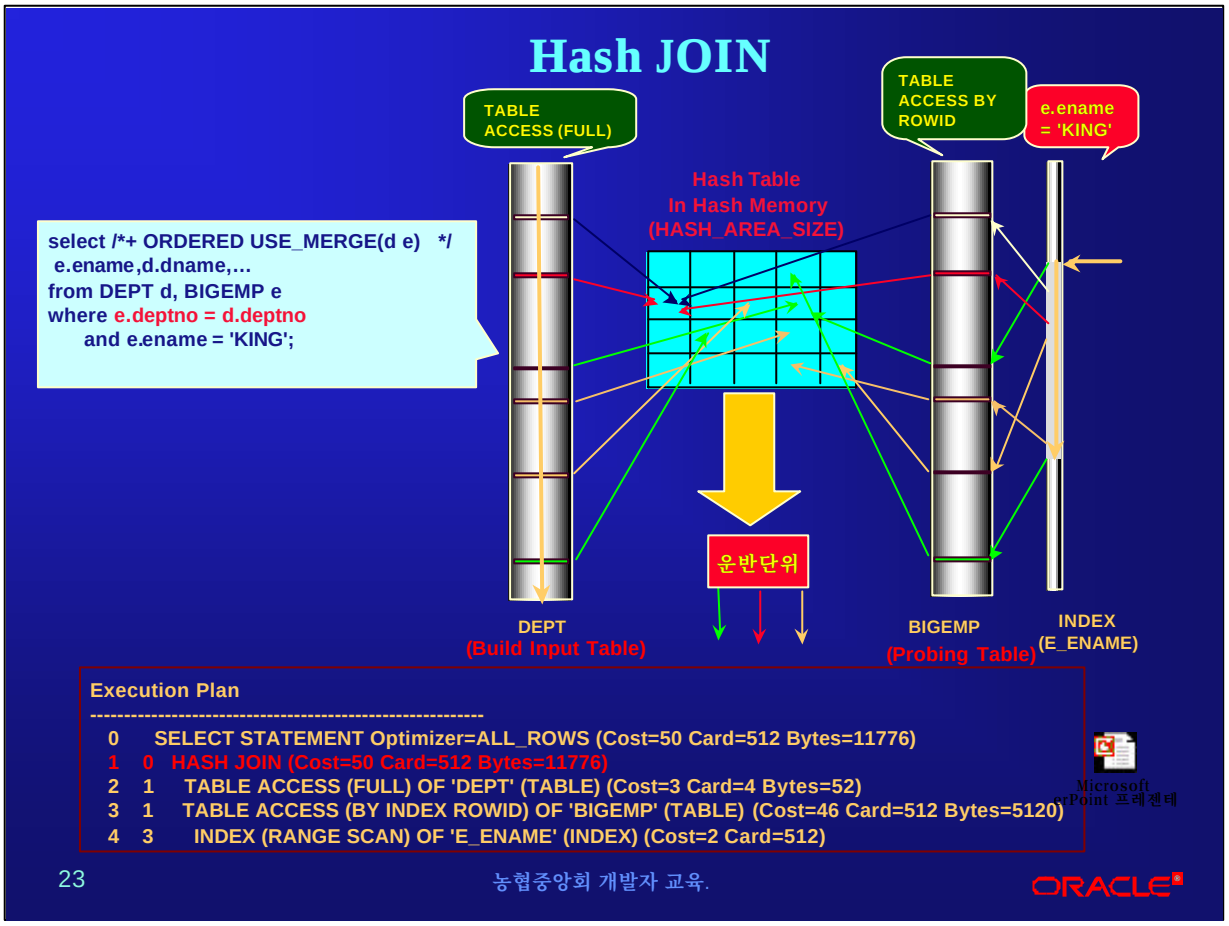
HASH_JOIN_ENABLED=TRUE is the default since Oracle Release 7.3.

Hash joins tend to be efficient because they only involve a single pass of each row source and the hashing algorithm and lookup is more efficient than sorting and merging.

Hash JOIN

- 어느정도 순차적, 독립적
- Driving Table(Build Input) Table로 Hash Memory에 Hash Table을 만들고 Proving Table로 찾는다. (순서 주의)
- Hash Table은 Where조건에 의해 Filter된 Row들만 대상 → ROWSET
- 적은 Rowset 와 큰 Rowset의 처리에 최적 (대량의 Data처리에 최적)
- Cost(HJ) = Read(B) + Build Hash Table in Memory (cpu) + Read(P) + Perform In memory Join(cpu) 비례
- Read(B), Read(P) : Build,Probe Table의 Access비용
- Build Hash Table in Memory (cpu) : Hash table 구성 비용
- Perform In memory Join(cpu) : Probing 비용





1. Hash Join은 두개의 Join Table중 Small Rowset(Where조건에 의해 Filtering된 Row수가 작은 Table)을 가지고 HASH_AREA_SIZE에 지정된 Memory내에 Hash Table을 만든다.
2. Hash Table을 만든 이후 부터는 NLJ의 장점인 순차적인 처리 형태이다. 그러므로 NLJ과 SMJ의 장점을 가지고 있다.
3. Hash Join은 Basic Join('=')만 가능하다.
4. NLJ과 같이 Driving Table의 Return되는 Row수와 Inner Table의 Access Pattern에 의에 Access 의 효율이 좌우되지 않으며, Join Table간의 자신의 처리범위로만 처리량을 결정하므로 독립적이다.
5. SMJ의 단점인 많은 Row들의 처리 또는 전체적으로 Select List의 Size의 합이 큰 Table의 Join시 Sort의 CPU사용에 대한 Overhead 및 Disk Sort와 같은 문제점은 없다. 그러므로 최소한 SMJ보다는 우수하다.
6. 한 Table은 작은 Rowset Size(Return되는 Row수 와 Select List기준), 한 Table은 아주 큰 Size의 Join에 유리하다. 이러한 경우는 반드시 작은 Size를 가지고 Hash Table을 만들어야 한다. (주의. Hash Join은 순서가 매우 중요하다. 즉 작은 Rowset으로 hash Table을 만들어야 하므로)
7. Hint를 잘 못 주어서 Big Rowset이 Return되는 Table부터 Driving(Build Table)된다면 HASH_AREA_SIZE의 Memory부족으로 TEMP Disk I/O가 발생한다. 그러므로 Hint를 줄 경우 반드시 Driving순서를 정확히 주어야 한다.
8. Disk I/O를 피할 수 없는 경우라면 HASH_AREA_SIZE(default : =SORT_AREA_SIZE * 2)를 SQL마다 Session Level에 할당해서 사용하도록 한다.
(WORKAREA_SIZE_POLICY가 MANUAL일 경우이거나 9i 이전 Version에서) 또한 TEMP Tablespace의 Extent Size도 충분히 크게 주도록 한다.
HASH_MULTIBLOCK_IO_COUNT는 Optimizer에게 자동 조정하도록 설정하지 않는다.

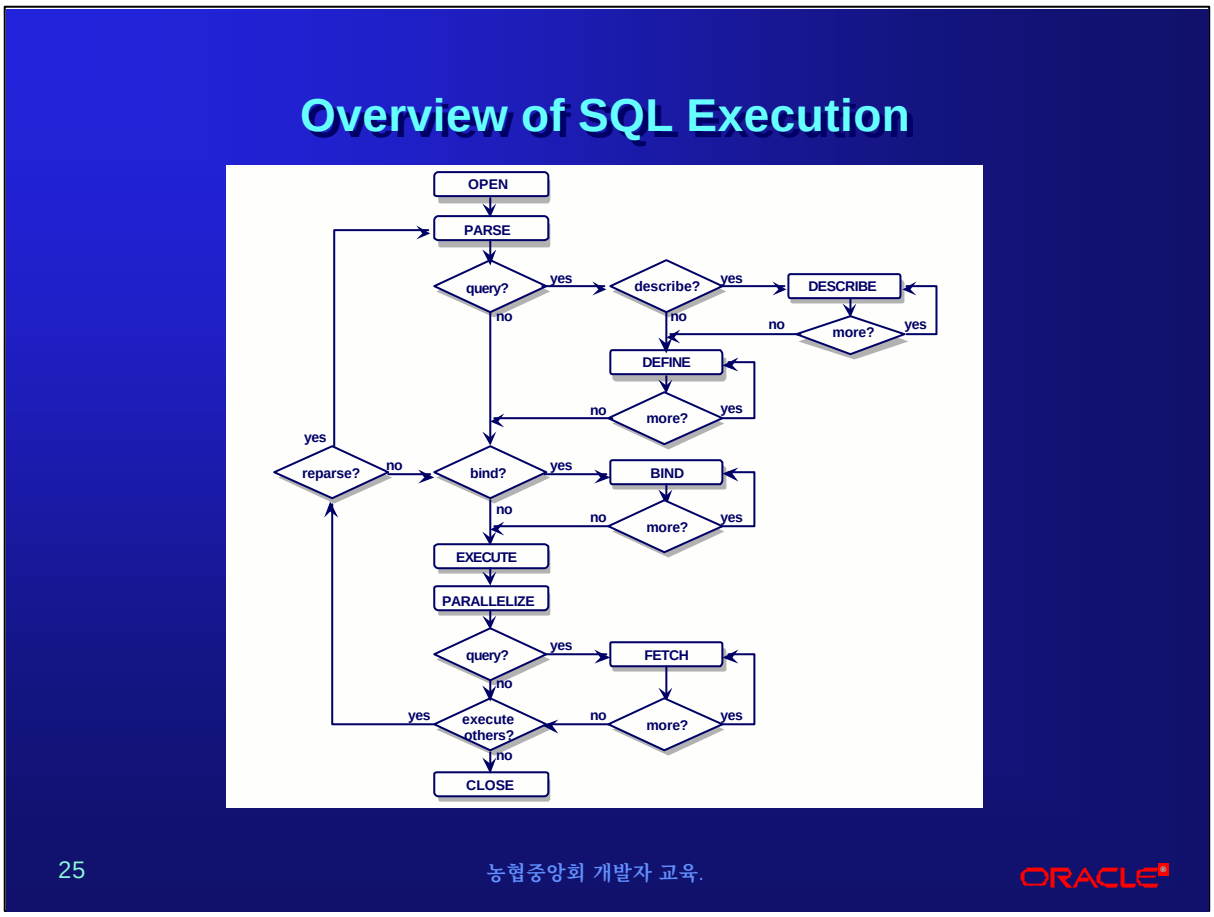
```
ALTER SESSION SET HASH_AREA_SIZE= 104857600;
```

9. Hash Memory의 Size는 (= Small Table의 Target rows * (total selected column's bytes) * 1.5) 이상 설정하되 PGA의 Memory의 한계로 인해 TEST를 통해 PGA Memory Allocation Error가 발생하지 않는 범위 내에서 설정하도록 한다. 필요시 10104 Trace를 이용해 점검한다.

```
ALTER SESSION SET EVENTS '10104 TRACE NAME CONTEXT FOREVER;
```

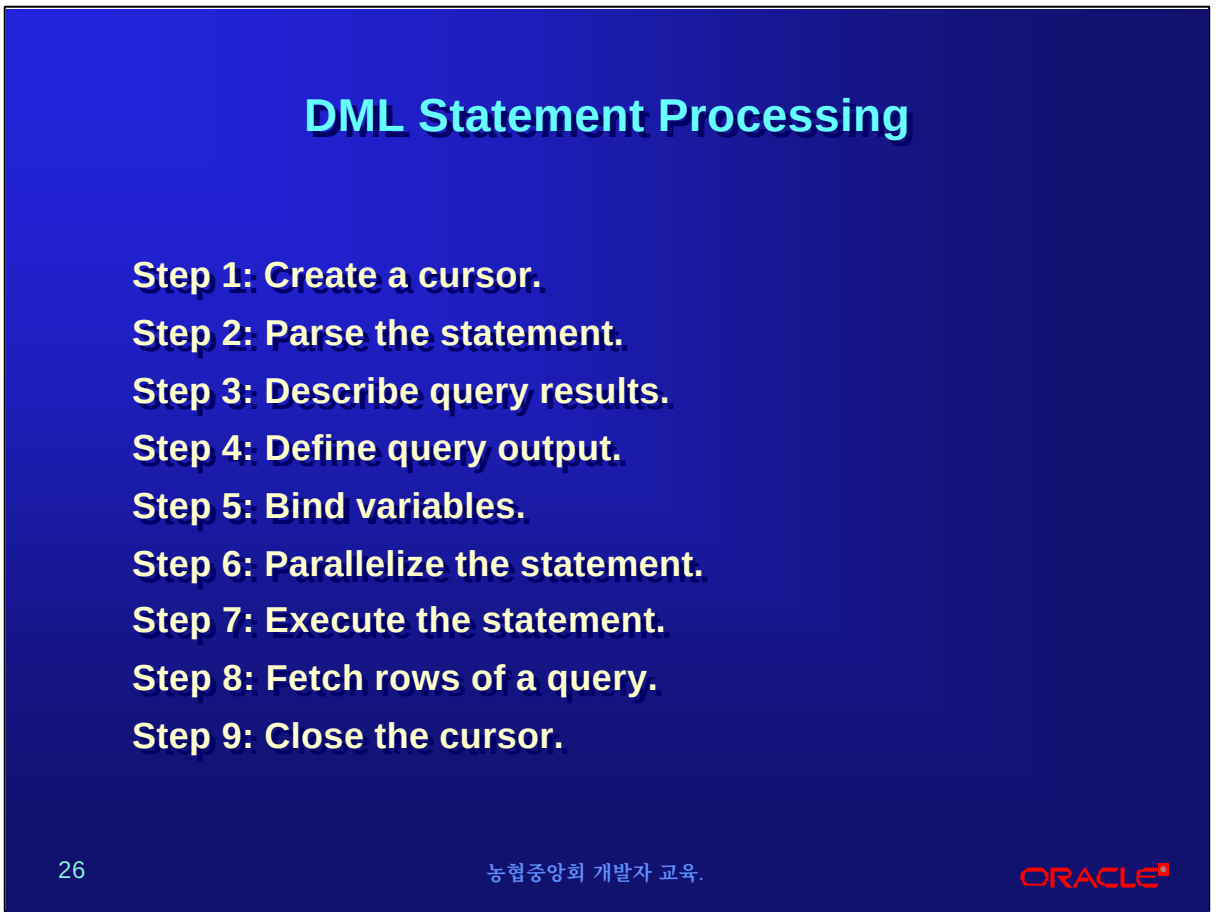
2

Query Execution Overview



Overview of SQL Execution

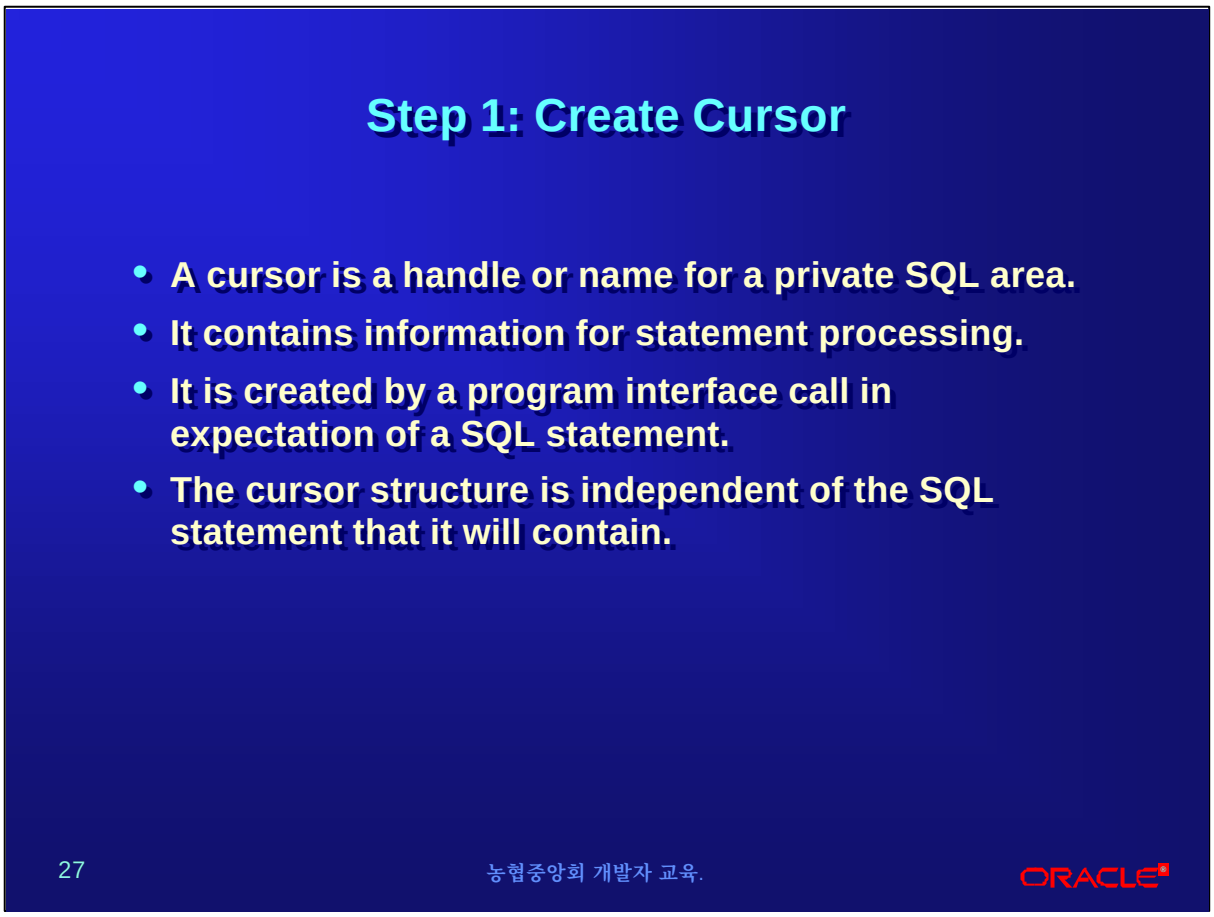
This diagram shows all the steps involved in query execution and can be found in *Oracle8i Supplied PL/SQL Packages Reference Release 2 (8.1.6)*, chapter 51 (DBMS_SQL) and in *Oracle Call Interface Programmer's Guide" Release 8.1.6.*



DML Statement Processing

Note that not all statements require all these steps. For example, DDL statements are handled in only two steps: Create and Parse.

Parallelizing the statement involves deciding that it can be parallelized as opposed to actually building parallel execution structures.



Step 1: Create Cursor

A cursor can be thought of as an association between a cursor data area in a client program and Oracle server's data structures. Most Oracle tools hide much of cursor handling away from the user, but OCI programs need the flexibility to be able to process each part of query execution separately. Therefore, precompilers allow explicit cursor declaration. Most of this can also be done using the PL/SQL package DBMS_SQL as well.

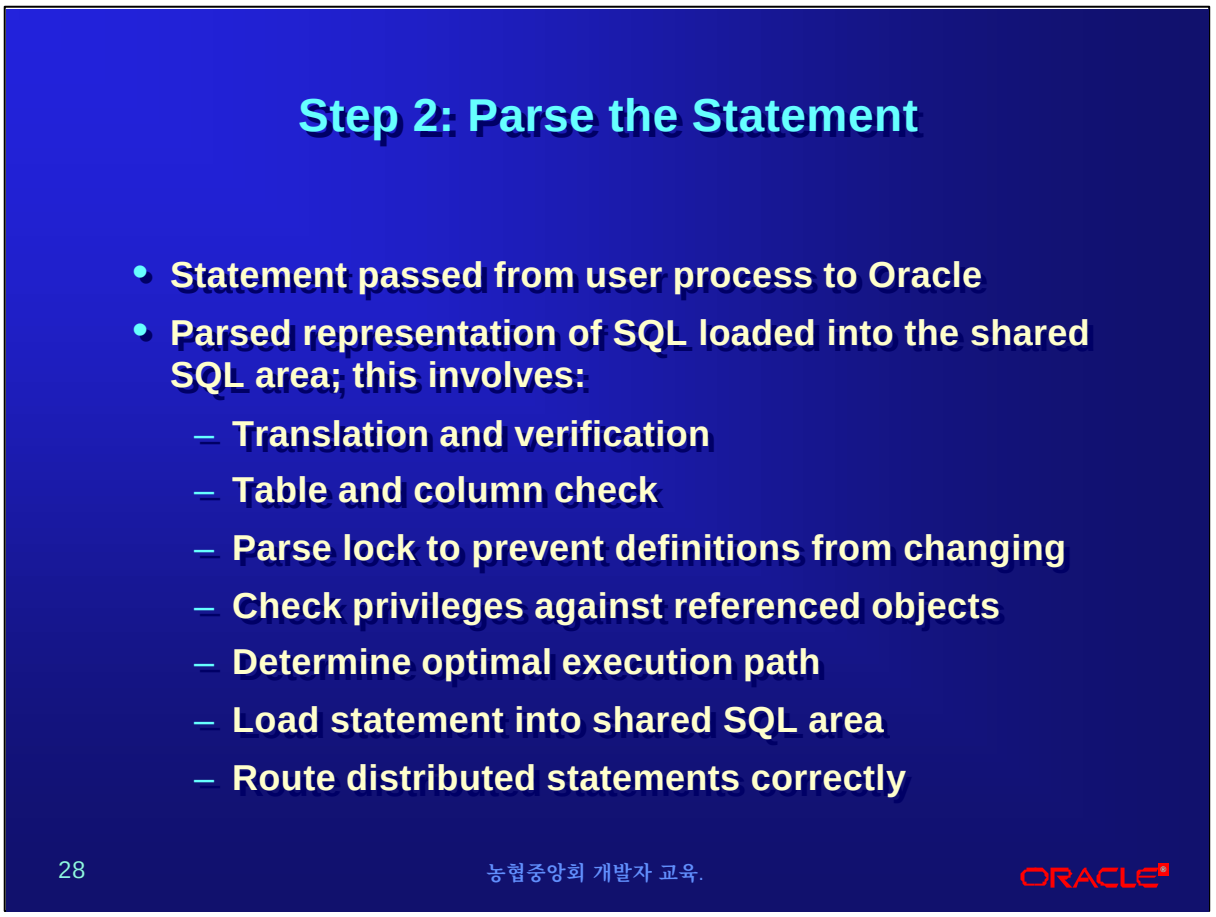
A handle can be thought of like the handle on a mug. Once you have hold of the handle you have hold of the cursor. It is a unique identifier for a particular cursor that can only be obtained by one process at a time.

Programs must have an open cursor to process a SQL statement. The cursor contains a pointer to the current row. The pointer moves as rows are fetched until there are no more rows left to process.

The following slides use the DBMS_SQL package to illustrate cursor management. This may be confusing to people unfamiliar with it; however, it is more "ST friendly" than PRO*C or OCI. It is slightly problematic in that it performs fetch and execute together, so the execute phase can't be separately identified in the trace.

Not all fields in the trace cursor output are discussed; only fields that are significant from a support perspective are included. The course shows the steps involved in query execution as opposed to an exhaustive trace file examination.

In Oracle8*i*, cursors are implemented conceptually as opposed to physically. Operations still proceed in the same manner as Oracle7.



Step 2: Parse the Statement

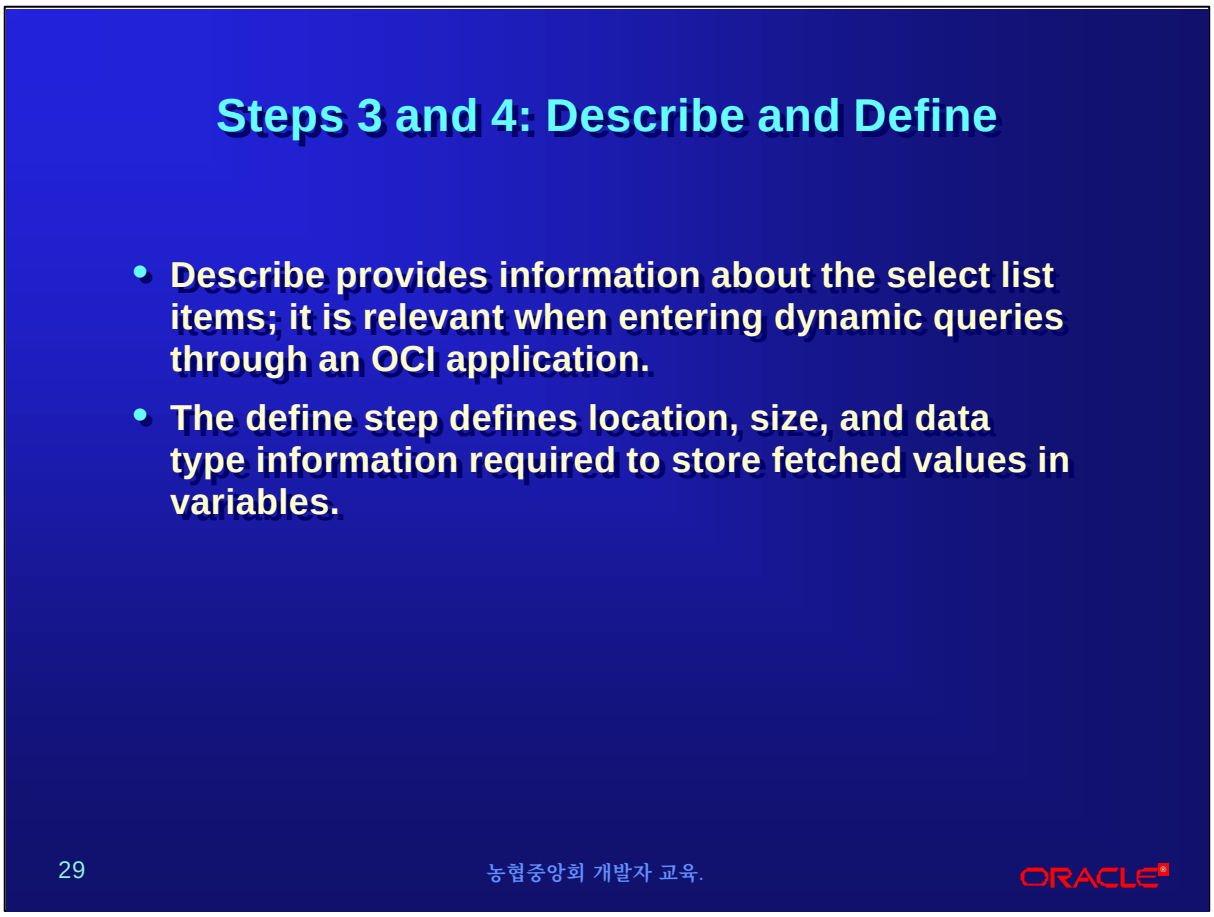
Translation and verification involves checking if the statement already exists in the library cache.

For distributed statements, check for the existence of database links.

Typically the parse phase is represented as the stage where the query plan is generated.

The parse step can be deferred by the client software to reduce network traffic. What this means is that the parse is bundled with the execute so there are fewer round-trips to the server.

It is also interesting to note that when bind variables are used, optimization still occurs in the parse phase. This makes it impossible for the optimizer to make decisions based on the actual values used for querying.



Step 3: Describe

Describe tells the application what select list items are required. If, for example, you enter a query such as:

```
SQL> select * from employees;
```

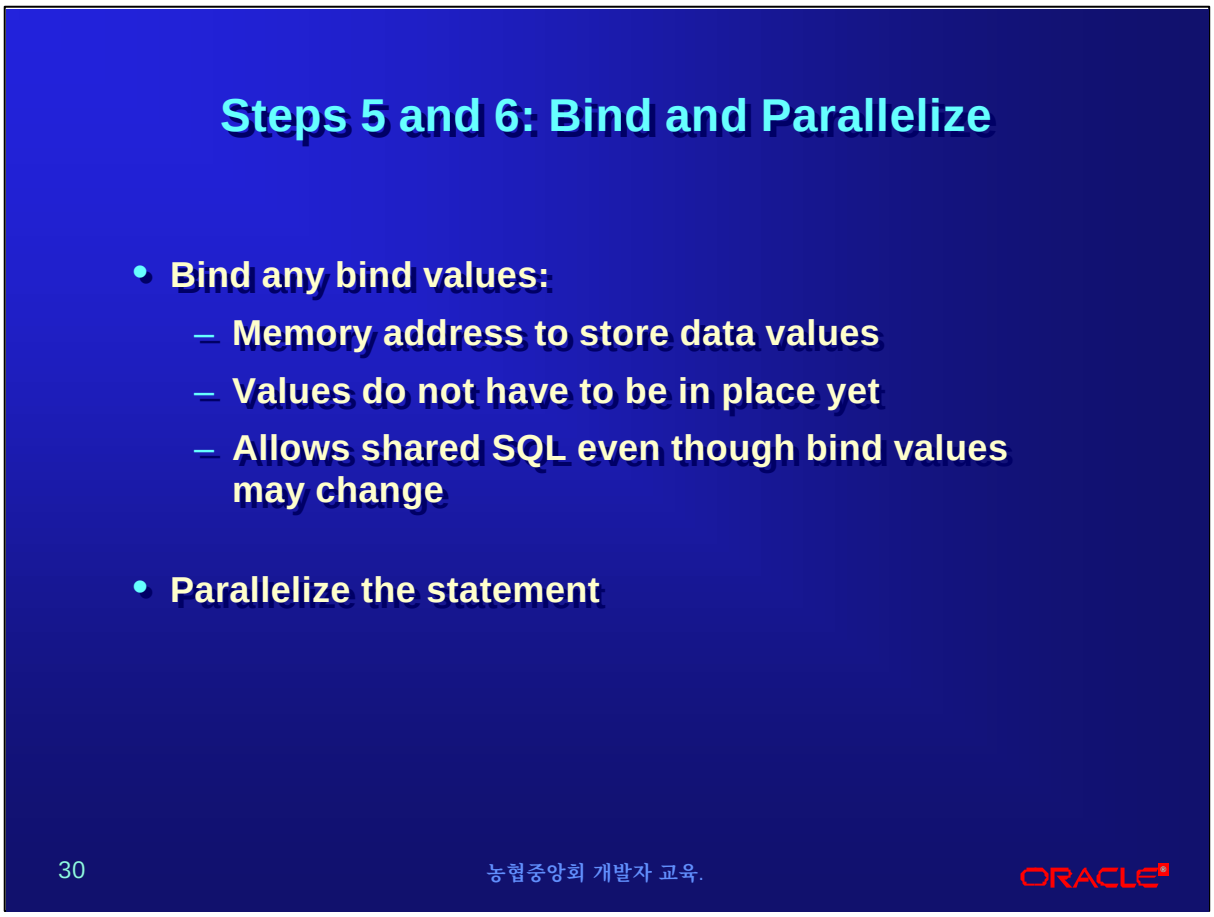
then information about the columns in the employees table is required.

Step 4: Define

Define sets up the memory for the variables that will hold the output information following the describe stage.

These two steps are generally hidden from users in tools such as SQL*Plus. However, with DBMS_SQL or OCI it is necessary to tell the client what data is going to be output and the setup areas to put it in.

Note: These two operations are only relevant for SELECT statements.



Step 5: Bind

Bind gives the Oracle server the address where bind values will be stored in memory.

```
SQL> execute :b1:='Y'
SQL> execute dbms_sql.bind_variable(:c1,':b1',:b1)
```

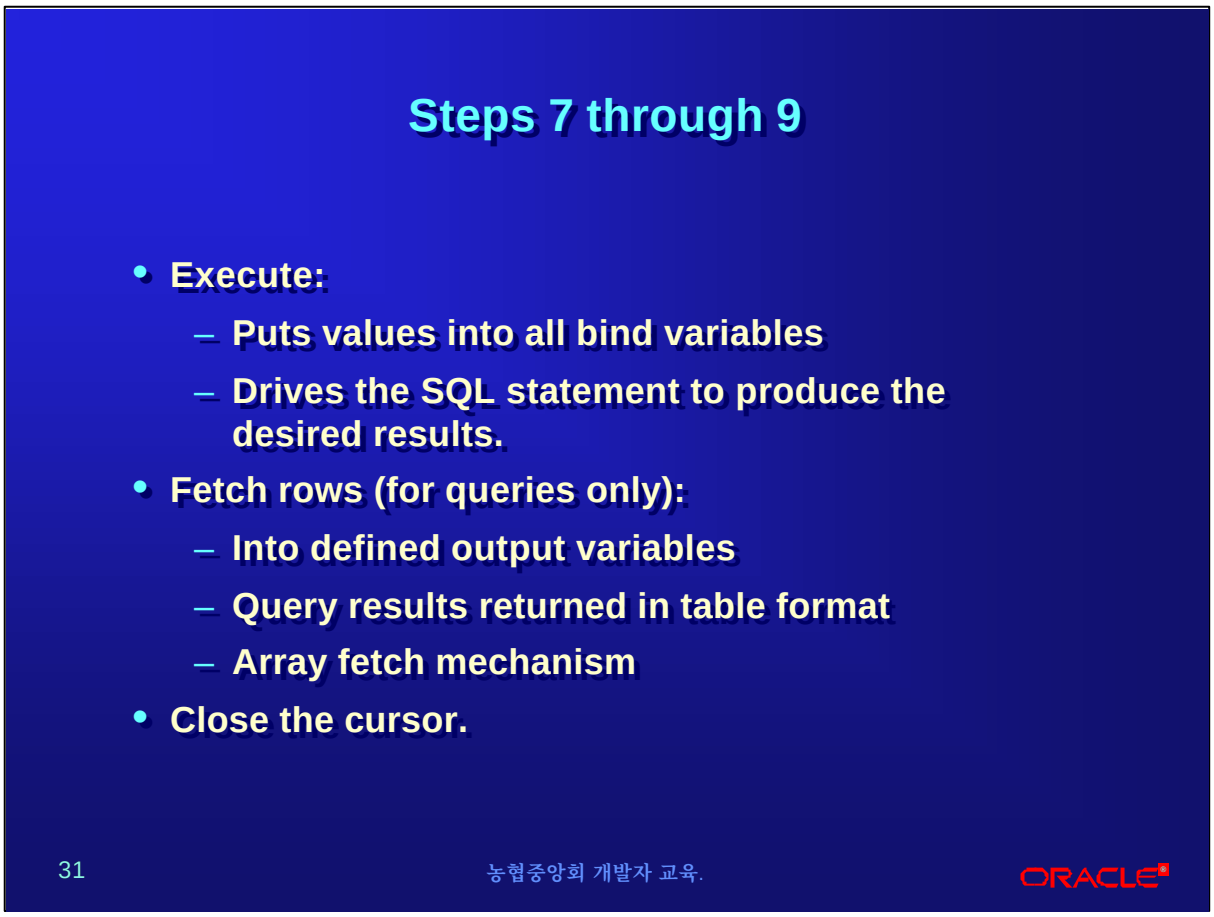
The third argument is the SQL*Plus variable; the :b1 in quotes is the one in the select that was parsed earlier.

CURBOUND indicates that the bind variables are now bound. In other words, pointers to relevant memory locations have been created.

Step 6: Parallelize

Parallelization involves dividing up the work of a query among a number of slave processes.

Parsing has already identified if a statement can be parallelized or not and has built the appropriate parallel plan. At execution time this plan is then implemented if sufficient resource is available.



Execute

The DBMS_SQL.EXECUTE procedure performs execute and fetch together in one call. This is why CURFETCH (and not CUREXEC) is found in the cursor dump below.

It is only internally that the execute and fetch is bound together; from a PL/SQL perspective it is still necessary to perform a DBMS_SQL.FETCH_ROWS to get the data.

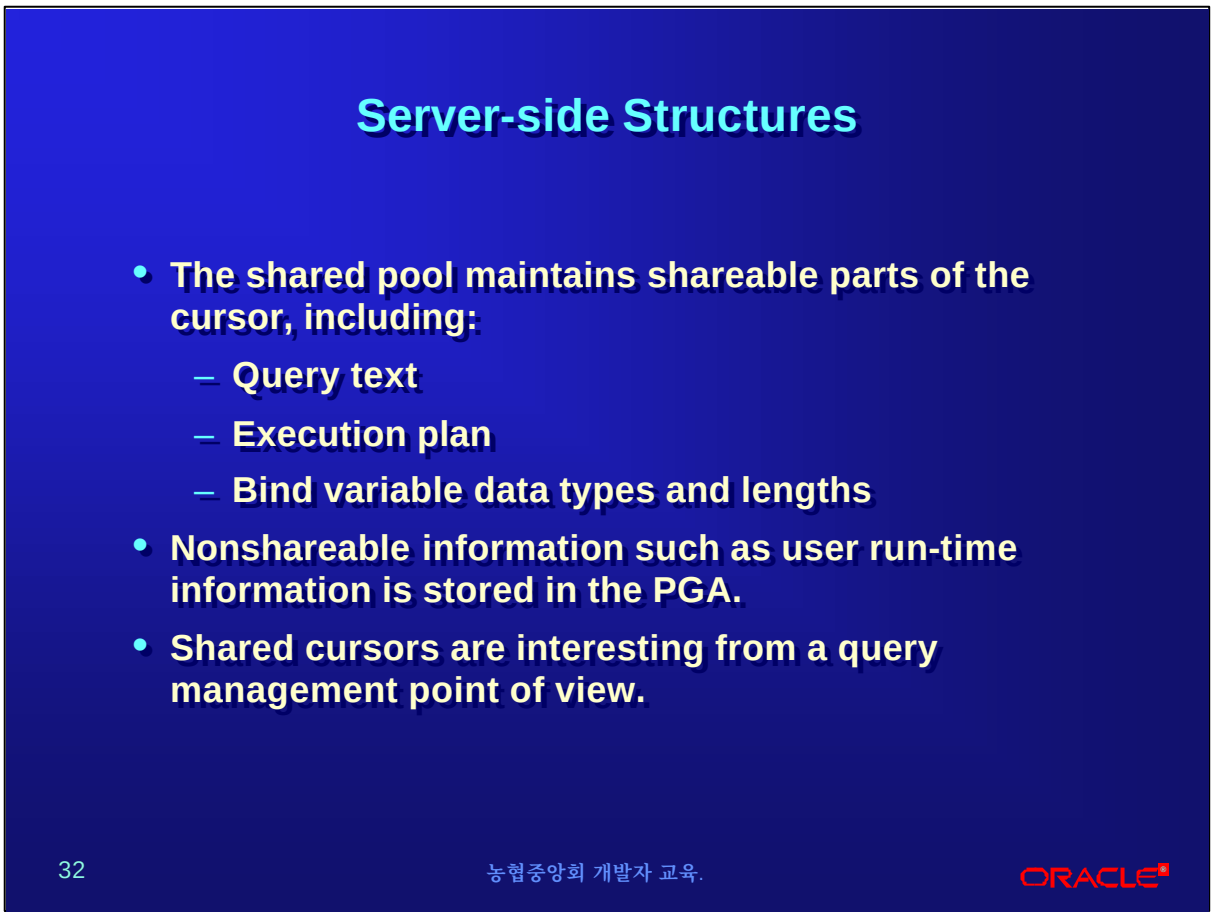
The actual bind values can be seen in the dump now the query has executed.

```
SQL> variable r number
SQL> execute :r := dbms_sql.execute(:c1);
```

```
Cursor 3 (8ec398): CURFETCH  curiob: 8ee13c
 curflg: 4f curpar: 0 curusr: 0 curses 8002d6bc
 cursor name: select null from dual where dummy = :b
 child pin:800faa88, child lock:800f9a2c, parent lock:800f95cc
 xscflg: 80310676, parent handle: 803ba53c xscfl2: 1000000
  nxt: 2.0x00000008  nxt: 1.0x00000334
Cursor frame allocation dump:
frm: -------- Comment --------  Size  Seg Off
 bind 0: dty=1 mxl=32(01) mal=00 scl=00 pre=00 oacflg=03
   oacfl2=1 size=32 offset=0 bfp=008f5b90 bln=01 avl=01 flg=05
   value="Y"
End of cursor dump
```

```
SQL> execute dbms_sql.close_cursor(:c1);
```

This removes the entry from the cursor dump altogether.

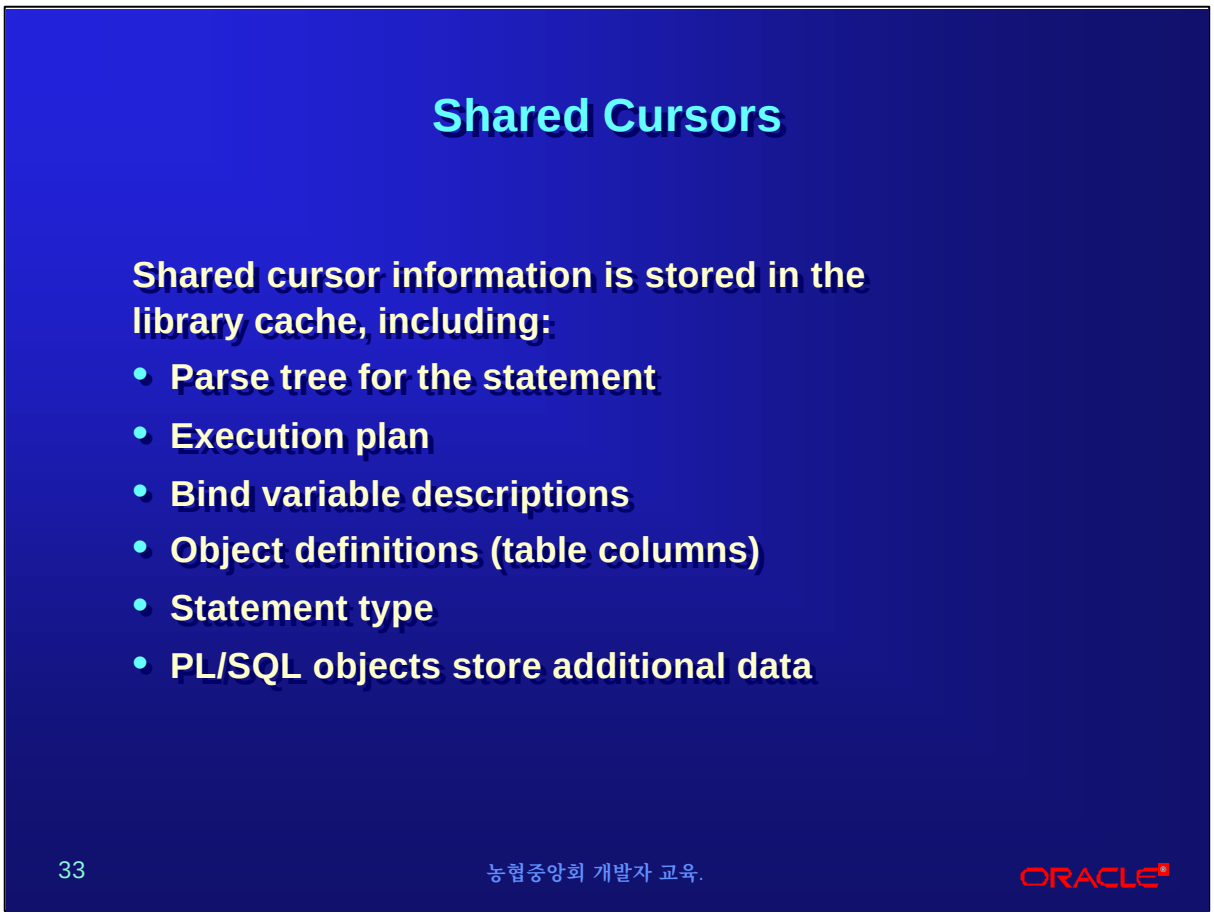


Server-side Structures

Shareable code reduces overall memory requirements and reduces parse times because parsed representations of SQL may already be stored in the library cache. The application should be designed to share code as much as possible to improve performance.

Shareability is often described in terms of persistence levels. The location of various structures is dependent on how long you want those structures to exist. Information that you want to share, such as SQL statements and plans, should persist the longest, whereas runtime-specific information has a low persistence level.

Sharing session information opens up possibilities for parallelism as multiple sessions need to see the same information. The parallel Query Coordinator process needs to be able to communicate with its slaves and vice versa. The information required is supplied by sharing.

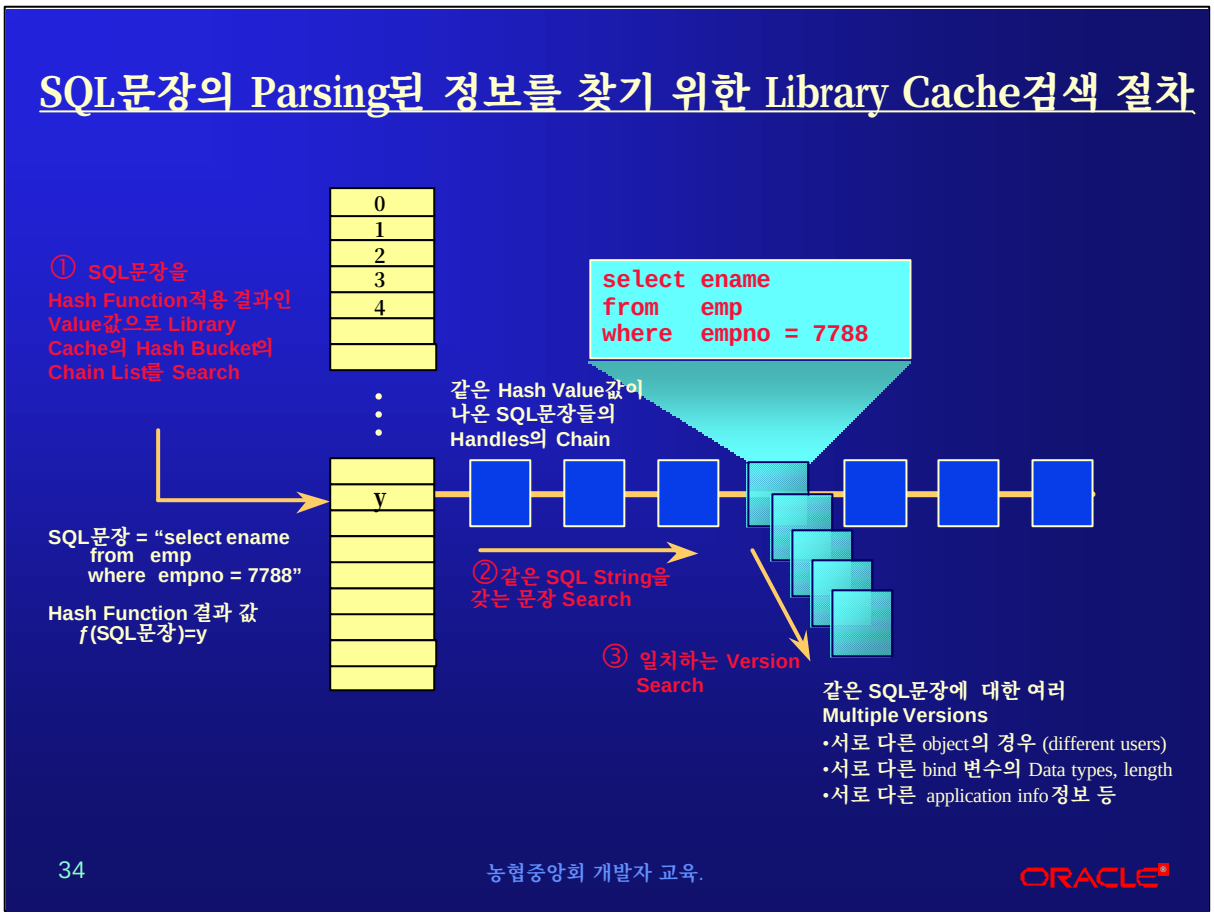


Shared Cursors

Shared cursors were a major new Oracle7 feature that enable users to share context areas associated with cursors. In the past, when a SQL statement was parsed, a context area, consisting of a large tree of data structures, was created giving a kind of execution plan of the cursor. Many types of data were intermixed within these data structures. As well as pure execution plan data, the state of execution was stored in the same context area.

In Oracle7, this state is stored separately so that the remainder (majority) of the context area can be shared. Oracle attempts to share objects as much as possible.

Only selects, inserts, updates and delete cursors are shared. DDL cursors are built in the PGA and therefore are not shared. This is based upon the premise that DDL is only executed once.



Library Cache Data Access

The library cache is a series of linked lists within the SGA containing information about a single SQL statement.

The library cache is implemented using a series of buckets, each associated with a particular hash value. Linked lists of SQL statements hang off these buckets. Each statement on a particular list hashes to a common hash value.

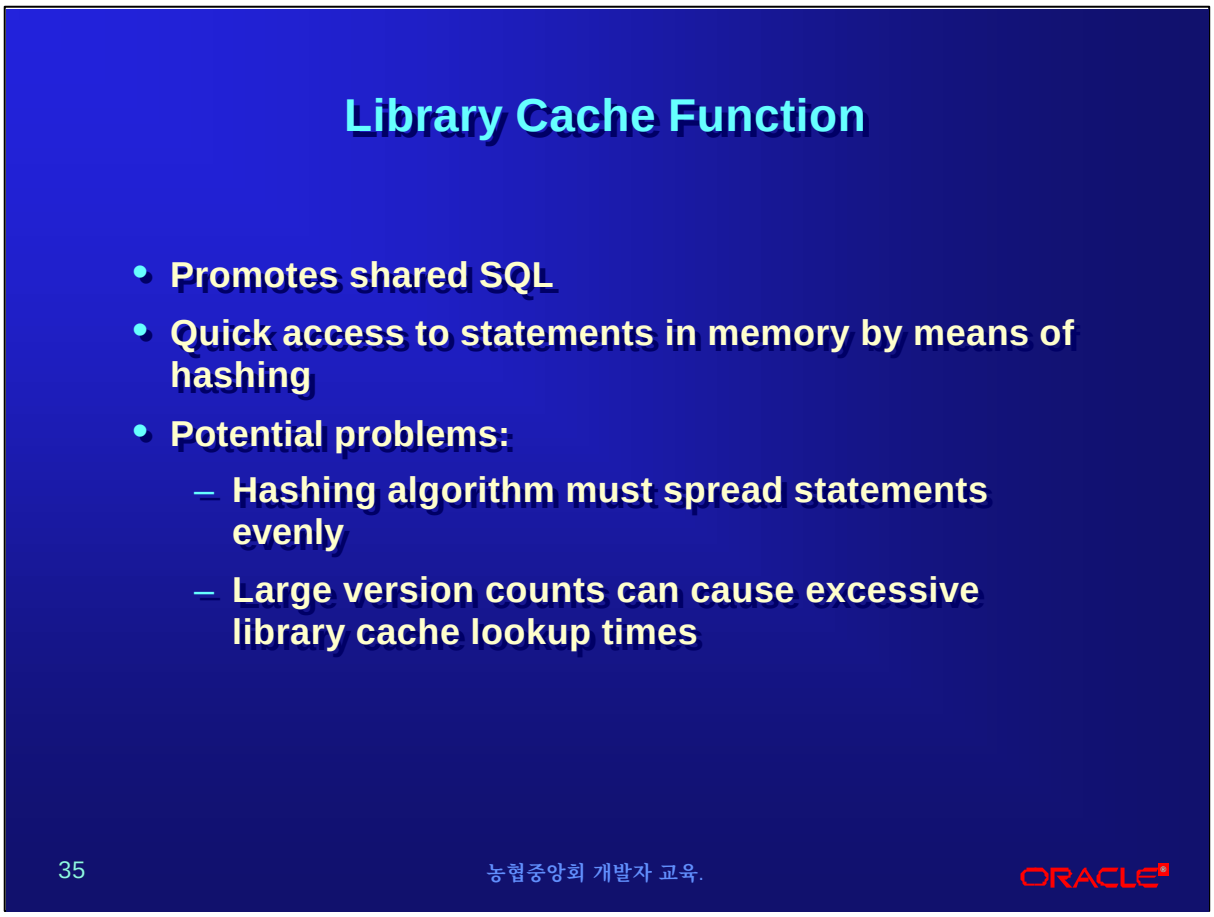
The linked list is a series of handles or references. In actual fact the SQL statement itself makes up the handle. Under each handle there can be multiple versions of the same SQL statement. Multiple versions of a statement typically occur when statements are identical but they:

- Point at different objects (different users)
- Have different bind variable types
- Have different application info details

When a new SQL statement is parsed, a hash function is applied to it, and it is placed on a linked list associated with that hash value. It is possible for different statements to have the same hash value.

To find a SQL statement in the library cache the statement has the hash function applied to it, and then the linked list is scanned looking for matching statements. If there are many versions of the statement then these are each checked through for a match.

Note: The default number of hash buckets is the number of db block buffers divided by four and rounded up to the next prime number. You can influence the number of hash buckets with the hidden *_db_block_hash_buckets* parameter.



Library Cache Function

For performance reasons it is not a good idea to have lots of SQL statements hanging off the same list. The most significant problems with the library cache were down to too many statements hashing to the same bucket, huge numbers of versions of statements, or excessive use of literal values instead of bind variables.

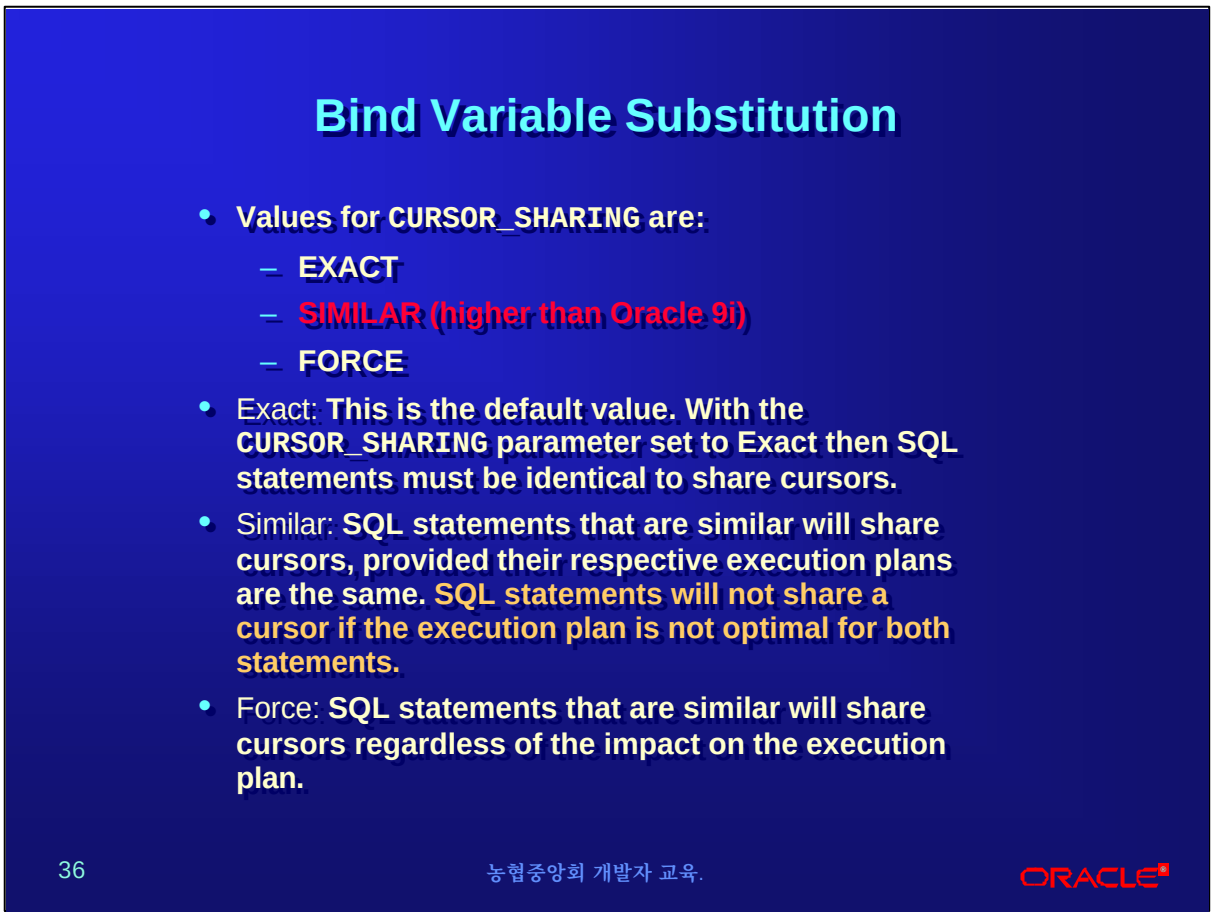
There is a history lesson in this: When the sharing of cursors was introduced in 1992 the number of concurrent users on an Oracle database rarely exceeded 500, and it was quite usual that Oracle tools were used—implying that SQL was only parsed when necessary. Furthermore, many developers at that time knew that parsing was expensive and should be avoided as much as possible.

Today many developers don't know this; more users, powerful servers and development tools not optimized for Oracle result in systems with extreme amounts of parse calls—and in many cases without bind variables.

These systems seem to have a common behaviour: response times are acceptable just after database startup, but after some time (maybe hours) the response times rise sharply. When examining V$SYSTEM_EVENT and V$LATCH you can see that it is the "library cache" and the "shared pool" latches that now are responsible for a significant part of the response time.

According to the published articles on these latches the first suggestion is to increase the shared pool. This helps in that it takes longer time after database start before the problem occurs, but then it reappears with even longer response times.

After this you can execute "alter system flush shared_pool" which in this case will bring the response times back down for a while. There is experience with systems that needed to be flushed every half hour in order to achieve an acceptable performance.



SQL Without Bind Variables

When CURSOR_SHARING = FORCE is specified either at the instance or the session level, cursors that differ only in literal values only will be rewritten using system generated bind variables before being parsed.

This will result in the optimizer not being able to determine the selectivity as precisely as with literal values (selectivity will be discussed in depth in a later lesson), but needs only one entry in the library cache.

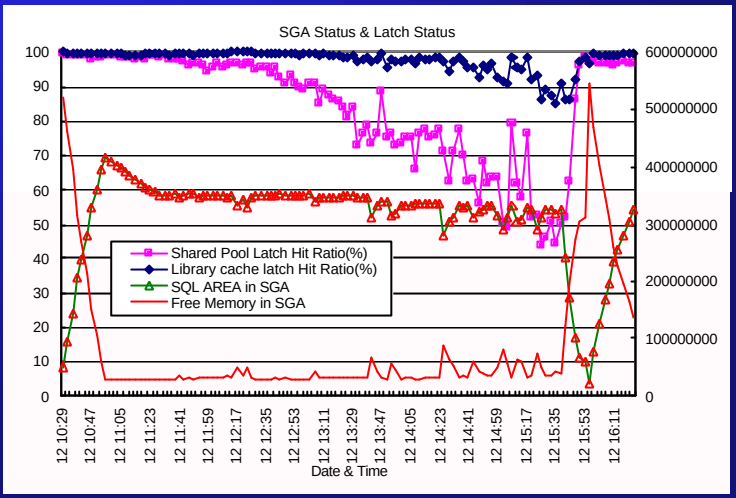
Therefore CURSOR_SHARING = FORCE will probably be best suited for OLTP systems.

From *Oracle8i Designing and Tuning for Performance*, chapter 19:

> You should consider setting CURSOR_SHARING to FORCE if you can answer "yes" to both of the following questions:
>
> 1. Are there statements in the shared pool that differ only in the values of literals?
> 2. Is the response time low due to a very high number of library cache misses?

```
-- CURSOR_SHARING = SIMILAR
-- 분포도가 Skewed 된 형태의 CURSOR_SHARING의 경우 Peaking 기능으로
-- Bind 변수를 사용하는 형태의 SQL 로 변경은 되었지만, Plan은
-- 상수일 때와 같은 것을 사용.

SQL> alter system set cursor_sharing=similar;

SQL> connect scott/tiger

SQL> select deptno,count(*) from testemp10 group by deptno;

    DEPTNO   COUNT(*)
---------- ----------
        10      18432
        20          5
        30          6

SQL> set autot traceonlyexplain
SQL> select * from testemp10 where deptno = 20;

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=5 Bytes=155)
   1   0  TABLE ACCESS (BY INDEX ROWID) OF 'TESTEMP10' (Cost=2 Card=5 Bytes=155)
   2   1    INDEX (RANGE SCAN) OF 'TESTEMP10_DEPTNO' (NON-UNIQUE) (Cost=1 Card=5)

SQL> select * from testemp10 where deptno = 30;

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=6 Bytes=186)
   1   0  TABLE ACCESS (BY INDEX ROWID) OF 'TESTEMP10' (Cost=2 Card=6 Bytes=186)
   2   1    INDEX (RANGE SCAN) OF 'TESTEMP10_DEPTNO' (NON-UNIQUE) (Cost=1 Card=6)

SQL> select * from testemp10 where deptno = 10;

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=12 Card=18432 Bytes=571392)
   1   0  TABLE ACCESS (FULL) OF 'TESTEMP10' (Cost=12 Card=18432 Bytes=571392)

SQL> select * from testemp10 where deptno = 20;
     .....

SQL> select * from testemp10 where deptno = 30;
     .....

SQL> select * from testemp10 where deptno = 10;
     .....

SQL> connect sys/manager as sysdba

SQL> select * from v$SQL;
```

비공유 SQL에 의한 Memory Fragmentation 발생현상 사례

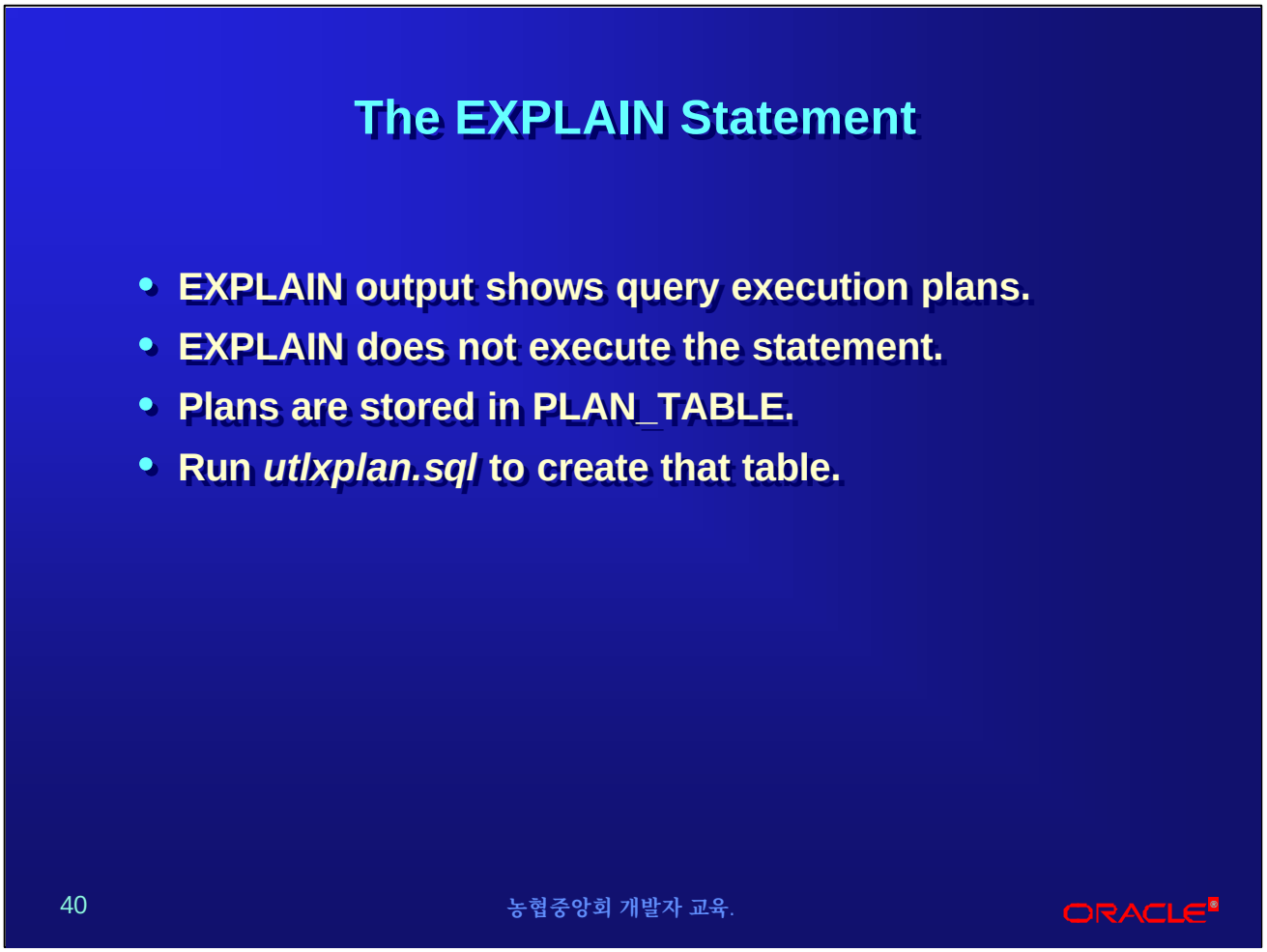


3

Query Tuning Tools and Plan Explanation

Row Sources and EXPLAIN PLAN

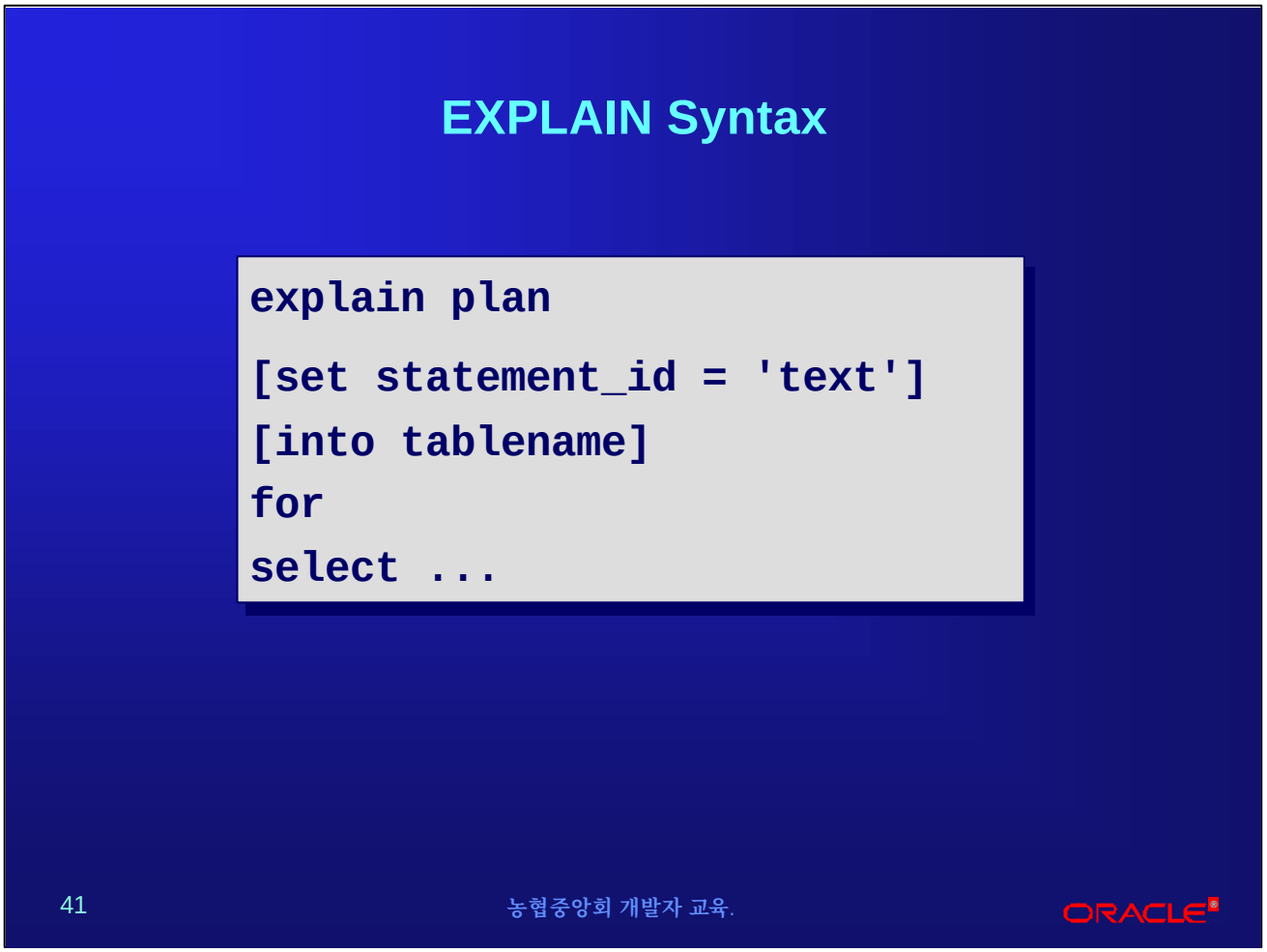
- The execution of a SQL statement is composed of small building blocks called “row sources.”
- The combination of row sources for a statement is called the execution plan.
- The execution plan gives important information when tuning SQL statements.



The EXPLAIN Statement

The EXPLAIN statement is available in all versions of Oracle since version 5.

On UNIX, run `$ORACLE_HOME/rdbms/admin/utlxplan.sql` before using EXPLAIN (on NT the path is something like `%ORACLE_HOME\rdbms\admin`). This script creates PLAN_TABLE, a utility table that holds the output from the EXPLAIN command.



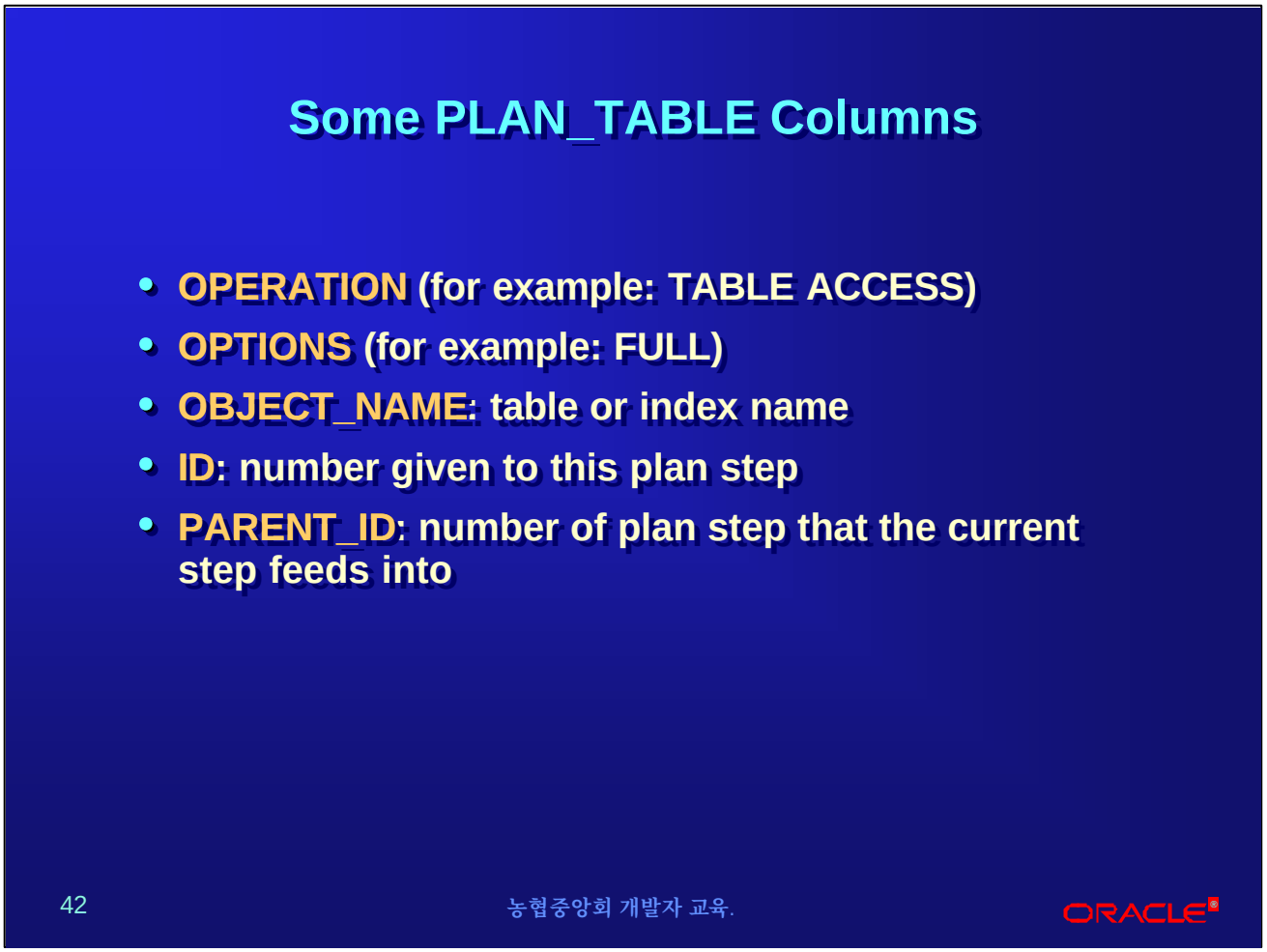
EXPLAIN Syntax

You need the STATEMENT_ID clause if you use PLAN_TABLE to hold several different query execution plans. The STATEMENT_ID acts as an identifier.

You need the INTO clause if you use a table other than PLAN_TABLE to hold the execution plans. PLAN_TABLE is the default. Any other table used for plans would have to have the same structure as PLAN_TABLE.

PLAN_TABLE is not self-clearing: rows should be deleted or truncated when finished working on a plan.

In this course the focus is on queries, but the EXPLAIN command can also be used for DML statements (INSERT, UPDATE, DELETE).



PLAN_TABLE Columns

The columns above show the basic information about steps in the query plan. They describe which objects are accessed, the access or manipulation method, and the order of plan steps.

For the first row in a plan, the OPERATION column contains a description of the statement (for example, SELECT STATEMENT for queries).

Other columns that are less often used:

Column	Description
STATEMENT_ID	Populated by SET STATEMENT_ID, if used
TIMESTAMP	Date and time of the EXPLAIN statement
REMARKS	Can only be updated directly, not through EXPLAIN
OBJECT_NODE	Link name for remote objects, table queue name for parallel query
OBJECT_OWNER	Schema name for the object
OBJECT_INSTANCE	Number of the object within the statement, counting from left to right
OBJECT_TYPE	For example, NON_UNIQUE for indexes
SEARCH_COLUMNS	Not currently used
POSITION	Order of processing if steps have the same PARENT_ID

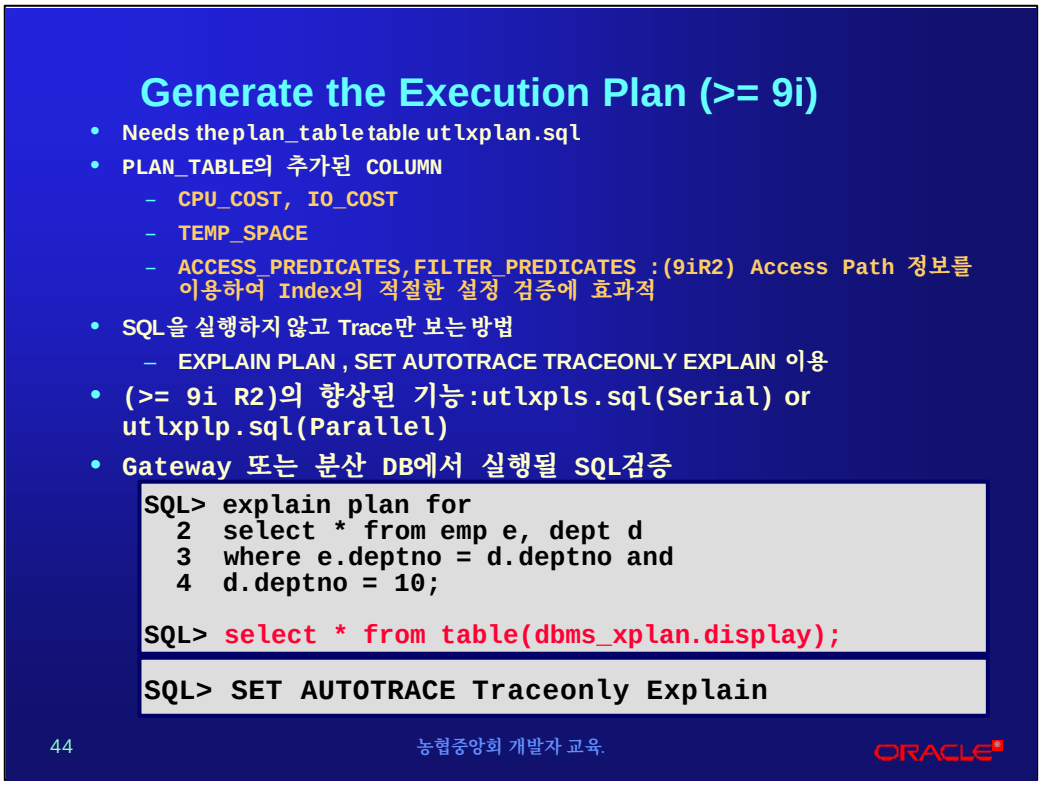
Formatting PLAN_TABLE Output

PLAN_TABLE is hierarchical. The link is established with the ID and PARENT_ID columns.

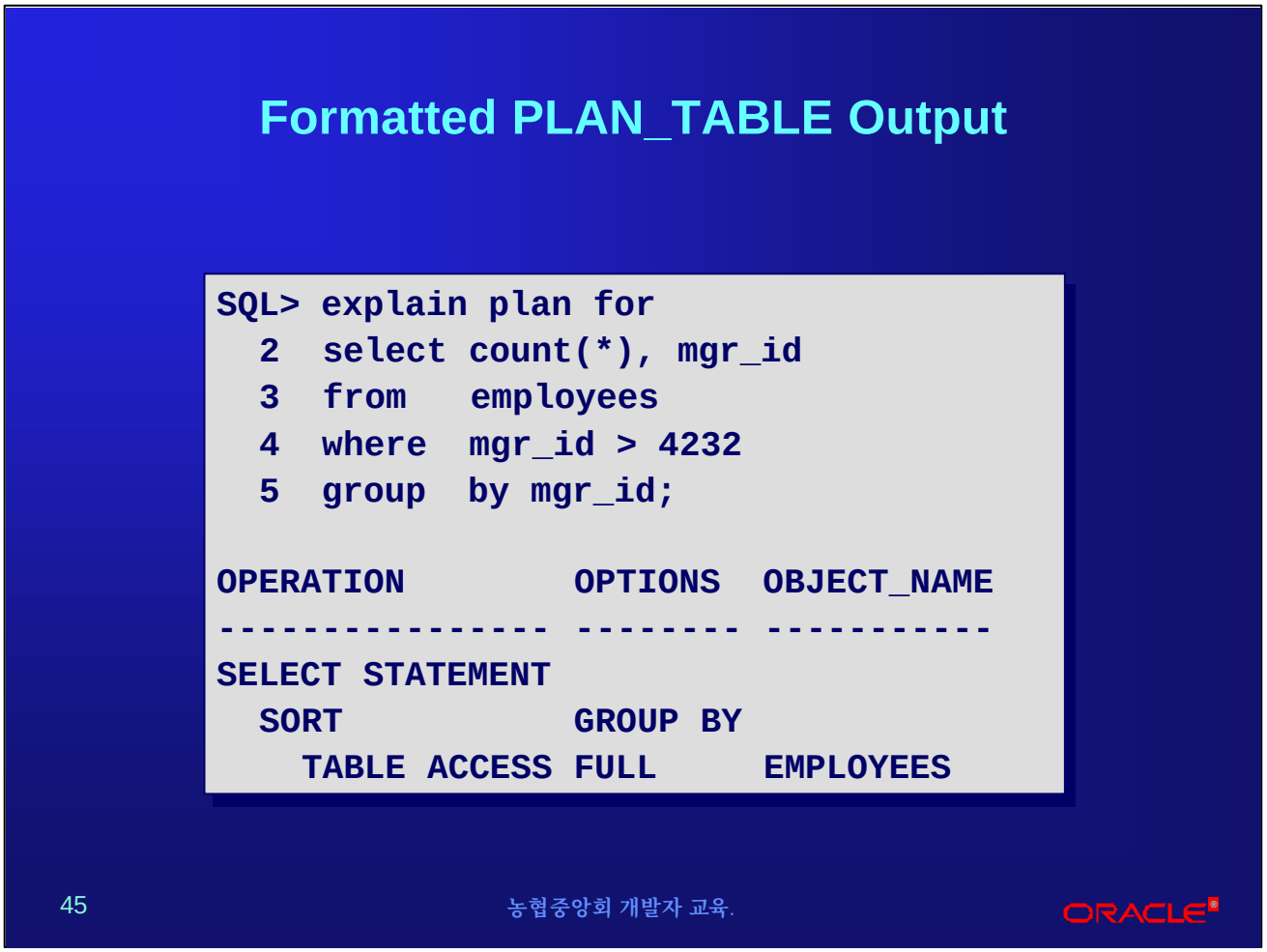
```
SQL> select  lpad('  ',2*level-1)||
  2          operation as operation
  3  ,       options, object_name
  4  from    plan_table
  5  start   with id = 0
  6  connect by prior id=parent_id;
```


Formatting PLAN_TABLE Output

The next slide gives an example of the output from this query.




```
SQL> explain plan for
  2  select * from emp e, dept d where e.deptno = d.deptno and d.deptno = 10;

Explained.

SQL> select * from table(dbms_xplan.display);

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

--------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost  |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |     4 |   248 |     3 |
|   1 |  NESTED LOOPS                |             |     4 |   248 |     3 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPT        |     1 |    30 |     1 |
|* 3  |    INDEX RANGE SCAN          | PK_DEPT     |     1 |       |     1 |
|* 4  |   TABLE ACCESS FULL          | EMP         |     5 |   160 |     2 |
--------------------------------------------------------------------------------

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

   3 - access("D"."DEPTNO"=10)
   4 - filter("E"."DEPTNO"=10)

Note: cpu costing is off

================================================================================
SQL> set autot traceonly explain
SQL> SELECT * FROM emp e, dept@scott_9ir2 d
  2  where e.deptno = d.deptno and d.deptno = 10
  3  order by ename;

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=6 Card=4 Bytes=248)
   1    0   SORT (ORDER BY) (Cost=6 Card=4 Bytes=248)
   2    1     NESTED LOOPS (Cost=3 Card=4 Bytes=248)
   3    2       REMOTE* (Cost=1 Card=1 Bytes=30)  SCOTT_9IR2.US.ORACLE.COM
   4    2       TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=5 Bytes=160)

   3 SERIAL_FROM_REMOTE            SELECT "DEPTNO","DNAME","LOC" FROM "DEPT" "D
                                   " WHERE "DEPTNO"=10
```



Formatted PLAN_TABLE Output

To get a display similar to the above it is necessary to format the columns in SQL*Plus, making the display shorter than the default. For example:

```
SQL> col operation   format a25
SQL> col options     format a20
SQL> col object_name format a25
```

Note: EXPLAIN output can only shows how the query would be implemented if the query was run at this moment. If the schema changes (for example, someone adds or drops an index) the query may be executed quite differently. This is often important to stress when a customer is providing trace output.

PLAN_TABLE: Optimization

The OPTIMIZER value for the first row in the table shows optimizer mode, as set at instance or session level. For later rows, it shows whether objects have been analyzed.

The COST column is only populated if the cost-based optimizer is used. This is the most definitive indicator of optimizer method, and overrides any different value in the OPTIMIZER column.

The figure in the COST column has no absolute meaning. Use it relatively: A higher figure means more work than a lower one.

The figure in COST is a sum of the cost of the row source itself and all row sources “below” this row source.

CARDINALITY and BYTES are both estimates.

Note: Optimization methods are dealt with in detail in another lesson of this course.

PLAN_TABLE: Parallelism

- **OBJECT_NODE**: Table queue name
- **OTHER**: SQL given to the query slaves
- **OTHER_TAG**: Describes parallel query and remote access steps
- **DISTRIBUTION**: The method used to distribute rows from producer query servers to consumer query servers.

PLAN_TABLE: Parallelism

Table queues are memory structures used to pass information between sets of query slaves. Each parallel operation feeds results into a single table queue, so this column can be useful in checking which steps are involved in the same parallel operation.

Possible values for OTHER_TAG are:

- SERIAL_FROM REMOTE
- SERIAL_TO_PARALLEL
- PARALLEL_TO_SERIAL
- PARALLEL_TO_PARALLEL
- PARALLEL_COMBINED_WITH_PARENT
- PARALLEL_COMBINED_WITH_CHILD

A null value means that the operation is done serially.

Possible values for DISTRIBUTION are:

- PARTITION (ROWID)
- PARTITION (KEY)
- HASH
- RANGE
- ROUND-ROBIN
- BROADCAST
- QC (ORDER)
- QC (RANDOM)

Parallel query operations are covered in more detail later in the course.

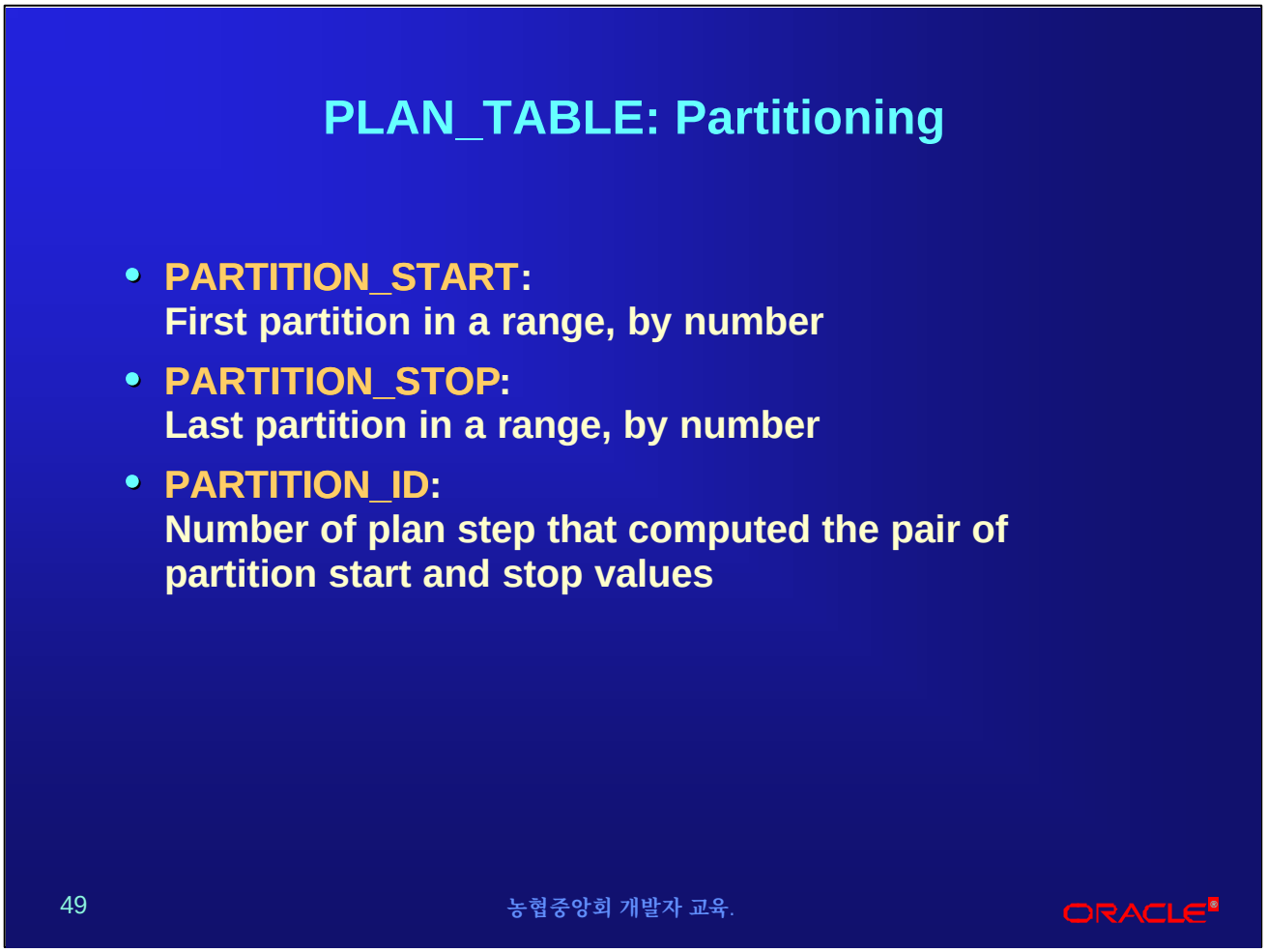
PLAN_TABLE: Remote Operations

- **OBJECT_NODE**: Database link name
- **OPERATION**: Shows 'REMOTE'
- **OTHER**: SQL statement sent to the remote database

PLAN_TABLE: Remote Operations

In the SQL statement, the column and table names show up in double quotes, and the table names are aliased. This avoids naming issues and conflicts.

To find out how the SQL will be executed on the remote node, it is necessary to run EXPLAIN against it there. The local EXPLAIN facility will not describe a remote execution plan.

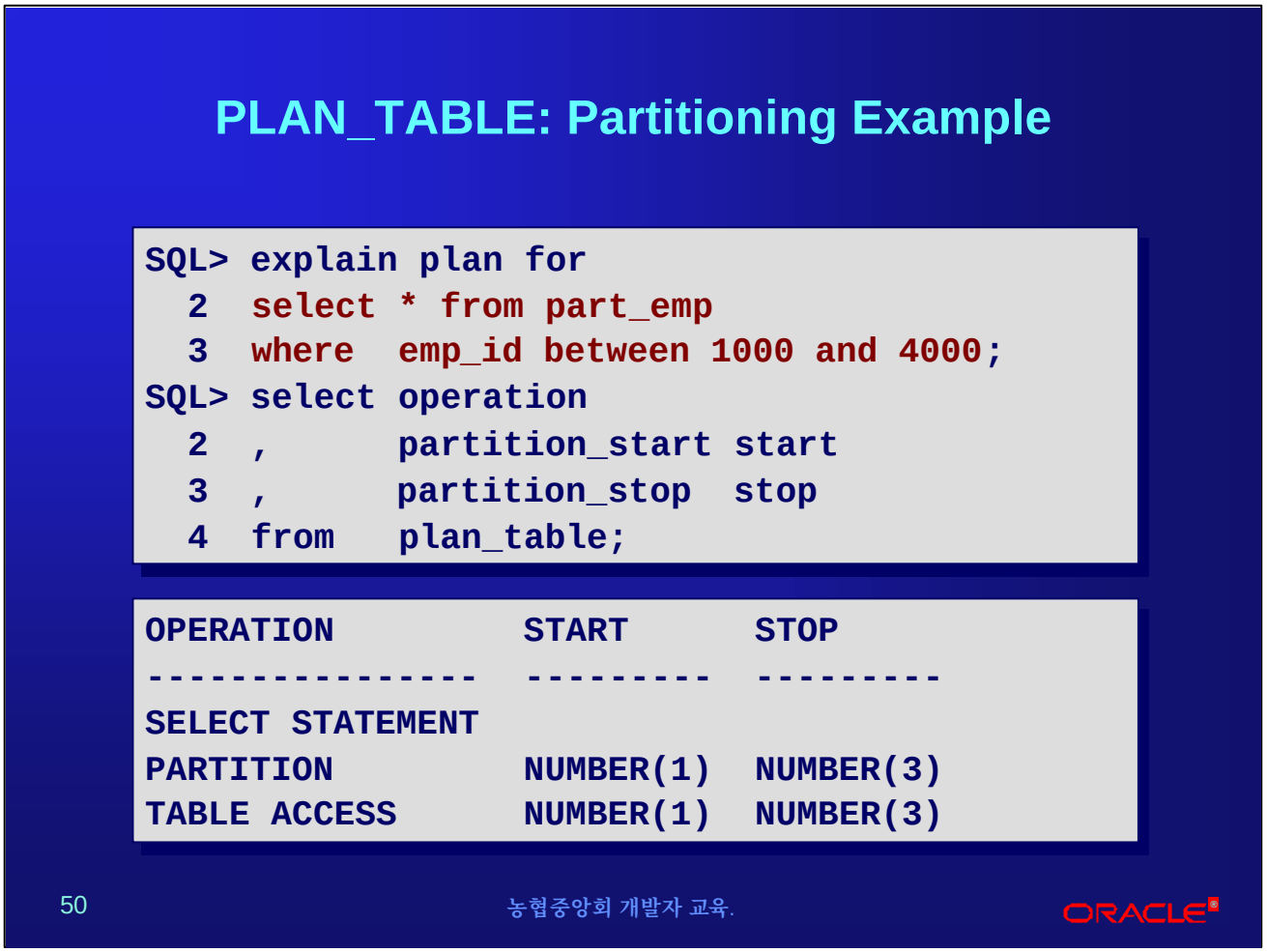


PLAN_TABLE: Partitioning

PARTITION_START and PARTITION_STOP can take these values:

- NUMBER(*n*): The SQL compiler has identified the partition; *n* is the value of DBA_TAB_PARTITIONS.PARTITION_POSITION for that partition.
- KEY: The partition will be identified at execution time (when using bind variables).
- ROW LOCATION: The partition will be computed at execution time from the record location (supplied by the user, or through a global index).
- INVALID: The range of accessed partitions is empty.

For more information, see the EXPLAIN section of the *Oracle8i SQL Reference Guide*.



PLAN_TABLE: Partitioning Example

In this example, the PARTEMP table has four partitions. Check DBA_TAB_PARTITIONS.PARTITION_POSITION to identify which actual partition will be accessed.

The PARTITION step simply indicates that partitioning is relevant to the query.

Reading EXPLAIN Output

- Using formatted output:
  - Each step (line in the execution plan) represents a row source
  - The output is a representation of a tree
  - If two steps are indented at the same level, the higher one is executed first
- The topmost leaf in the tree is where the execution starts

Reading EXPLAIN Output

```
SQL> select crs.short_name
  2  from   courses crs, classes cls
  3  where crs.crs_id = cls.crs_id
  4    and cls.class_id = 54303;
Execution Plan
----------------------------------------------
SELECT STATEMENT Optimizer=FIRST_ROWS
 NESTED LOOPS                                  0
  TABLE ACCESS (BY INDEX ROWID) OF 'CLASSES'   2
   INDEX (UNIQUE SCAN) OF CLS_PK (UNIQUE)      1  (leaf)
  TABLE ACCESS (BY INDEX ROWID) OF 'COURSES'   4
   INDEX (UNIQUE SCAN) OF CRS_PK (UNIQUE)      3  (leaf)
```

In this query the operation is a nested loops join. The first action (the topmost leaf) is:

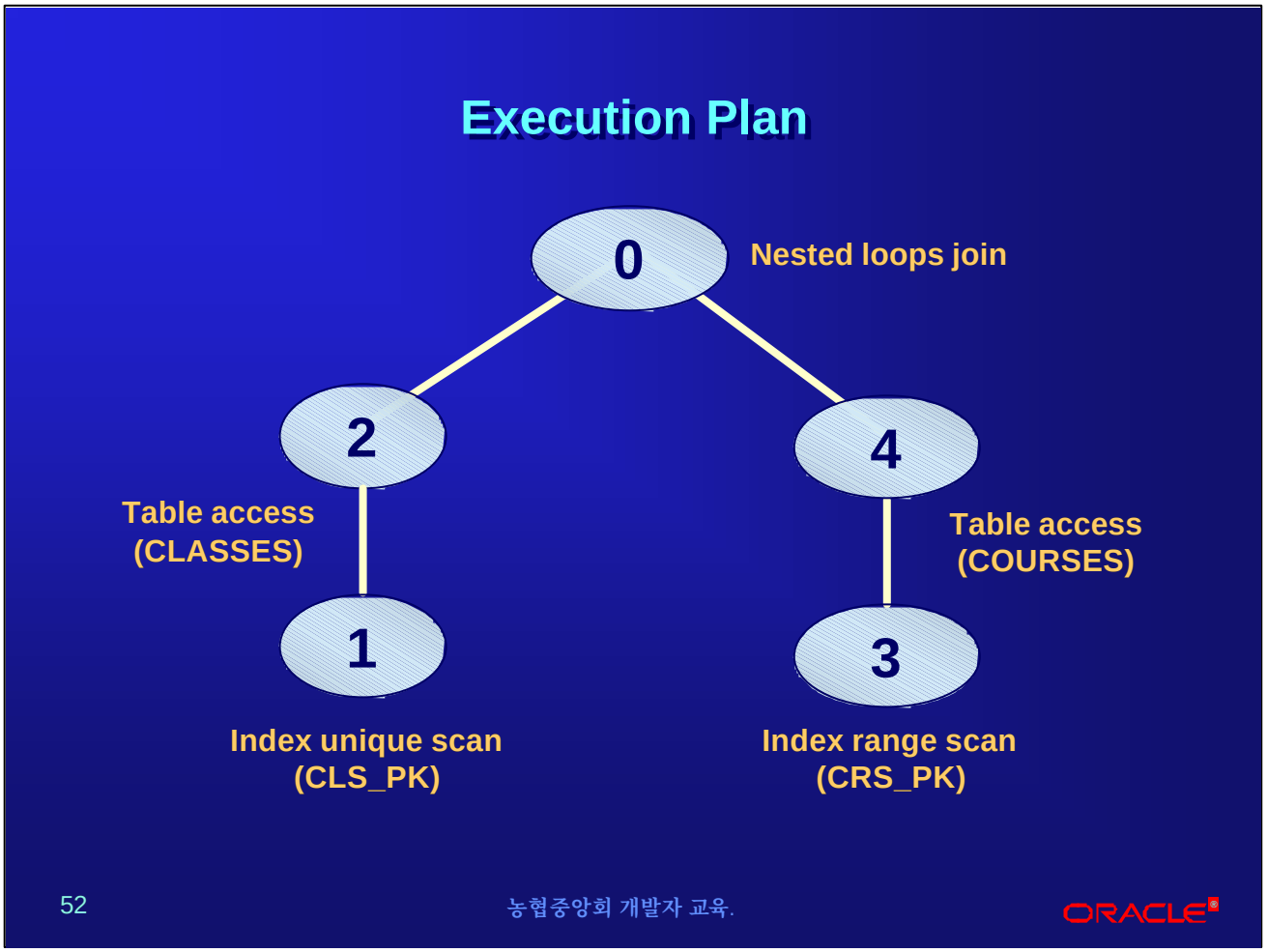
```
INDEX (UNIQUE SCAN) OF CLS_PK (UNIQUE)
```

This is the driving object. Rows from this object then identify rows in the object CLASSES this is the next action. The hereby retrieved rows are then used to access rows in the probed object COURSES. COURSES is accessed using the CRS_PK index. This is the third action.

```
INDEX (UNIQUE SCAN) OF CRS_PK (UNIQUE)
```

Finally the table lookup takes place:

```
TABLE ACCESS (BY INDEX ROWID) OF ‘COURSES’
```

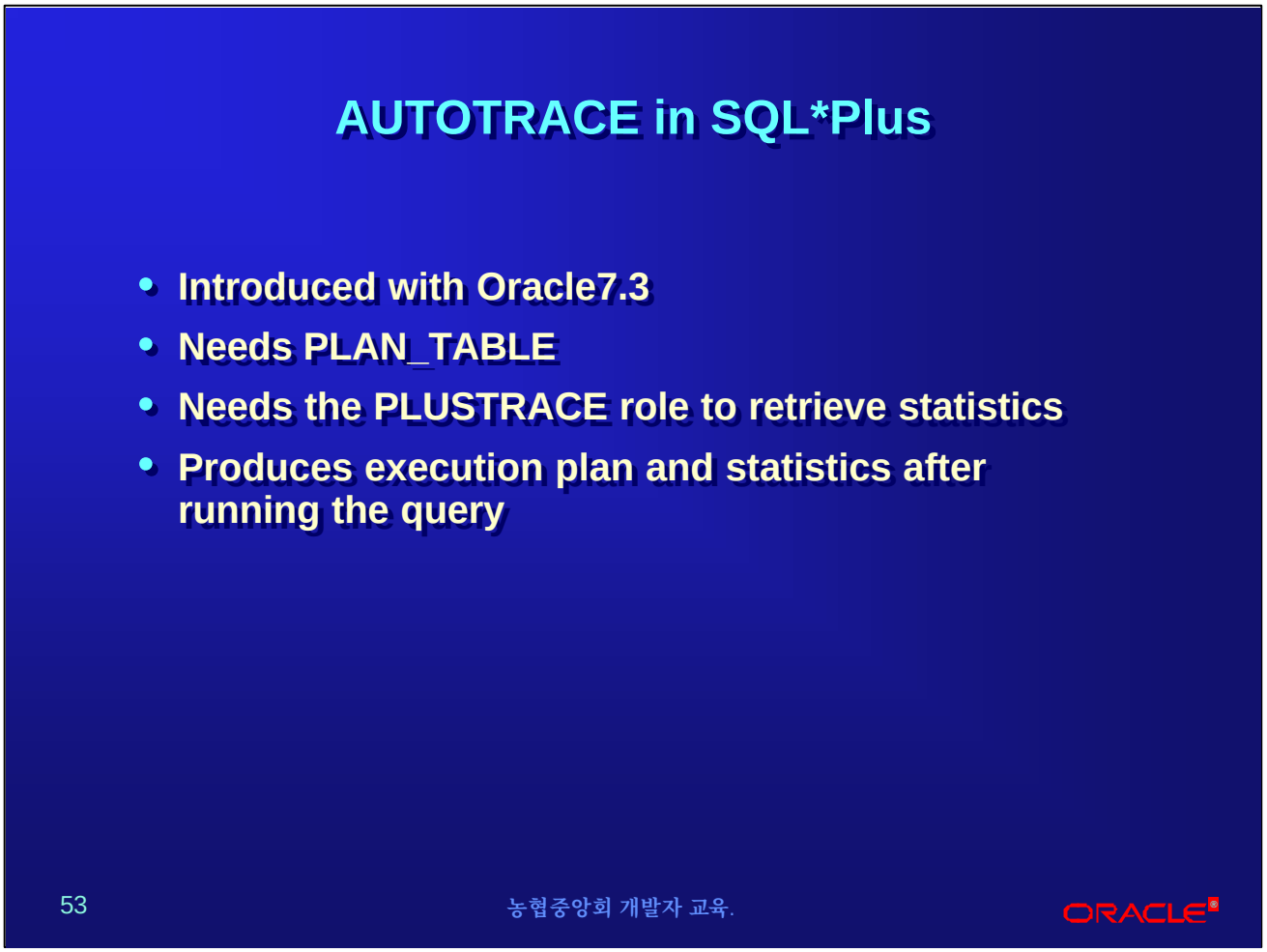


Execution Plan

This diagram shows the operation described in the previous example.

In this representation it is clearer how row sources generate rows that in turn are processed by other row sources before the final result set of rows is returned (from the nested loops join operation).

It is also clearer that the rows originate from leafs in the tree and then are sent upwards in the tree.



AUTOTRACE in SQL*Plus

The PLAN_TABLE table has to be accessible under the current schema to use AUTOTRACE.

The PLUSTRACE role is created and granted to the DBA role by running the supplied script `$ORACLE_HOME/sqlplus/admin/plustrce.sql` (on NT this script can be found in `%ORACLE_HOME\sqlplus\admin`). On some versions and platforms this is run by the database creation scripts. If this is not so on the actual platform (for example, 8.0.4 on NT), run `plustrce.sql` as SYS or from CONNECT INTERNAL.

The PLUSTRACE role contains select privilege on three v$ views; these privileges are necessary to generate AUTOTRACE statistics. The PLUSTRACE role can be granted manually to non-DBA users.

Note: AUTOTRACE spawns an additional database session, which can be seen from v$session.

Using AUTOTRACE

- **SET AUTOTRACE TRACEONLY**
  - Runs the query
  - Does not show query results
  - Shows statistics and execution plan
- **SET AUTOTRACE TRACEONLY EXPLAIN**
  - Shows the execution plan without statistics
  - Does *not* execute the query

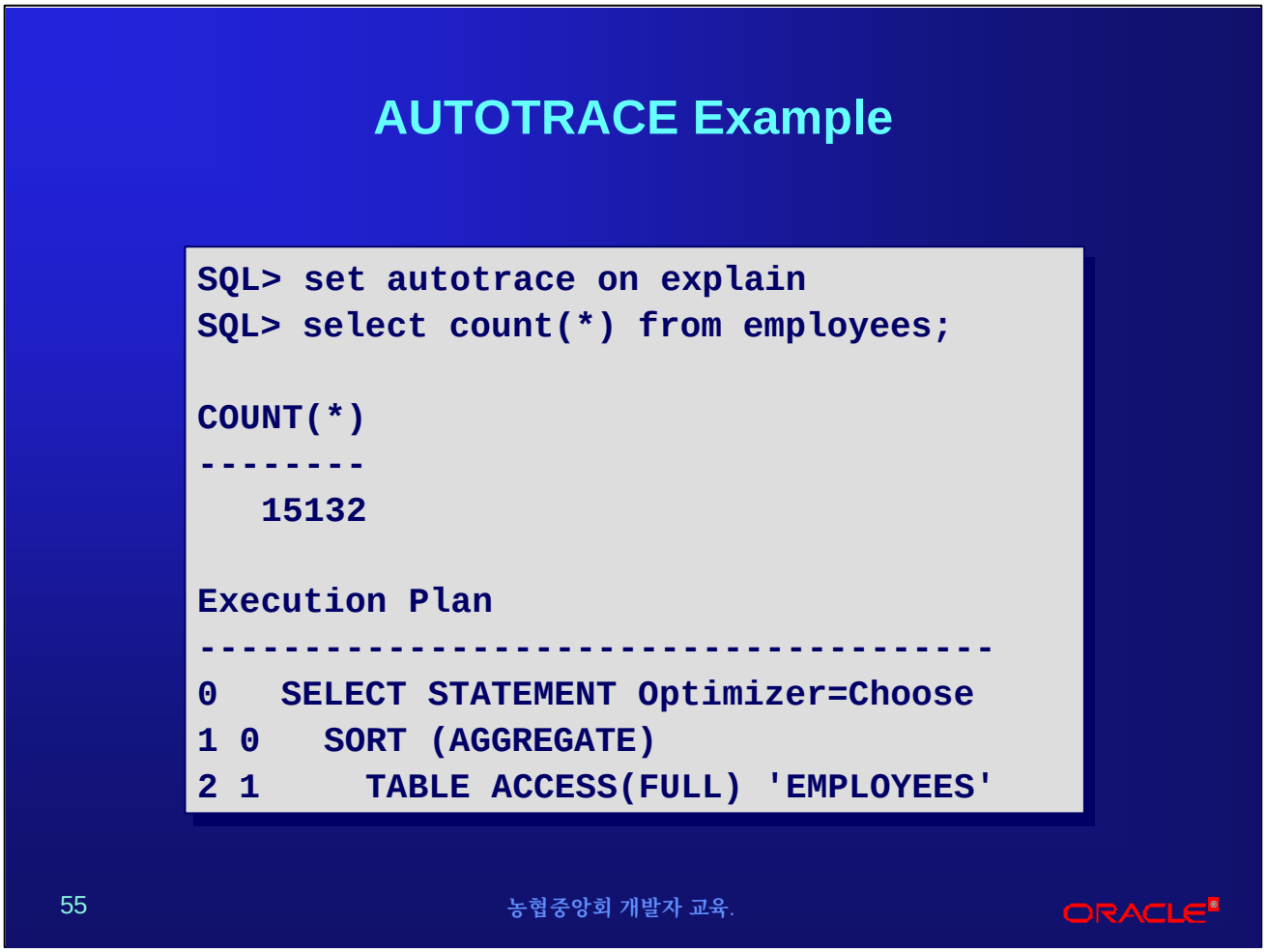
Using AUTOTRACE

Either the query execution plan, the statistics, or both can be displayed:

```
SET AUTOTRACE {OFF|ON|TRACEONLY} [EXPLAIN] [STATISTICS]
```

Using `TRACEONLY` with the `EXPLAIN` option avoids the overhead of executing the query. `TRACEONLY STATISTICS` does run the query, since it gives information about the resources used; there is an example in a later slide.

Note that all keywords of the SET AUTOTRACE command can be abbreviated, like all other SQL*Plus commands.



AUTOTRACE Example

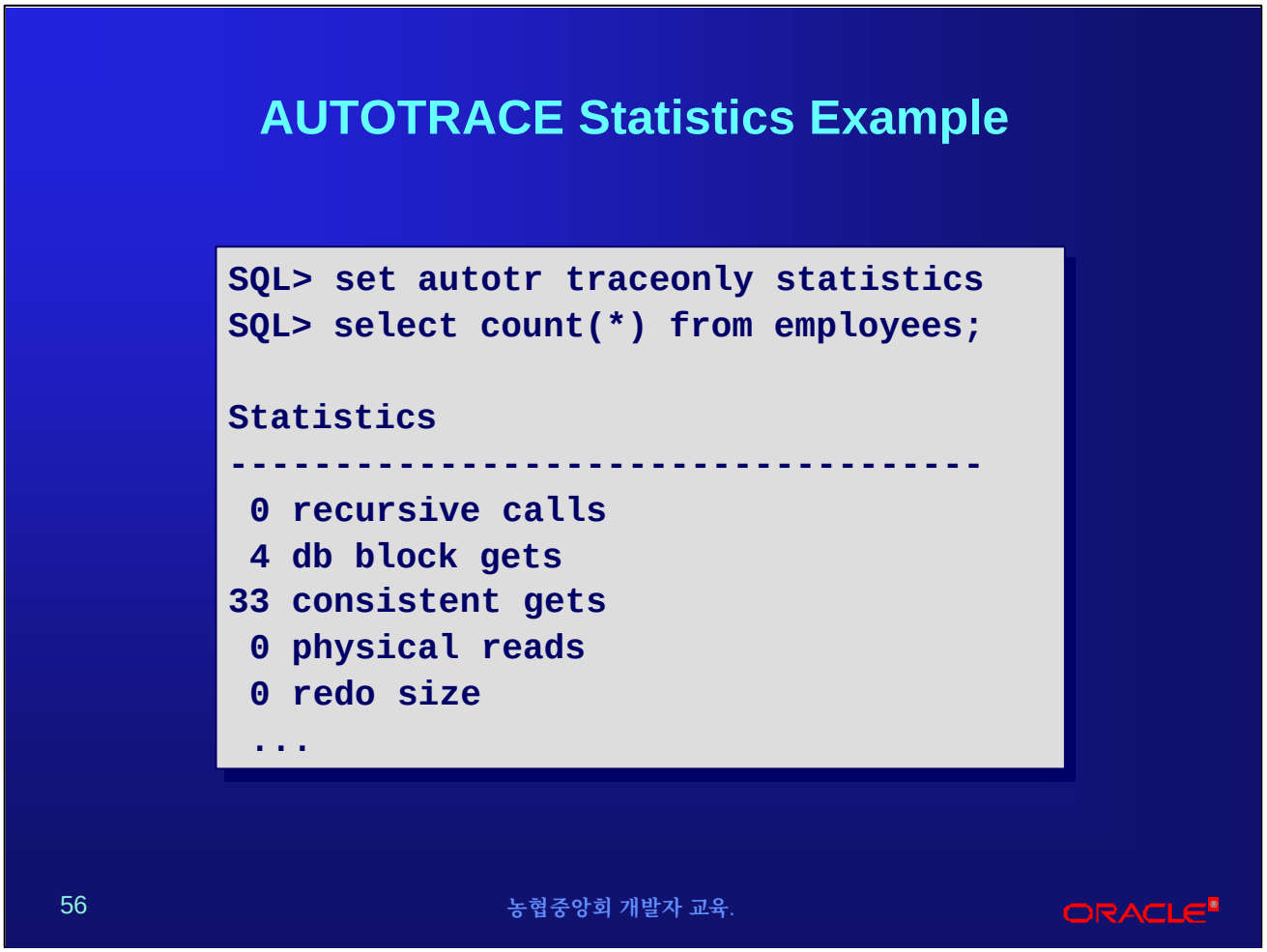
When using the cost-based optimizer, AUTOTRACE also displays the COST and CARDINALITY values for each plan step.

AUTOTRACE run against parallel queries shows the contents of the OBJECT_NODE, OTHER, and OTHER_TAG columns, as well as the kind of output shown above.

Influence the AUTOTRACE Output Layout

The AUTOTRACE output layout can be influenced by issuing the SQL*Plus COLUMN command against the following implicit columns (see the *SQL*Plus User's Guide and Reference*):

- ID_PLUS_EXP
- PARENT_ID_PLUS_EXP
- PLAN_PLUS_EXP
- OBJECT_NODE_PLUS_EXP
- D_PLUS_EXP
- OTHER_TAG_PLUS_EXP
- OTHER_PLUS_EXP



AUTOTRACE Statistics

The statistics shown by AUTOTRACE for a query are:

- Recursive calls
- Db block gets
- Consistent gets
- Physical reads
- Redo size
- Bytes sent via SQL*Net to client
- Bytes received via SQL*Net from client
- SQL*Net roundtrips to/from client
- Sorts (memory)
- Sorts (disk)
- Rows processed

Although the execution plan is usually the most useful part of the output, the statistics can also give tuning pointers. For example, the number of block reads (*db block gets* plus *consistent gets*) can show whether partitions have been eliminated.

Note: Previous versions of Oracle (with the exception of 8.0.6) all had a problem with reporting at least 1 sort in memory - even when there was no sort operation involved.

AUTOTRACE Statistics

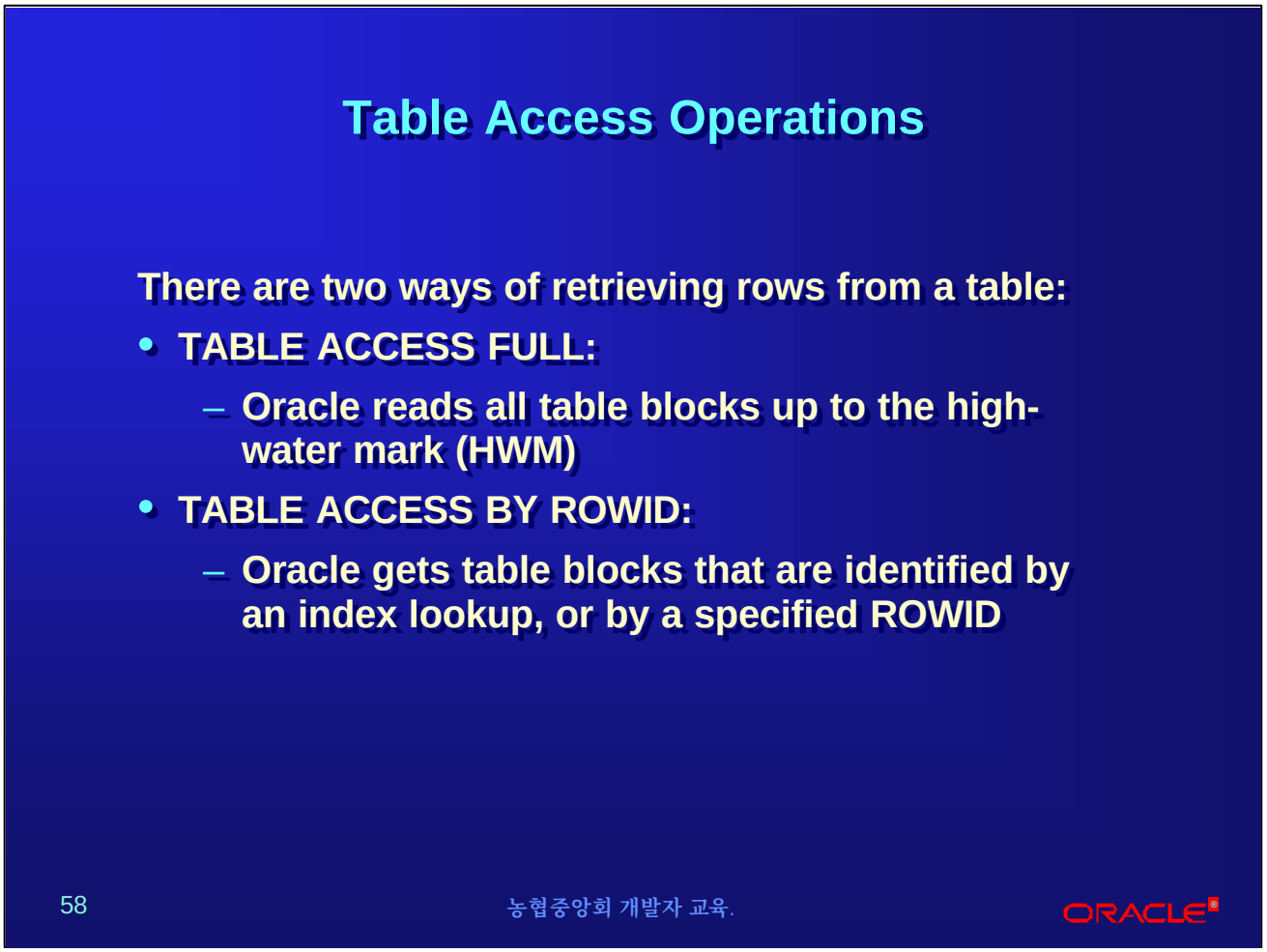
```
SQL> set pagesize 1000
SQL> set linesize 110
SQL> set arraysize 10
SQL> set autotrace on
SQL> select file_name from  dba_data_files;
.....11 rows selected .....

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         39  consistent gets
          0  physical reads
          0  redo size
        599  bytes sent via SQL*Net to client
        191  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         11  rows processed
```

- **Buffer Cache Hit ratio**
  = (Logical R – Physical R)/Logical R
  = (db block gets + consistent gets - physical reads) /(db block gets + consistent gets) = 39/39 * 100 = 100%

- **Bytes sent via Sql*net to client**
  = 38 + 29+430+102 = 599 bytes
- **Bytes received via Sql*net from client**
  = 90+32+62+7= 191 bytes
- **SQL*Net roundtrips to/from client**
  왕복 4번의 network traffics

- **Disk Sort Ratio**
  = sorts(disk) / (sorts(disk) + sorts(memory))
  =0/(0+0) = 0 * 100 = 0 %

Microsoft PowerPoint 프레젠테



Full Table Scans

This access method uses multiblock reads. You can set the *db_file_multiblock_read_count* parameter to regulate the number of blocks in a single read.

When the Oracle server reads blocks in a serial full table scan, they go to the least-recently-used (LRU) end of the buffer cache LRU list. For parallel scans, where the blocks do not already exist in the buffer cache, the blocks are read and written using direct I/O into the query slaves' PGAs.

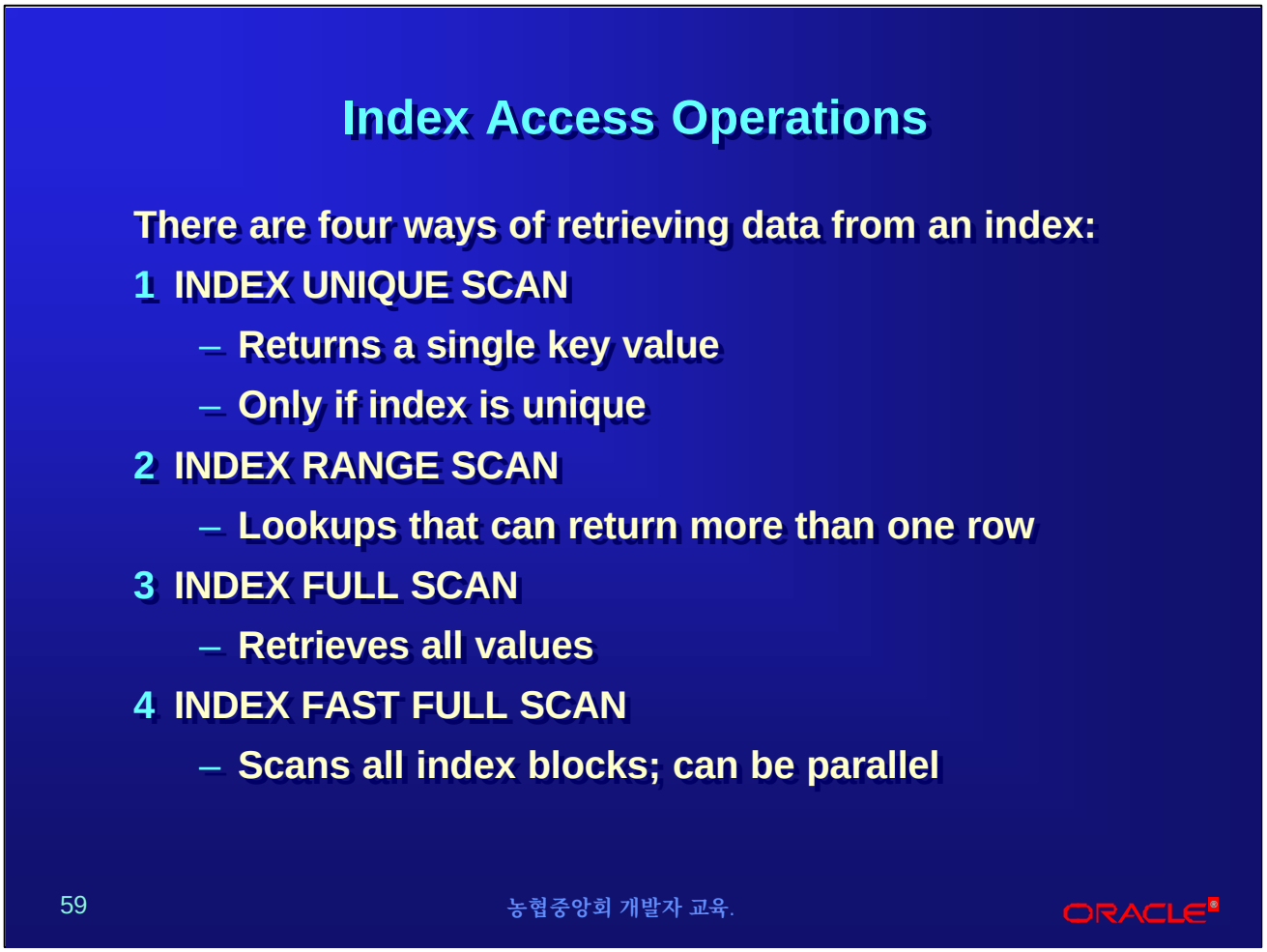
Full table scans can be efficient if they retrieve a large number of rows (more than about 10% of the total) or if they are executed in parallel. Otherwise, index access is usually quicker.

The high water mark is the last block ever used by the table. If rows have been deleted many of the table's blocks may be empty or partly empty, but they are still read.

Table Access by ROWID

Usually, the step before a ROWID access shows the index that is used. Occasionally applications supply the ROWID, so there is no index access step.

Note: This page and the following ones describe most steps seen in EXPLAIN output. For a complete list, see the *Oracle8i Designing and Tuning for Performance*, chapter 5.



Index Range Scan

This will be used for all index lookups using a nonunique index (as more than one row might be returned) and range predicates (for example BETWEEN, < or >)

Index Full Scan

Index full scans return the data in sorted order, and the CBO may decide that this is a cheaper access method than a full table scan followed by a sort. But these scans cannot use multiblock reads.

The Oracle server only uses an index full scan when at least one column in the index is not null (otherwise there is no guarantee that all rows will be present in the index.)

In earlier versions of Oracle (pre 7.3) the index full scan did exist for sorting operations, but was reported as an “index range scan” in an execution plan (as the full index scan is just an extreme index range scan).

Index Fast Full Scan

Index fast full scans are only available when using the cost-based optimizer.

Index fast full scans use multiblock reads and can be parallelized, but will not necessarily return the data in sorted order. Fast full scans, unlike other index scans, can access the second or subsequent column in a concatenated index, without the first column.

To use index fast full scans in Oracle7.3, the parameter *v733_plans_enabled* should be set to TRUE. However there are various bugs logged against this access method in this version; see for example Note 47742.1.

Join Operations

All join operations take place on two row sources. These are usually tables, but may also be rows produced by a previous operation in the plan.

Merge Joins

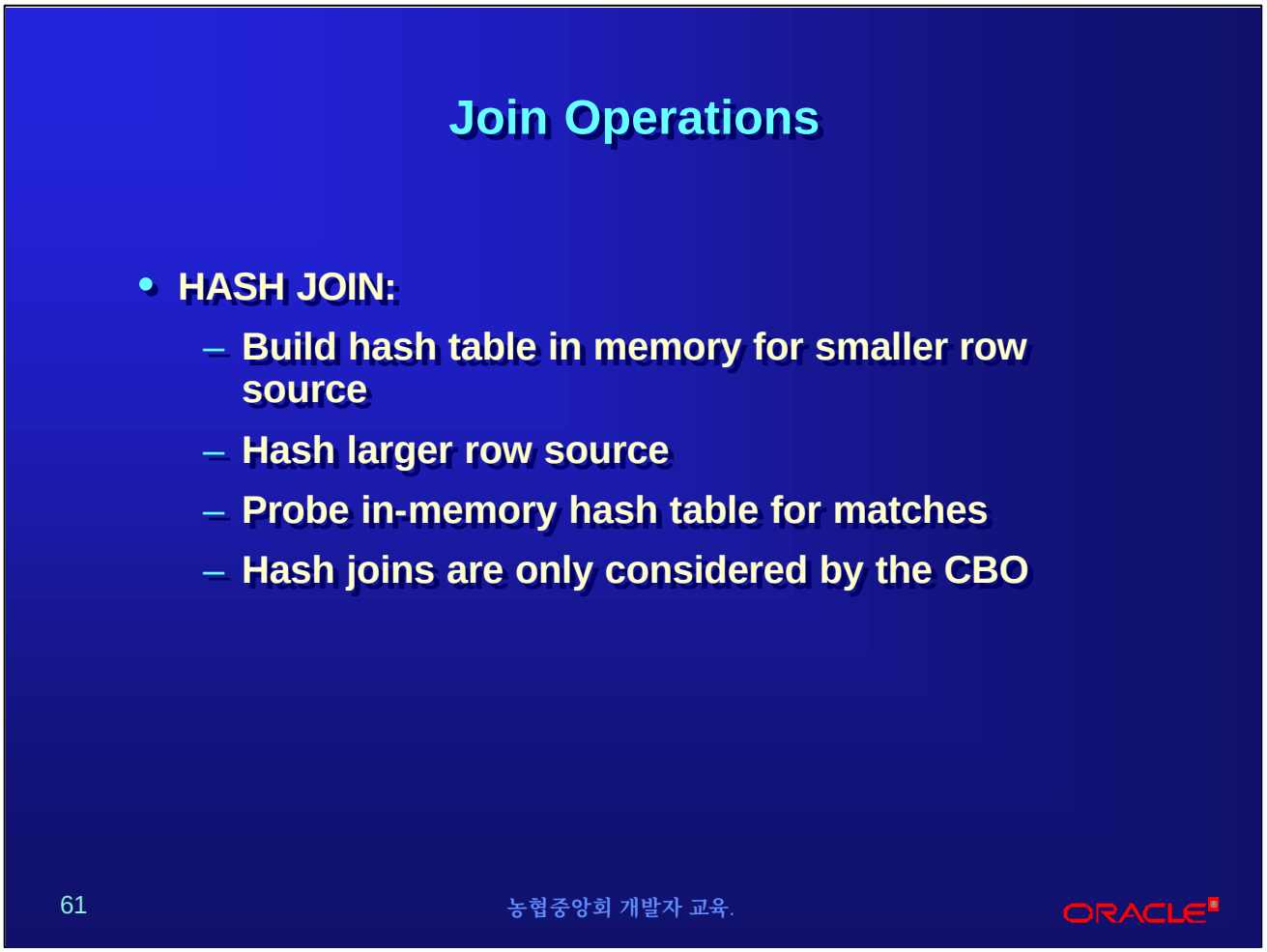
These are often described as sort-merge joins. The table data must be sorted before merging. There will usually be two sort steps just before the merge join in the plan, one for each row source. If the rows have been sorted earlier in the plan, or have been retrieved from an index, no sort step will be needed.

Until hash joins were introduced in Oracle7.3, sort-merge joins were the only practical way of joining large row sources. They are expensive, because sorts are expensive.

Nested Loops Joins

The first operation in a nested loop join is a row retrieval from the first row source, often called the outer, or driving, table. Then the Oracle server looks up the corresponding rows in the second table.

The outer row source should contain as few rows as possible. The retrieval method for the second row source should usually be an index lookup. If these conditions are true, nested loops can be very efficient.



Hash Joins

Hash joins are only available in Oracle7.3 onwards.

The example above applies when the whole of the smaller row source can fit into memory. Sometimes it is too big for available memory, and must be partitioned into separate hash tables. Oracle always builds a bitmap for the hash table and uses a quick lookup in the bitmap to check whether the row is currently in memory. It is a performance enhancement to avoid having to go to disk for rows that are not there.

Hash joins can be more efficient than any other join method, but this partly depends on the setting of the *hash_area_size* initialization parameter. The defaults is *sort_area_size* * 2. The *Oracle8i Data Warehousing Guide* suggests 1 MB as an absolute minimum (8 MB for data warehouses.) The hash area is part of the PGA.

Joins: Example 1

```
SQL> set autotrace traceonly explain
SQL> select crs.description, cls.status
  2  from   courses crs, classes cls
  3  where  crs.crs_id = cls.crs_id;
Execution Plan
----------------------------------------
0   SELECT STATEMENT Optimizer=RULE
1 0   MERGE JOIN
2 1     SORT (JOIN)
3 2       TABLE ACCESS(FULL) OF 'CLASSES'
4 1     SORT (JOIN)
5 4       TABLE ACCESS(FULL) OF 'COURSES'
```


Joins: Example 1

The first operation here is the full table scan of CLASSES. Then the Oracle server sorts the table on the CRS_ID column.

Next, the Oracle server performs a full table scan of COURSES and sorts that table on CRS_ID as well.

Finally, the sorted rows are merged, matching them on CRS_ID columns in both tables, and the columns required for the result are returned.

Joins: Example 2

```
SQL> set autotrace traceonly explain

SQL> select crs.description, cls.status
  2  from   courses crs, classes cls
  3  where  crs.crs_id = cls.crs_id
  4  and    cls.days = 5;

Execution Plan
----------------------------------------------
0   SELECT STATEMENT Optimizer=RULE
1 0  NESTED LOOPS
2 1   TABLE ACCESS(BY INDEX ROWID) OF 'CLASSES'
3 2    INDEX(RANGE SCAN) OF CLS_DAY (NON-UNIQUE)
4 1   TABLE ACCESS(BY INDEX ROWID) OF 'COURSES'
5 4    INDEX(UNIQUE SCAN) OF CRS_PK(UNIQUE)
```


Joins: Example 2

In this example (a similar one was discussed in earlier slides) an index lookup using the index on CLASSES.DAYS is used to find the classes that last five days.

Then the index on COURSES.CRS_ID is used to find the corresponding rows in the COURSES table and join them with the row retrieved from the first operation.

Other Typical Operations

- AND-EQUAL
  - Retrieves ROWIDs from two or more indexes
  - Returns the ROWIDs which occur in all indexes
- VIEW
  - View operation is executed first
  - Then rows are retrieved from the result set

AND_EQUAL

This operation is sometimes called an index merge. Where there are two (or more, up to five) equality conditions in the WHERE clause, and each column is indexed with an single column index, the Oracle server may search the indexes for matching ROWIDs. If the ROWID occurs in all the relevant indexes, it satisfies the conditions, and the matching row from the table can be retrieved.

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.

Sometimes a view is “nonmergeable” and must be executed independently of the user query against it, in a separate query block. For example, views containing GROUP BY functions are non-mergeable.

FILTER

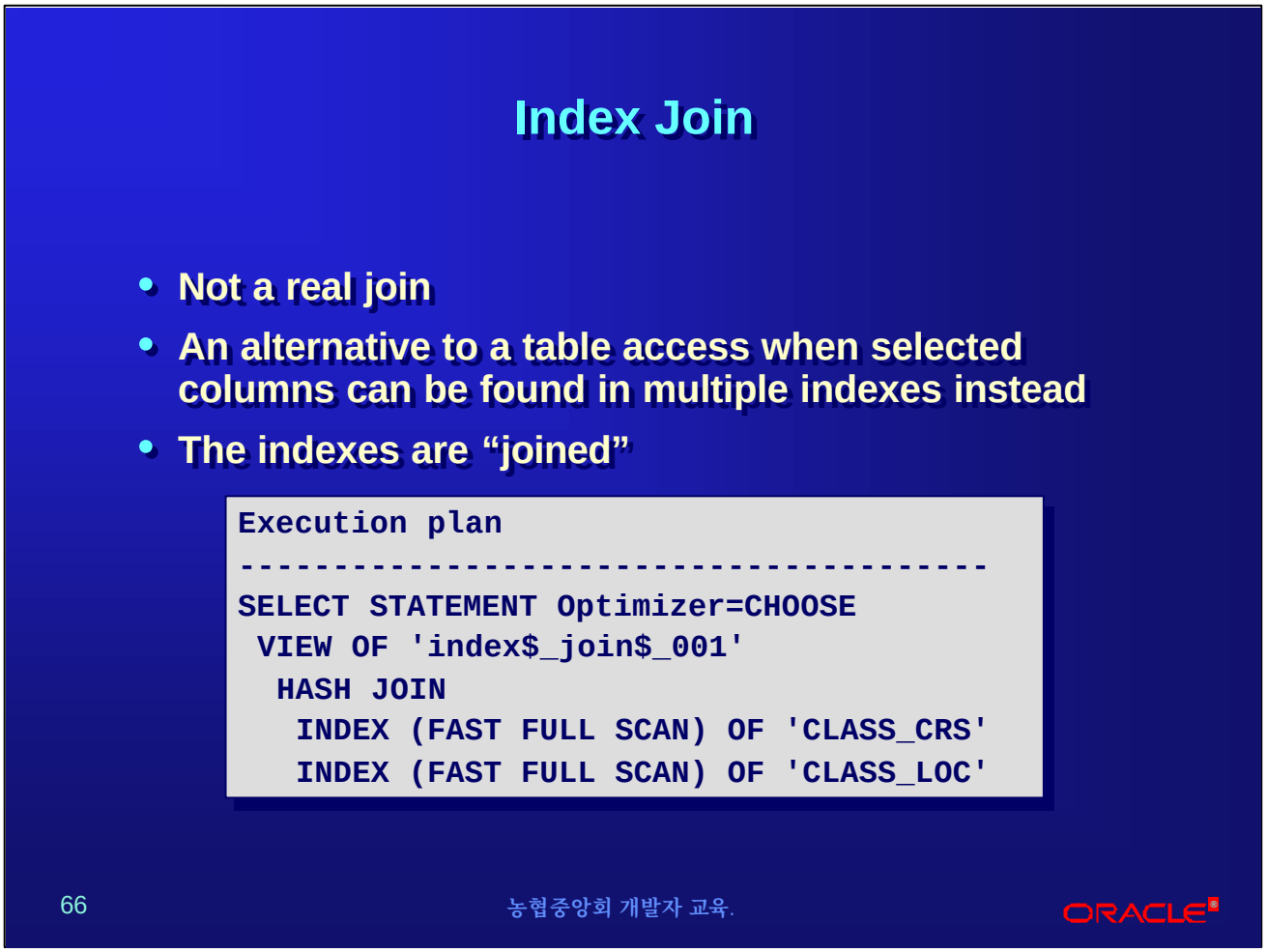
The FILTER step may indicate partition elimination in partition views, if it appears immediately after a PARTITION step. Otherwise, it means that a predicate that discards some of the rows returned from the previous step is applied. For example, here all managers that do not have at least 10 employees are discarded:

```
SQL> select mgr_id
  2  from employees
  3  group by mgr_id
  4  having count(*) >= 10;

EXECUTION PLAN
----------------------------------------------
0    SELECT STATEMENT Optimizer=CHOOSE
1 0   FILTER
2 1    SORT (GROUP BY)
3 2     TABLE ACCESS (FULL) OF 'EMPLOYEES'
```

SORT

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

The DISTINCT operator forces a UNIQUE sort. Users should be aware of the performance implications of adding unnecessary sorting.

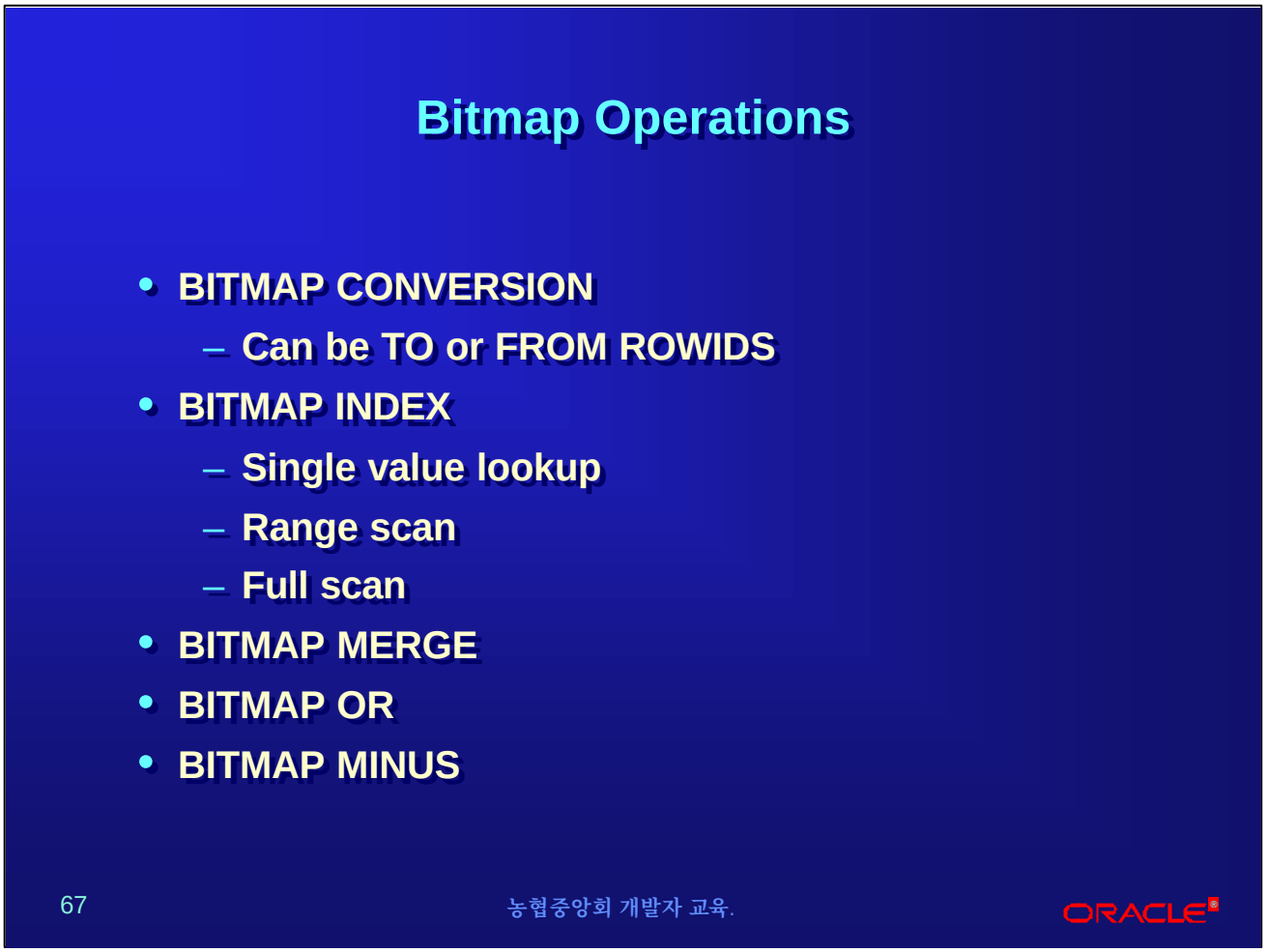


Index Join

The index join is not an operation in itself, but is built using index accesses and a join operation.

Example:

```
SQL> select /*+ index_join(c)*/
  2         loc_id, crs_id
  3  from   classes c;

Execution plan
----------------------------------------------------------
--
  0 SELECT STATEMENT Optimizer=CHOOSE
1 0   VIEW OF 'index$_join$_001'
2 1     HASH JOIN
3 2       INDEX (FAST FULL SCAN) OF 'CLASS_CRS' (NON-
UNIQUE)
4 2       INDEX (FAST FULL SCAN) OF 'CLASS_LOC' (NON-
UNIQUE)
```



Bitmap Operations

The Oracle server must convert bitmaps to ROWIDs to find the rows required. A FROM ROWID conversion is used less often to convert B*-tree index ROWID values to bitmaps so that they can be merged or compared with existing bitmap indexes.

Full or range scans can be used against bitmap indexes.

Most bitmap indexes are single-column ones, so if you have several WHERE conditions using the AND operator the Oracle server must combine, or merge, several bitmaps.

BITMAP OR is used with the OR operator.

BITMAP MINUS is used to subtract bits in one index from bits in another. For example, if using a condition such as:

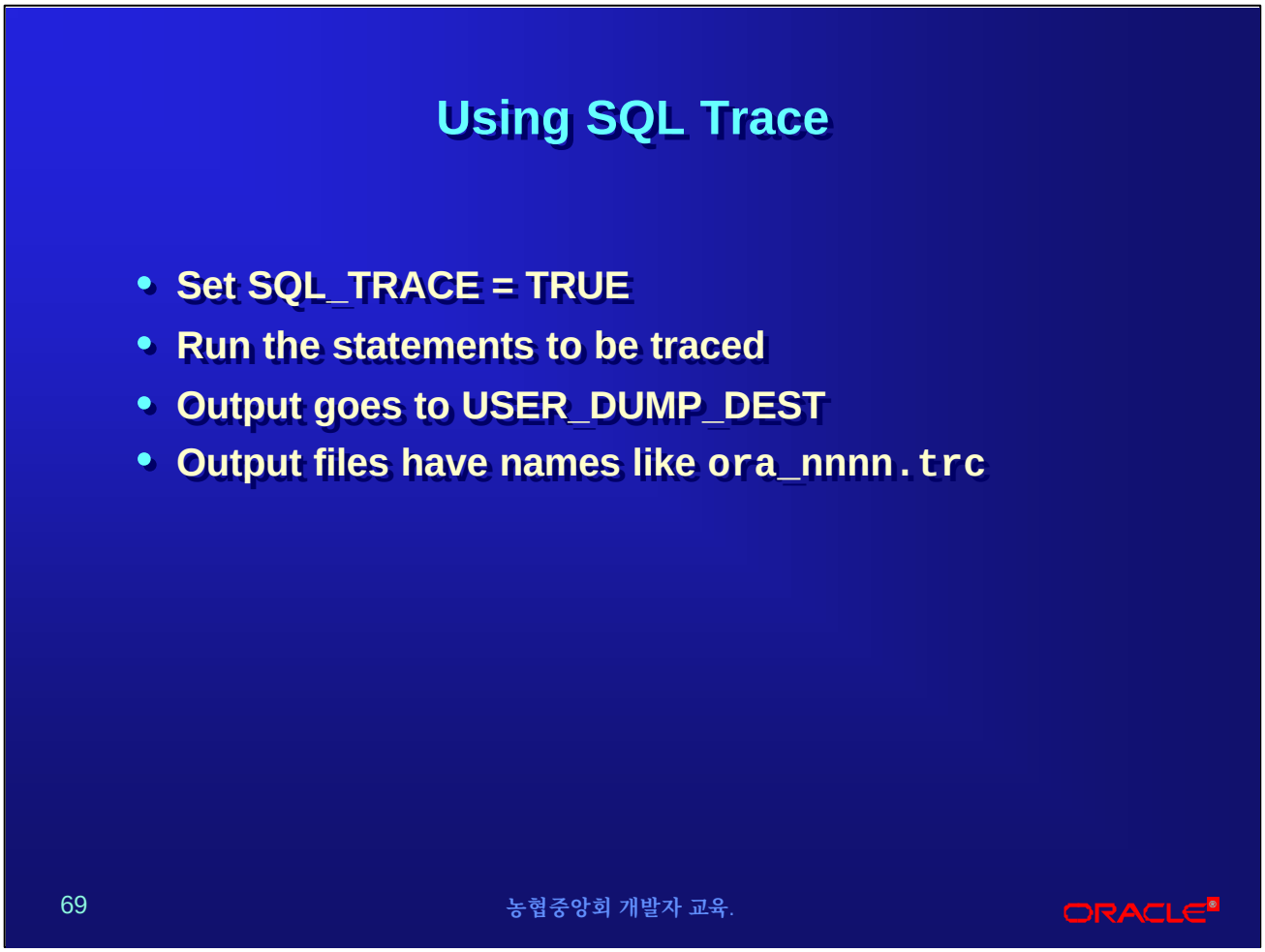
```
c1 = x AND c2 != y
```

The bits representing rows where `c2=y` are subtracted from the bits representing rows where `c1=x`.

Note: Fast full index scans cannot be performed on bitmap indexes.

SQL Trace

- The SQL trace facility logs information about a session's SQL and PL/SQL activity against the database.
- The output is a trace file generated on the database server.



Using SQL Trace

The output files are text files, but they are easier to read if you run the TKPROF utility against them. The next few slides deal with this utility and with trace interpretation.

Activating SQL Trace

- Can be set at instance or session level; session level usually more useful

```
SQL> alter session set sql_trace=true;
```

```
dbms_session.set_sql_trace()
dbms_system.set_sql_trace_in_session()
```

- Usually needs TIMED_STATISTICS to be set
- Important initialization parameters: USER_DUMP_DEST and MAX_DUMP_FILE_SIZE

Activating SQL Trace

You can turn tracing on at instance level by setting the *sql_trace* initialization parameter to TRUE, but this usually gives much more output than you need.

Using ALTER SESSION allows you to run and trace only the statements that you are currently concerned with.

Tracing can also be turned on from a PL/SQL environment by using the `dbms_session.set_sql_trace` procedure.

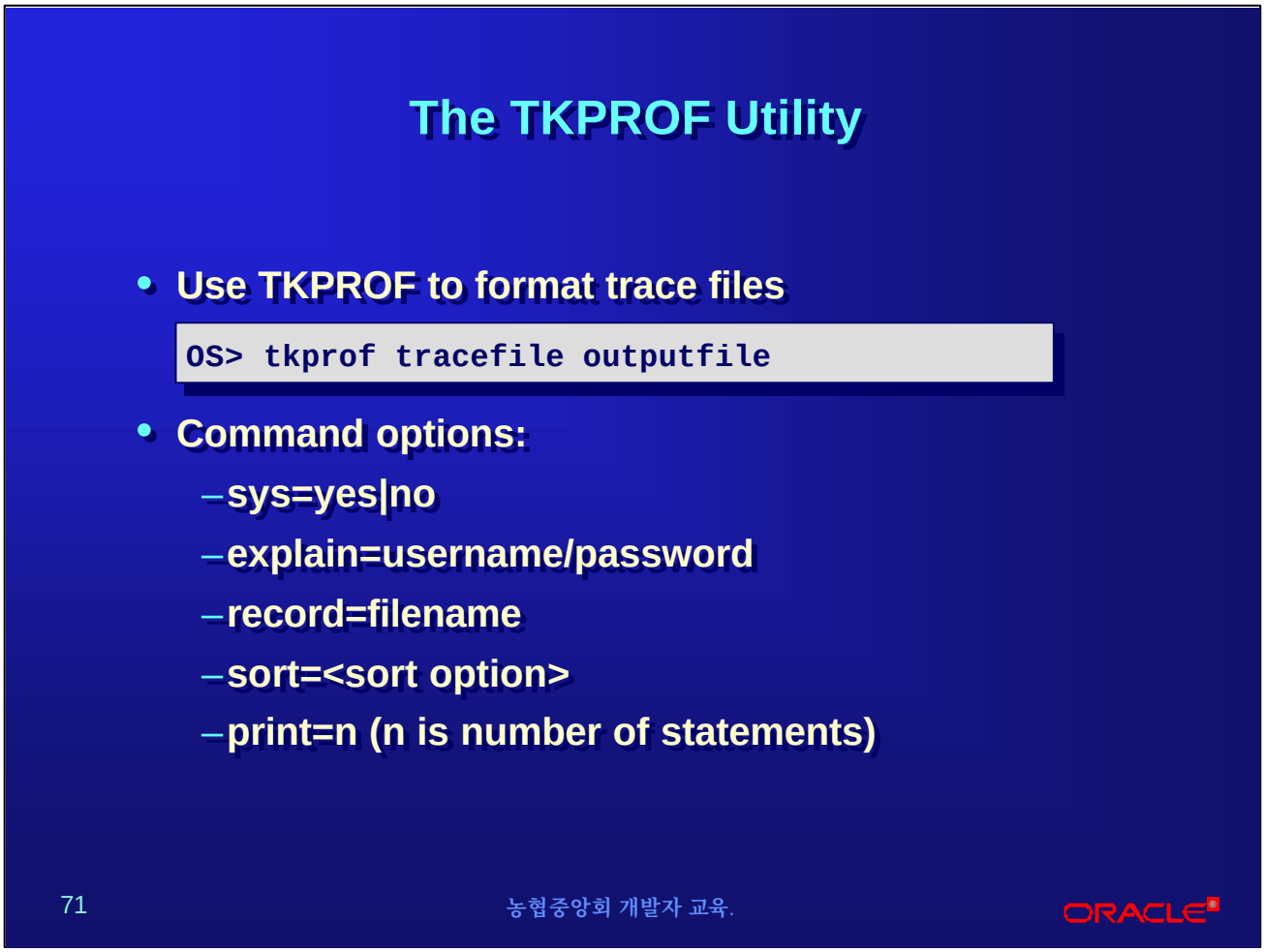
In some cases it is not possible to execute an “alter session” command. This can happen when tracing a C program where you do not have the possibility to change the source code and recompile.

In these cases you can either use SQL trace for the whole instance or use the `set_sql_trace_in_session` procedure of the `dbms_system` package.

Output is much more meaningful if you use TIMED_STATISTICS. You can set this at session or instance level:

```
SQL> alter session set timed_statistics = true;
```

Note: On HP/UX the system call to get the current time from the operating system currently takes longer than on other operating systems; on that platform, the CPU usage for database operations increases by approximately 10% when TIMED_STATISTICS is enabled.



The TKPROF Utility

Sys	Yes (the default) traces all recursive SQL used your statement; setting sys = no omits this. Typically you are not interested in the underlying SQL that is performed in the background. Note that sys = no does not exclude SQL from PL/SQL although this is also labelled "recursive."
Explain	Runs EXPLAIN under the specified username, and prints the EXPLAIN output together with the SQL statements. The users need not currently have a PLAN_TABLE, but must have the privilege to create it. Note that the execution plans are the ones that would be chosen when TKPROF was run.
Record	Records all the traced SQL statements in a file, so that you can rerun them later for comparison. Note that this does not include the content for bind variables, so you would still have to modify statements with bind variables to make them run.
Sort	Allows you to sort the output file in order of resource used. For example: sort = exeela orders by elapsed time during execution; sort = fchcpu execpu orders by fetch cpu time and execution cpu time, whichever is the largest.
Insert	Generates a script to insert all statistics back into a database table, for more flexible retrieval and reporting.

All sort options, and some less commonly used tkprof options, are listed in *Oracle8i Designing and Tuning for Performance*, chapter 6.

TKPROF Example: Setup

- Go to user_dump_dest
- Find the trace file (by date and time)
- tkprof tracefile newfile sys = no explain = un/pw

```
SQL> alter session
  2  set timed_statistics=true;
SQL> alter session set sql_trace=true;
SQL> select count(*) from classes;
SQL> alter session set sql_trace=false;
SQL> exit
```


TKPROF Example: Setup

TKPROF keeps timing information if the parameter *timed_statistics* is set to TRUE, but only to hundredths of a second. So times are not necessarily accurate for very small operations.

Note the ALTER SESSION SET SQL_TRACE = FALSE command.

This statement is not absolutely necessary before exiting from SQL*Plus—the tracing will stop as soon the session is exited—but SQL*Plus does not perform an explicit close on its main cursor before exit and this makes important information missing from the trace file.

This information is the number of rows from the individual row sources in the execution plan. This information—which when present is included in the TKPROF output—is only written to the trace file when a close is performed on the cursor or when a new statement is reparsed in the cursor.

As you can not explicitly close a cursor in SQL*Plus the only way to get the important information written to the trace file is to parse a new statement.

This statement can be anything—"alter session", as in the above example—or a simple "select * from dual" after the SQL statement of interest.

TKPROF Example: Output Excerpt

```
call    count  cpu  elapsed disk query current rows
------- ------ ---- ------- ---- ----- ------- ----
Parse       1 0.01    0.01    0     0       0    0
Execute     1 0.00    0.00    0     0       0    0
Fetch       2 0.09    0.12   73    73       4    1
------- ------ ---- ------- ---- ----- ------- ----
total       4 0.10    0.13   73    73       4    1
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 28 (SST)
Rows  Execution Plan
----  -------------------------------------------
    0 SELECT STATEMENT     GOAL: CHOOSE
    0  SORT (AGGREGATE)
11163   TABLE ACCESS GOAL:ANALYZED(FULL) OF 'CLASSES'
```


TKPROF Example: Output Excerpt

Count	Number of calls for each call type
CPU	CPU time in seconds (always zero if *timed_statistics* are not set)
Elapsed	Elapsed time in seconds (also depends on *timed_statistics*)
Disk	Number of blocks read from disk
Query	Number of blocks read in consistent (query) mode. These may be current copies, or copies which have been reconstructed, using rollback, to reflect the SCN for the query. Oracle will not write to a block which is in query mode.
Current	Number of blocks read in current mode. This includes blocks you are writing, and also segment headers which you never get in consistent mode. If there are buffers already in the cache in current mode that you can use for this query, they are added to the “current” total.
Rows	Number of rows retrieved by this step—one in this case because you asked for a COUNT.

Event 10046

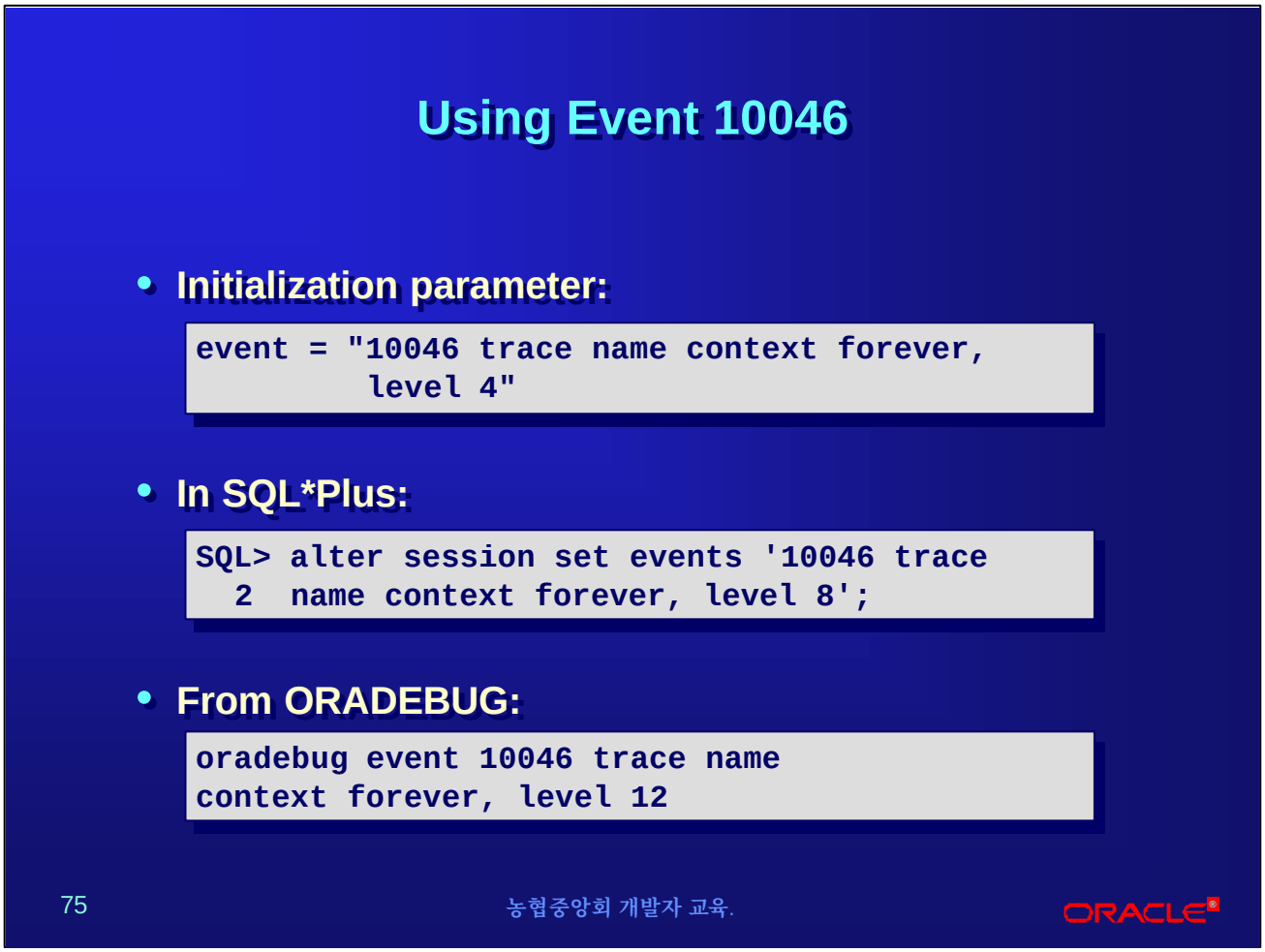
Setting event 10046 gives the same output as SQL trace when set at level 1.

Higher levels give more information:

- Level 4 includes bind variable values
- Level 8 includes wait events
- Level 12 includes both the above

Event 10046

If no level is specified for event 10046 it defaults to level 1.



Using Event 10046

Steps to use ORADEBUG:

```
svrmgrl
connect internal
```

When using ORADEBUG, it is necessary attach to the relevant process first, using

```
oradebug setospid <process id>
```

where the process ID is found in V$PROCESS.SPID.

Event 10046: Level 1 Output

- APPNAME: set with DBMS_APPLICATION_INFO
- PARSING IN CURSOR #n
- PARSE ERROR (if any)
- Operations; for example PARSE, FETCH
- ERROR: Errors at execute or fetch stage
- STAT #n
- XCTEND: Transaction end
  - rlbk = 1 if there was a rollback, otherwise 0
  - rd_only = 1 if transaction read only, else 0
- Higher levels have WAITS and BINDS

Event 10046: Level 1 Output

The parsing, operations, statistics, and XCTEND sections are covered in the next slides.

Level 1 output is exactly the same as an unformatted SQL_TRACE file.

You can use the TKPROF utility against trace files produced by the 10046 event even if you have used a higher level; TKPROF only omits the extra information.

Event 10046: Trace예

```
PARSING IN CURSOR #1 len=29 dep=0 uid=28 oct=42 lid=28
tim=3728235290 hv=3732290820 ad ='c4623380'
select count(*) from classes
END OF STMT
PARSE#1:c=1,e=1,p=0,cr=0,cu=0,mis=1,
        r=0,dep=0,og=4,tim=3728237882
EXEC#1: c=0,e=0,p=0,cr=0,cu=0,mis=0,
        r=0,dep=0,og=4,tim=3728237882
FETCH#1:c=9,e=12,p=73,cr=73,cu=4,mis=0,
        r=1,dep=0,og=4,tim=3728237894
FETCH#1:c=0,e=0,p=0,cr=0,cu=0,mis=0,
        r=0,dep=0,og=4,tim=3728237898
XCTEND rlbk=0 rd_only=1
STAT #1 id=1 cnt=0 pid=0 pos=0 obj=0 op='SORT AGGREGATE'
STAT #1 id=2 cnt=11163 pid=1 pos=1 obj=3050 op='TABLE
ACCESS FULL CLASSES'
```


Event 10046: PARSING IN CURSOR Output

#n	Cursor number
len	Length of statement in characters
dep	PGA depth (1 for recursive SQL, usually 0)
uid	Parsing user, as in DBA_USERS.USER_ID
oct	Command type, as for V$SESSION.COMMAND
lid	Privilege user
tim	Timestamp = value in V$TIMER when line written
hv	Hash value of the SQL statement, as in V$SQL
ad	Address of the statement, as in V$SQL

Event 10046: Operations Output

There is a separate line for each call.

The extra fetch always appears, even if the arraysize had been set higher than 1.

This is caused by the " piggybacked" fetch that automatically follows the execute (see the previous lesson, "Query Execution Overview").

The detail is similar to TKPROF output: c stands for cpu time, e is elapsed time, and so on.

This output also shows:

Dep	Call depth (0 for user, 1+ recursive)
Og	Optimizer goal
Tim	Timestamp for call

Values for the og (optimizer_goal) keyword are:

1-All_rows, 2-First_rows, 3-Rule, 4-Choose

Event 10046: Statistics Output

The statistics output refers to steps in the execution plan, as they would be shown in PLAN_TABLE.

The statistics output will only be written to the trace file when the cursor (in above example #1) is explicitly closed.

This is the information that TKPROF outputs as number of rows from each row source in the execution plan:

#1	The cursor number
Id	Identifier of this step in the plan
Cnt	Number of rows accessed
Pid	Identifier of the parent step in the plan
Pos	Plan_table position (order of processing if steps have the same parent ID)
Obj	Object ID of the row source, as in DBA_OBJECTS.OBJECT_ID for a table. Here the sort operates on the rows produced by the scan; this is not a persistent row source, so the object ID is 0 for this step
Op	The operation itself

Event 10046: Row count in different versions of Oracle

It is worth mentioning that the number of rows accessed (the cnt value) has changed somewhat between Oracle8 and Oracle8*i* in the sense that in Oracle8 (and Oracle7) the number was defined as the number of rows processed by a row source.

This meant that you could not directly identify, for example, the number of rows filtered away by a nonindexed predicate as the table access indeed had processed all rows.

Consider the statement:

```
SQL> select * from classes
  2  where  status = 'AVAI' -- Indexed predicate
  3  and    type = 'OSEC'   -- Applied after the index retrieval
```

In Oracle8 (and Oracle7) this would give the following TKPROF output:

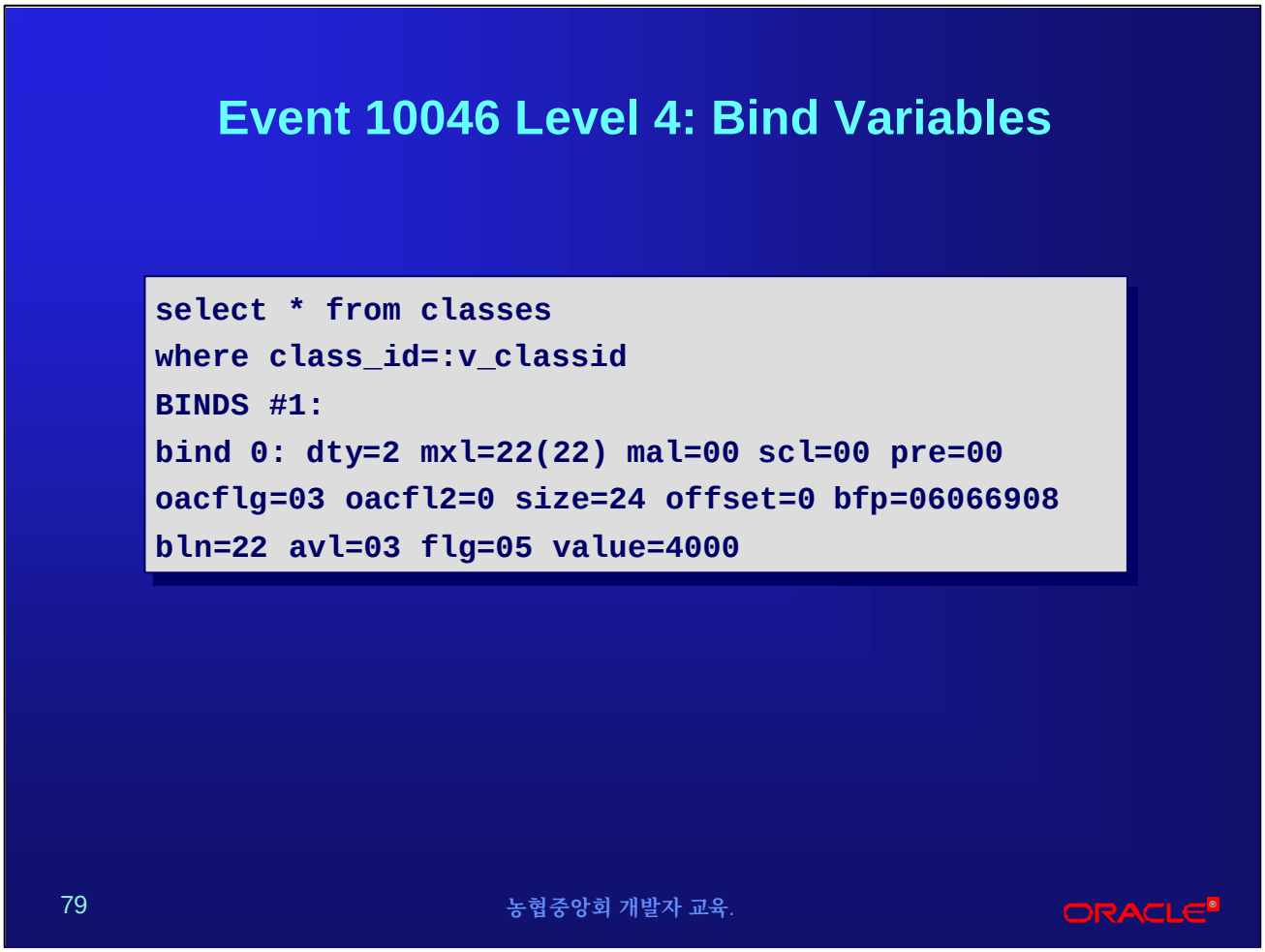
```
Rows  Execution plan
----- -------------------------------------------------
--
    0 SELECT STATEMENT GOAL: CHOOSE
  152  TABLE ACCESS GOAL: ANALYZED(BY INDEX ROWID) OF
CLASSES
  153   INDEX(RANGE SCAN) ON CLS_STATUS
```

Even if the unindexed predicate filtered away 151 of the 152 rows, leaving only one for output, this cannot be identified directly. (In this case you could look at the rows column in the statistical report preceding the execution plan in the TKPROF output.)

In Oracle8*i* the definition of the cnt column in the #STAT lines have changed more towards being the number of rows returned from the row source.

This means that the TKPROF output in Oracle8*i* for above example would be:

```
Rows  Execution plan
----- -------------------------------------------------
--
    0 SELECT STATEMENT GOAL: CHOOSE
    3  TABLE ACCESS GOAL: ANALYZED(BY INDEX ROWID) OF
CLASSES
  153   INDEX(RANGE SCAN) ON CLASS_STAT
```



Event 10046 Level 4: Bind Variables

The #1 represents cursor #1.

Bind values are given in order (0 for the first variable, 1 for the second, and so on).

Dty Data type; 2 = number, 1 = varchar2, 96 = char, date = 12

Mxl Maximum length of the bind variable, with private length in brackets. This variable was declared simply as a number, with no length specified.

Mal Array length

Scl Scale

Pre Precision

Oacflg/oacfl2/flg

Flags signifying bind type. See Note 39817.1 for more information.

Size Total size for all binds; only on first bind variable

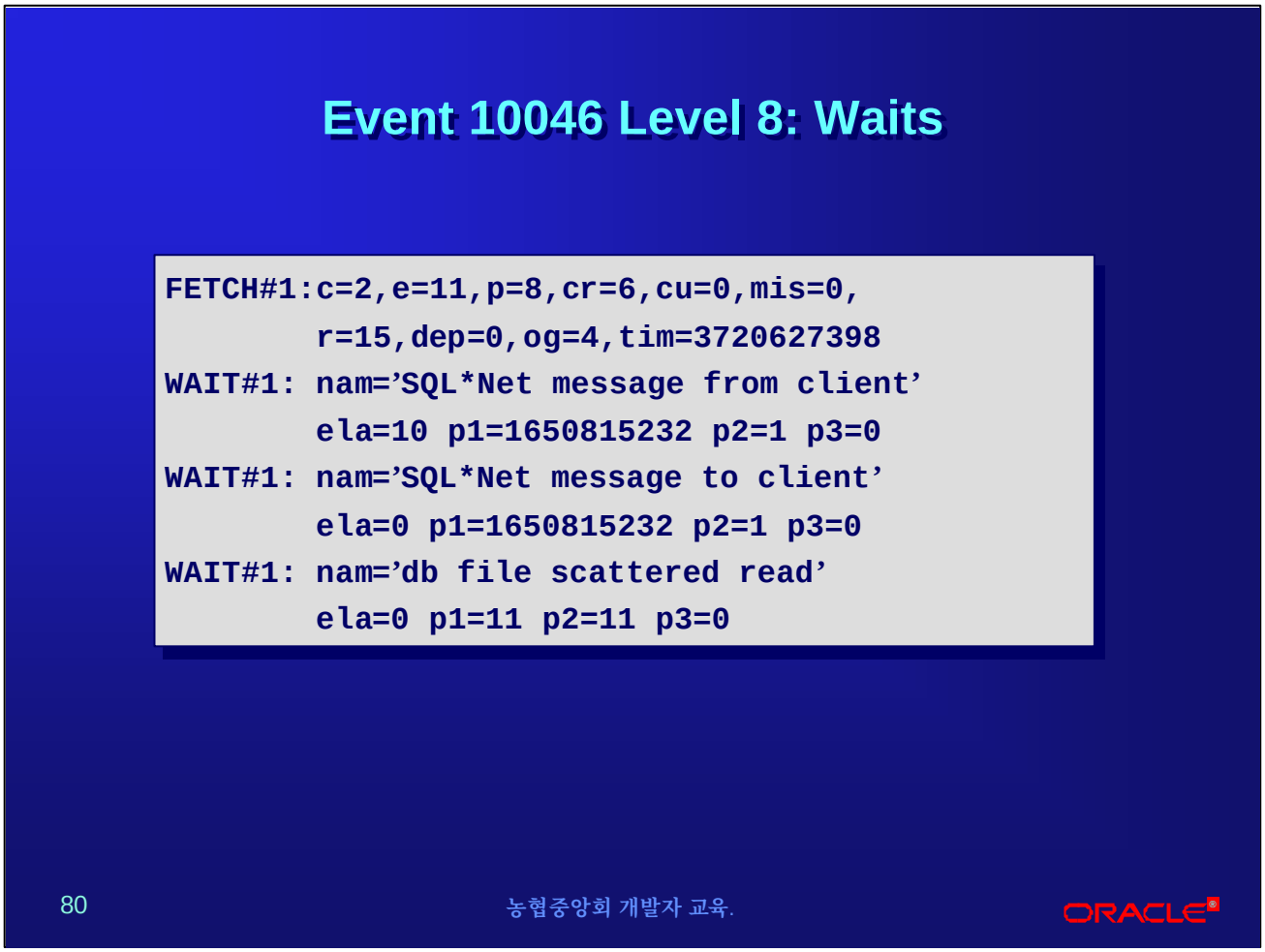
Offset Bind address offset from first bind variable

Bfp Bind address

Bln Bind buffer length

Avl Actual value length

Value Actual value of the bind variable



Event 10046 Level 8: Waits

Again, the #1 relates to the number of the cursor.

The wait names are as given in V$SYSTEM_EVENT, the parameters (p1 to p3) are as in V$SESSION_WAIT. Values are documented in the *Oracle8i Reference Release 2* (8.1.6); also in separate support notes, one for each wait event.

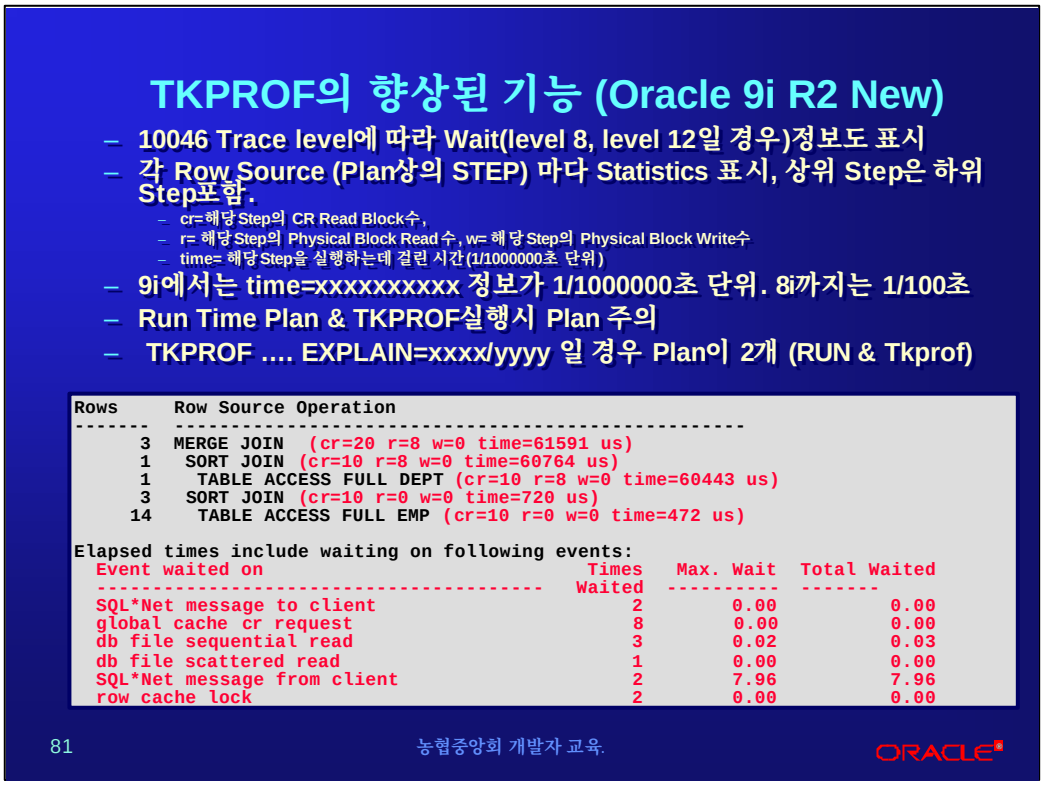
For the SQL*Net waits above, the first parameter is the address of the disconnect function of the current driver, the second is the number of bytes sent or received. This wait simply means that the server is waiting for something to be typed in at the client side.

The "db file scattered read" wait is for a multiblock read. The parameters represent the file number, the block number and the number of blocks to be read. It has this name because multiple blocks are read into into multiple buffers in the buffer cache; they are probably not adjacent, so they are scattered around the buffer cache.

"db file sequential read" shows that the read is using an index, or accessing a rollback segment.

Wait events can be very useful in diagnosis: for example, an operation may be waiting on a latch.

Note: Check Note 62327.1 for an internal Support tool called TRCSUMMARY that also formats information about wait events found in your trace files.




```
********************************************************************************

select *
from
 emp e,dept d where d.deptno = e.deptno and d.deptno = 10


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.03       0.09          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.01       0.06          8         20          0           3
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.04       0.15          8         20          0           3

Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 60  (SCOTT)

Rows     Row Source Operation
-------  ---------------------------------------------------
      3  MERGE JOIN  (cr=20 r=8 w=0 time=61591 us)
      1   SORT JOIN (cr=10 r=8 w=0 time=60764 us)
      1    TABLE ACCESS FULL DEPT (cr=10 r=8 w=0 time=60443 us)
      3   SORT JOIN (cr=10 r=0 w=0 time=720 us)
     14    TABLE ACCESS FULL EMP (cr=10 r=0 w=0 time=472 us)


Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
      3   MERGE JOIN
      1    SORT (JOIN)
      1     TABLE ACCESS (FULL) OF 'DEPT'
      3    SORT (JOIN)
     14     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMP'


Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                        2        0.00          0.00
  global cache cr request                          8        0.00          0.00
  db file sequential read                          3        0.02          0.03
  db file scattered read                           1        0.00          0.00
  SQL*Net message from client                      2        7.96          7.96
  row cache lock                                   2        0.00          0.00
```

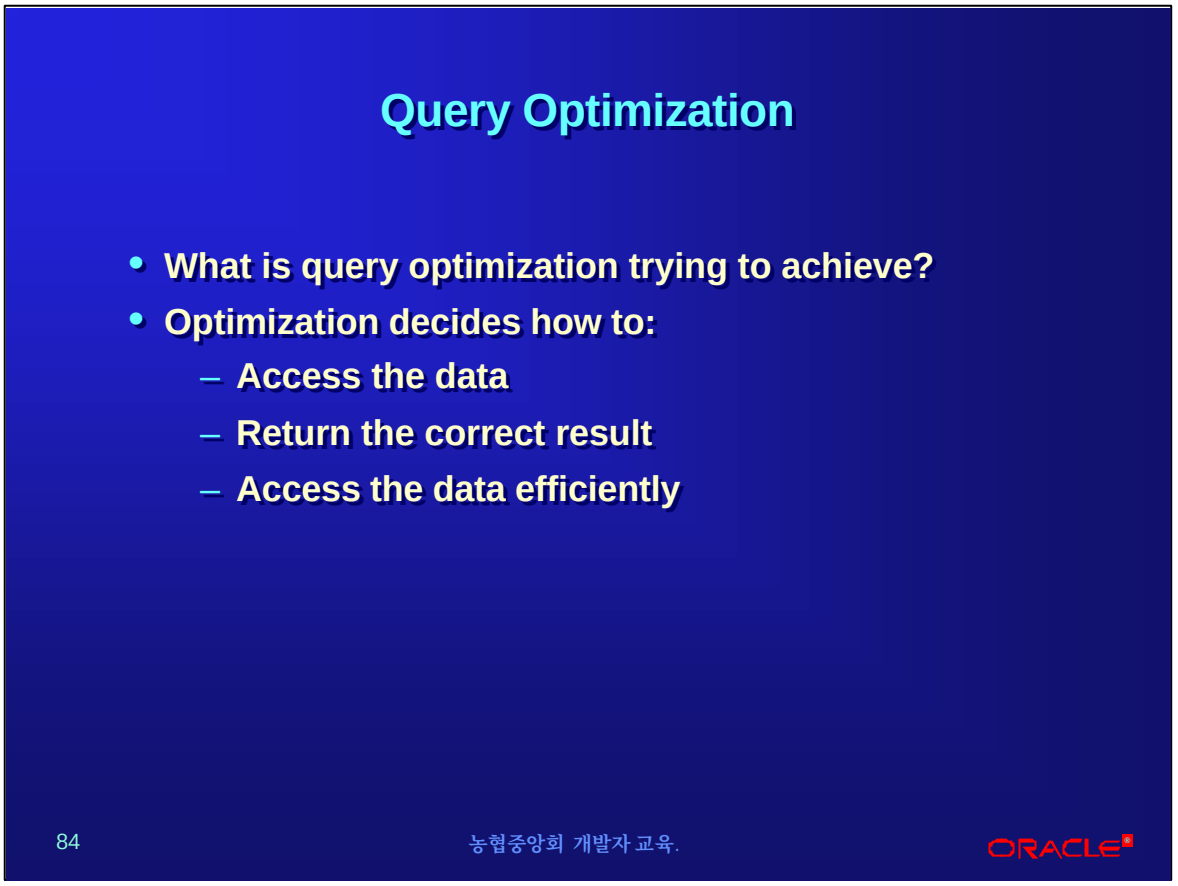
Cached Execution Plans(V$SQL_PLAN – 9i)

- v$sql_plan dynamic performance view
- 실제 Run Time시 실행된 plan정보
- PLAN TABLE과 항목이 거의 같다.

```
SELECT hash_value,
    (select sql_text from v$sql s
     where s.hash_value = p.hash_value and s.address = p.address
           and rownum <= 1),
    child_number,ID ,PARENT_ID ,
    LPAD(' ',2*(depth))||OPERATION||DECODE(OTHER_TAG,NULL,'','*')||
    DECODE(OPTIONS,NULL,'',' ('||OPTIONS||')')||DECODE(OBJECT_NAME,NULL,'','
OF '''||OBJECT_NAME||''''))||
DECODE(OBJECT#,NULL,'','(Obj#'||TO_CHAR(OBJECT#)||')')||DECODE(ID,0,DECODE(O
PTIMIZER,NULL,'',' Optimizer='||OPTIMIZER))||
DECODE(COST,NULL,'',' (Cost='||COST||DECODE(CARDINALITY,NULL,'','
Card='||CARDINALITY)||DECODE(BYTES,NULL,'',' Bytes='||BYTES)||')') SQLPLAN,
OBJECT_NODE, PARTITION_START ,PARTITION_STOP, PARTITION_ID, CPU_COST,
IO_COST, TEMP_SPACE, DISTRIBUTION, OTHER , ACCESS_PREDICATES ,
FILTER_PREDICATES
FROM v$sql_plan p
START WITH ID=0
CONNECT BY PRIOR ID=PARENT_ID AND PRIOR hash_value=hash_value AND
                     PRIOR child_number=child_number
ORDER BY hash_value,child_number,ID,POSITION
```

4

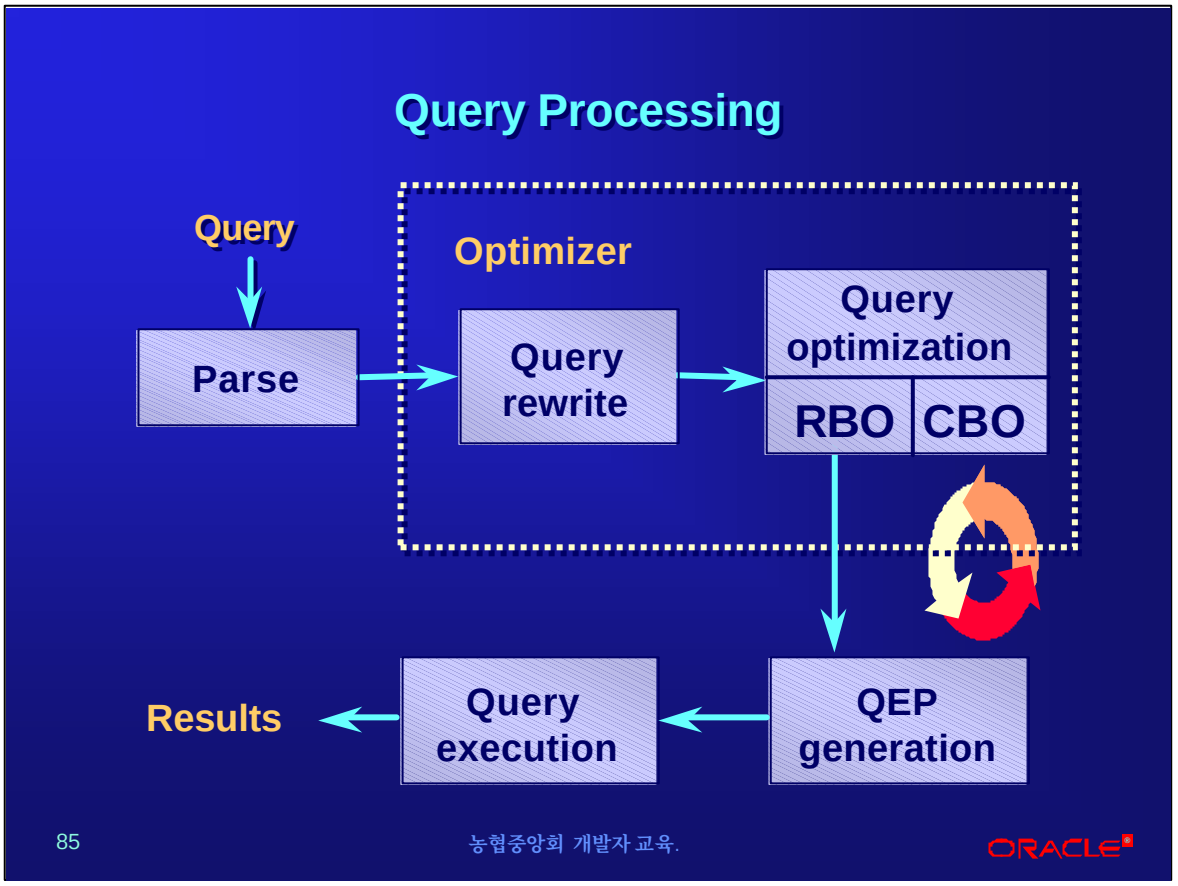
Optimizer(RBO,CBO) Basics



Query Optimization

Optimization is based on either hardcoded best practices (for the rule-based optimizer) or costs derived from gathered statistical information (for the cost-based optimizer). There is more information on this later.

Returning the correct result is clearly the most important factor in optimization. The result must be retrieved in a timely fashion though.



Query Processing

Generally everything up to query execution is thought of as parsing. In terms of the code, the main divisions fall as follows:

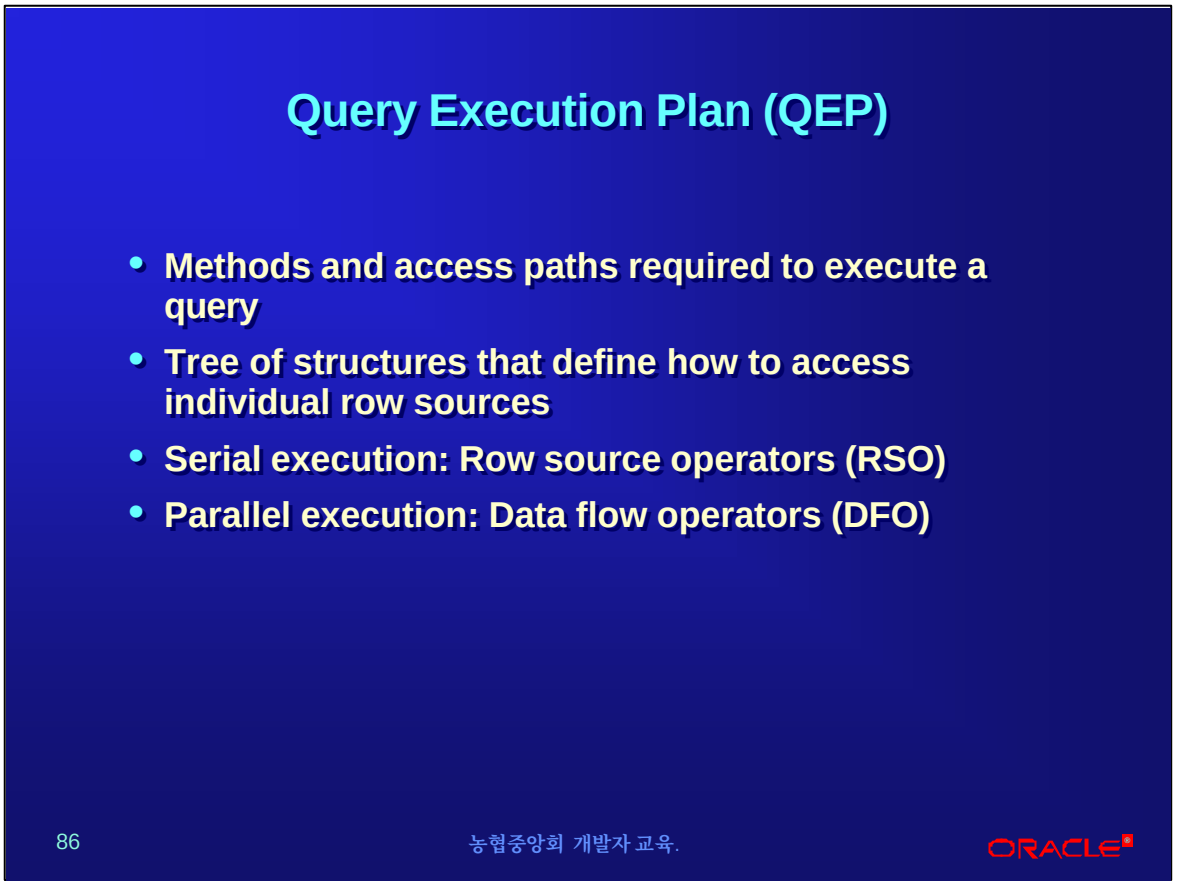
Optimization	Query rewrite and optimization
SQL Compilation	QEP generation
SQL Execution	Query execution

The sections perform the following functions:

Parse	Checks syntax, security, and semantics and does simple transformations
Query Rewrite	Merges subqueries and views, does OR expansion
Optimization	Determines access path for query
QEP Generation	Generates structure details for executing the query; this is known as the query execution plan or QEP
Query Execution	Executes the statement using the QEP

Before a query can be executed by a client process, it will have to bind values to any bind variables in the query. It will also have to know the number of columns and corresponding data types of the result set it will fetch. The server considers the execution complete when it is ready to return the first row (if any) to the client.

Some steps may be deferred to reduce network traffic or for other performance reasons.



Query Execution Plans

The optimizer generates a query execution plan (QEP) which characterizes the *retrieval strategy* for the query; that is, structures required to enable user processes to retrieve data from objects.

This plan is executed by the server process in order to retrieve rows. The plan is stored along with the query in the shared pool to prevent duplication of effort if the query will be executed several times by one or more processes.

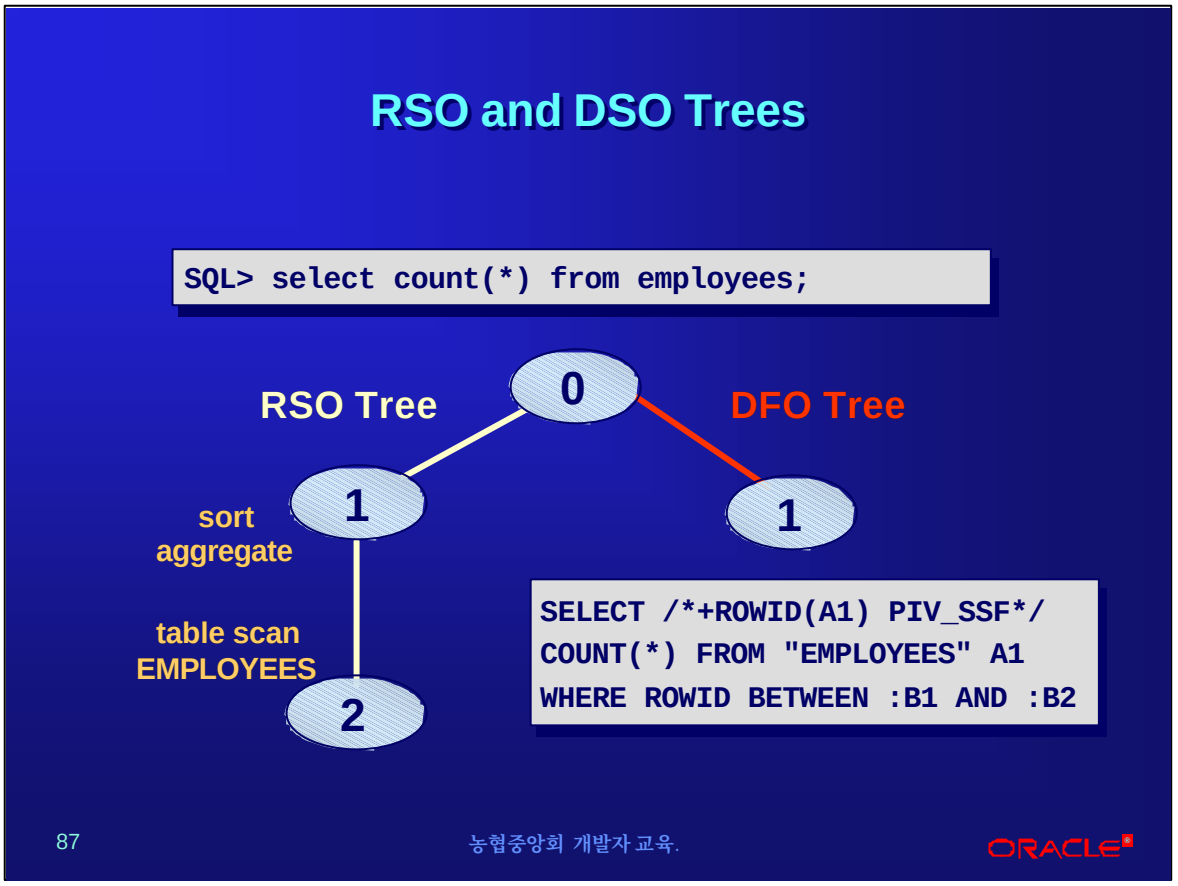
The QEP is produced in two forms to enable the Oracle server to execute the statement in serial or parallel:

- **RSO** = Row source operator (serial)
- **DFO** = Data flow operator (parallel)

If the optimizer has detected that the query cannot be parallelized then only an RSO tree will be produced. A DFO tree is not produced and the query will not be parallelized. If parallelism is possible then both RSO and DFO trees are produced. DFO trees will only be produced with the cost-based optimizer.

To execute queries in parallel special information on how the query can be parallelized is needed. This information is stored in the DFO tree; this is discussed in more detail within the “Parallel Query” lesson of this course.

QEP Execution: Once the QEP is built, it may be executed by instantiating and executing the RSO (serial) or DFO (parallel) tree.



RSO and DSO Trees

In the example

```
SQL> select count(*) from employees;
```

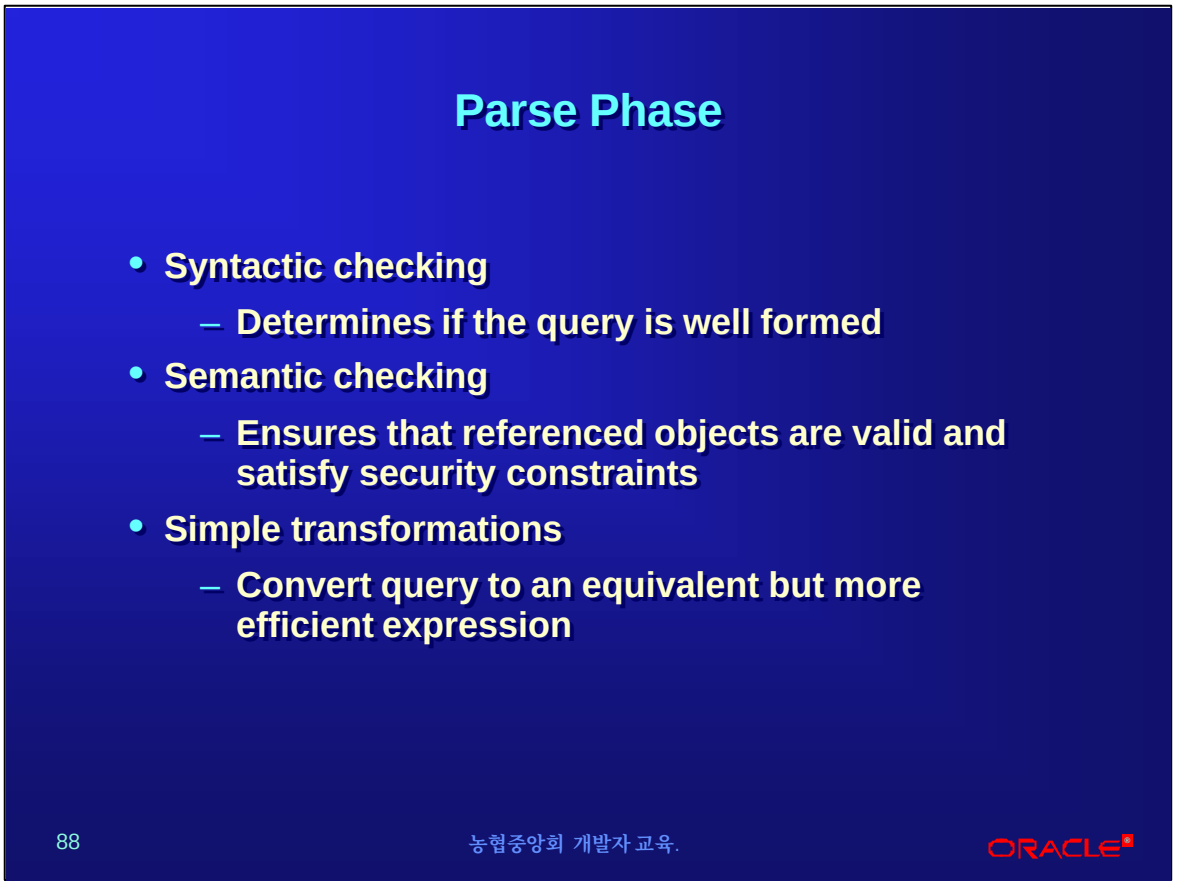
The following serial execution plan might be the result:

```
Query Plan
----------------------------
0-SELECT STATEMENT
1- SORT AGGREGATE
2-  TABLE ACCESS FULL EMPLOYEES
```

On the slide, the serial plan (RSO) is shown to the left and the parallel plan (DFO) to the right. At execution time, the query coordinator process divides work between the slave processes. Each slave receives the select statement shown and executes it with ROWID ranges provided by the query coordinator process. Notice that there is no mention of the sort aggregate operation; this is handled by the query slave and table queue structures.

If the select was parallelizable the Oracle server would produce a RSO and a DFO tree; if the select was not parallelizable only a RSO tree would be produced. If insufficient resources were available at execution time to achieve the requested degree of parallelism the RSO tree would be executed.

This is discussed more extensively in a later lesson.

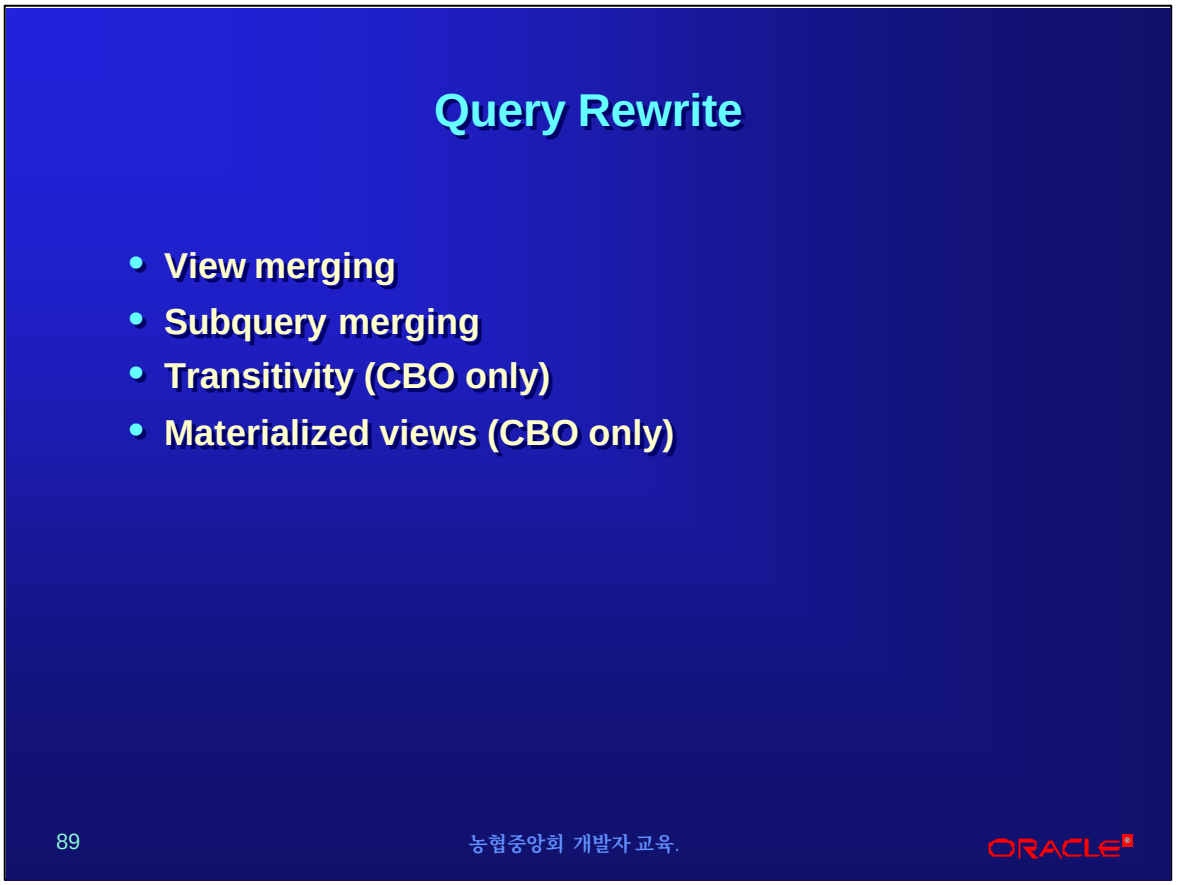


Parse Phase

The first phase does basic checking to ensure that the SQL statement is worth evaluating.

Some simple transformations:

Example Expression	Transformation
`lastname LIKE 'WARD'`	`lastname='WARD'`
`lastname IN ('KING','WARD')`	`lastname='KING' OR lastname='WARD'`
`lastname=ANY/SOME('KING','WARD')`	`lastname='KING' OR lastname='WARD'`
`mgr_id != ALL(1001,20023)`	`mgr_id != 1001 AND mgr_id != 20023`
`salary BETWEEN 2000 and 3000`	`salary >=2000 AND salary <= 3000`
`NOT(salary<1000 OR job is null)`	`salary >= 1000 and job is not null`



Query Rewrite

Rewriting queries is done to open up access paths that otherwise may be unavailable to the optimizer. This may encourage the use of a more optimal access path.

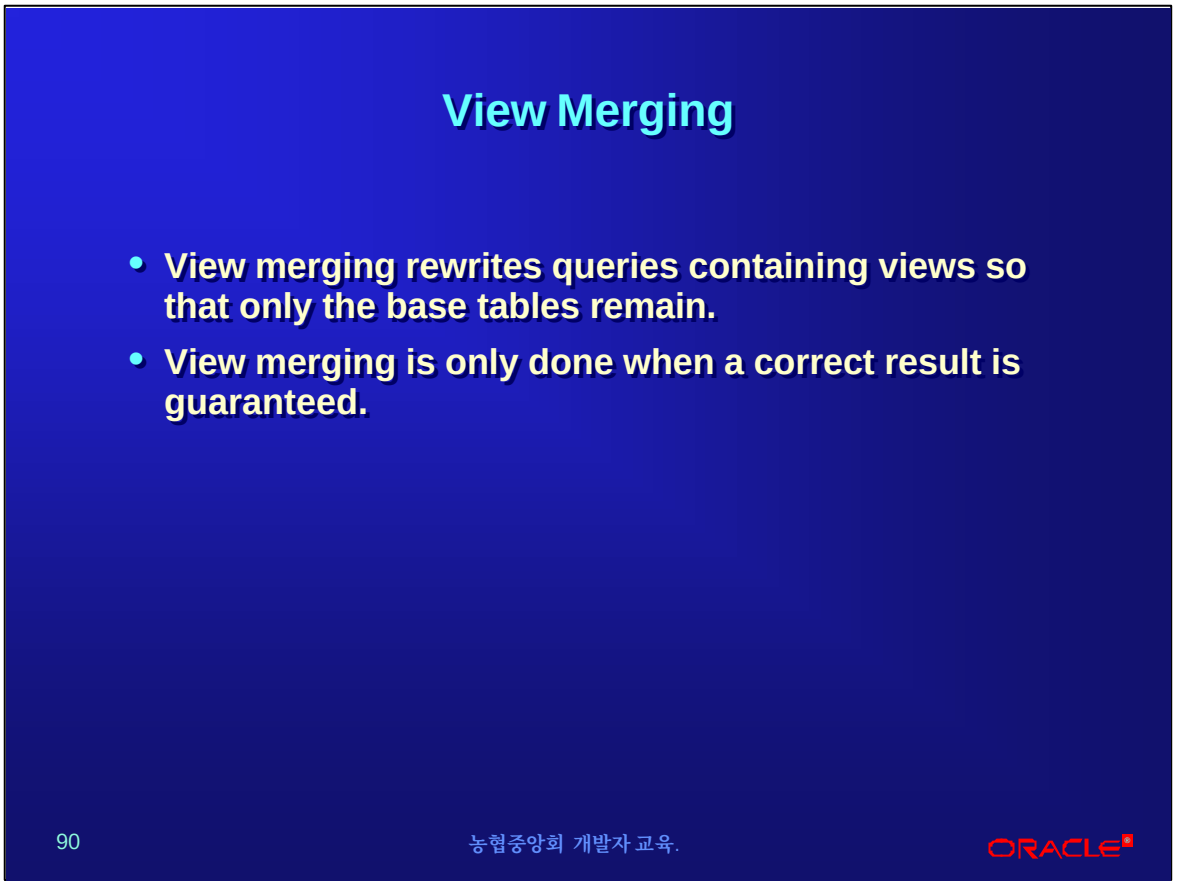
Merging of views, flattening of subqueries, and pushing of predicates into views are handled by both RBO and CBO optimizers. Manual partition functionality requires CBO.

It is useful to know that such rewriting takes place as it can explain why certain access paths have become available.

Transitivity is actually the first phase of the CBO's activities and is explained later.

Rewriting a query to use a materialized view actually means that the optimizer redirects the query from the set of tables originally specified in the query to another (redundant) data segment. This is described later.

Note: An understanding of what query rewrite goes on can help resolve query issues more quickly. Query rewrite may or may not enable index usage.

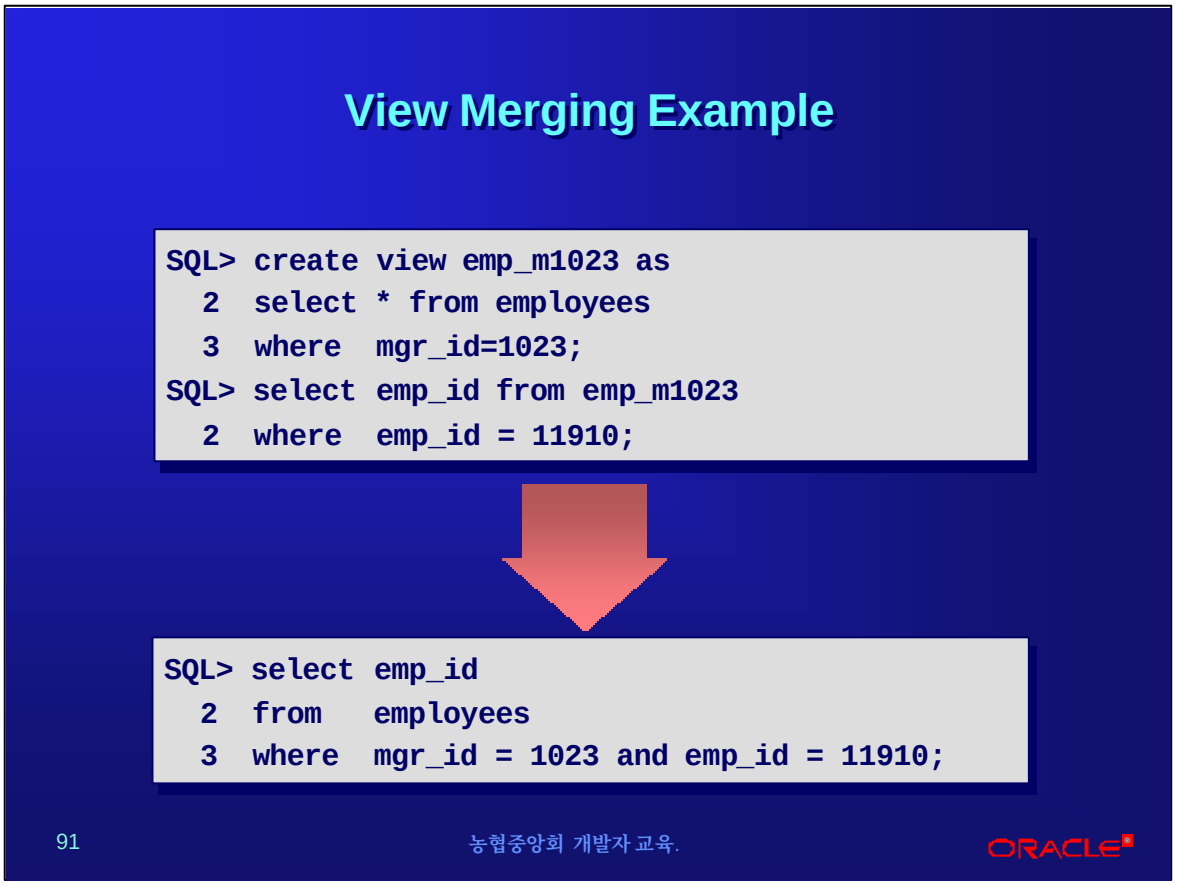


View Merging

The view is rewritten as a SELECT statement on the view's base tables. The SELECT statement is incorporated back into the original query, potentially opening new access path or join possibilities.

There are two ways of merging a view with a query. The contents of the view can be pushed out into the query or the contents of the query can be pushed into the view.

If a view cannot be merged then the select and the view must be executed as separate query blocks. In this case the execution plan will contain a VIEW operation.



View Merging Example

In this example, merging may allow the use of a concatenated index on the MGR_ID and EMP_ID columns. This index would not normally be used in the base query because only the second column is supplied.

This rewritten view text cannot be seen anywhere. It exists only within state objects.

If the view had not been merged, the execution plan within the VIEW operation would have shown that a potential unoptimized path had been used.

View Merging Example (continued)

For a more complex example of view merging consider following view definition:

```
SQL> create view v_cls as
  2  select crs_id, description, days, instr_id
  3  from   classes, class_types
  4  where  classes.type = class_types.type;
```

If the view is merged with an outer query:

```
SQL> select crs.short_name, cls.instr_id
  2  from   courses crs, v_cls cls
  3  where  crs.crs_id = cls.crs_id
  4  and    crs.days = 10;
```

The following execution plan is generated and as you can see the view is "broken down" into its elements and COURSES and CLASSES are joined together first:

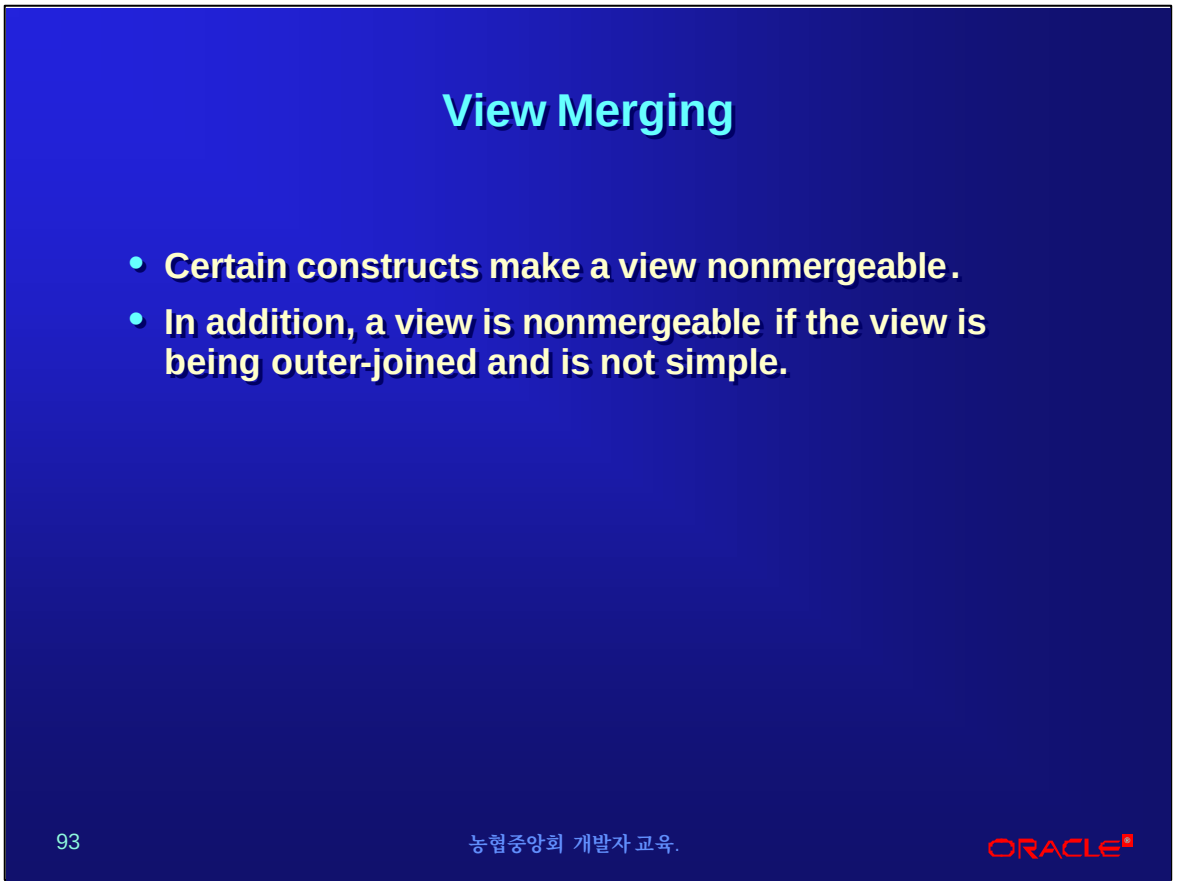
```
SELECT STATEMENT
  NESTED LOOPS
    HASH JOIN
      TABLE ACCESS (FULL) OF COURSES
      TABLE ACCESS (FULL) OF CLASSES
    INDEX (UNIQUE SCAN) OF CTYPE_PK
```

If view merging is disabled (in this case with a hint):

```
SQL> select /*+NO_MERGE(CLS)*/
  2         crs.short_name, cls.instr_id
  3  from   courses crs, v_cls cls
  4  where  crs.crs_id = cls.crs_id
  5  and    crs.days = 10;
```

Then the view is evaluated "as a whole" and the two tables in the view are joined together before being joined to the outer query:

```
SELECT STATEMENT
  HASH JOIN
    TABLE ACCESS (FULL) OF COURSES
    VIEW OF V_CLS
      NESTED LOOPS
        TABLE ACCESS (FULL) OF CLASSES
        INDEX (UNIQUE SCAN) OF CTYPE_PK
```



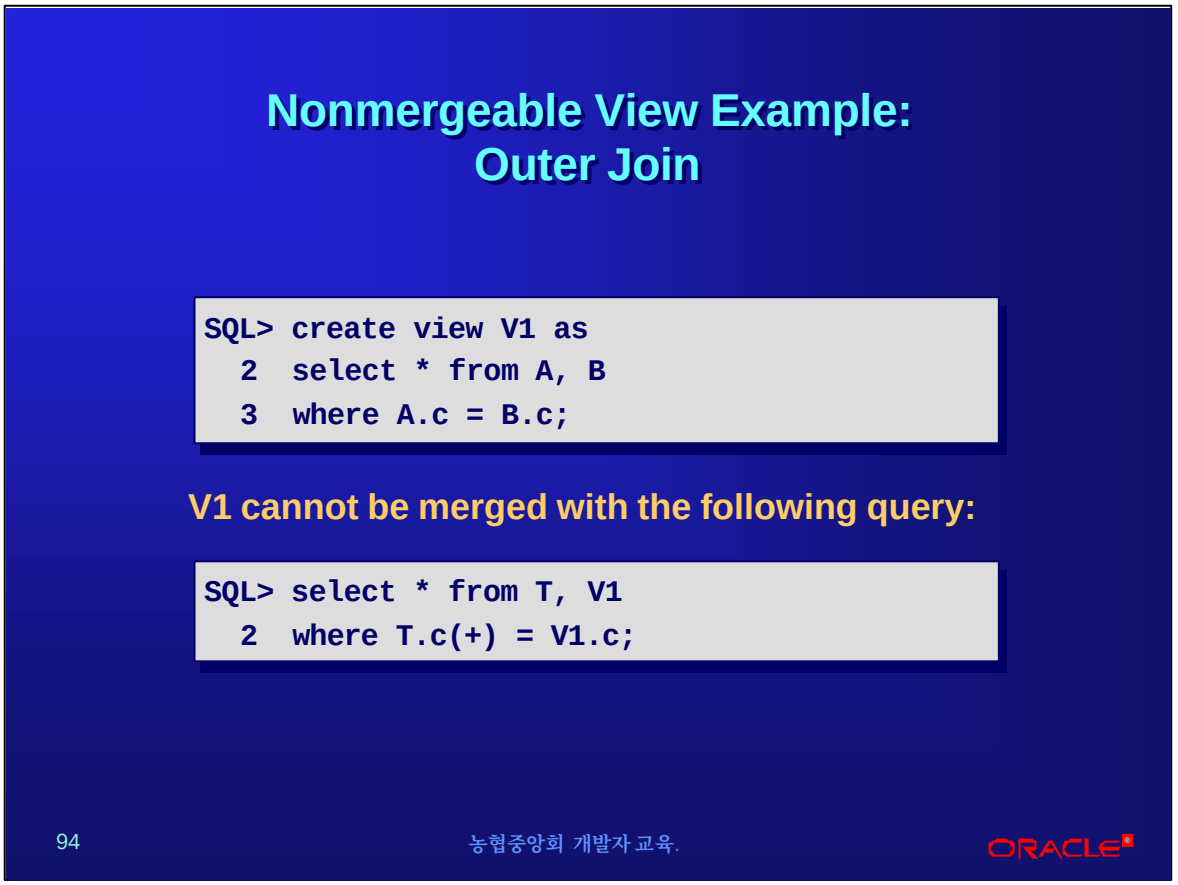
View Merging

View definitions with queries that contain the following are not merged:

- GROUP BY clause
- All AGGREGATE functions
- ROWNUM reference
- START WITH/CONNECT BY clause
- All set operations (UNION, MINUS, ...)
- DISTINCT (unless the inner query also contains a DISTINCT)
- The view is a join with another table in the original query

If the view is nonmergeable, the view definition as a whole is placed into the FROM clause as an in-line view and this is executed as a separate query block. This may result in certain indexes not being used or joins not being available.

A simple view is a view that contains only one table and whose WHERE clause does not contain predicates without references to columns (for example constant predicates, like 1 = 1), ROWNUM predicates, subqueries, or ORs.



Nonmergeable View Example

Restrictions for merging views that are being outer joined:

- The view must be a single table (that is, a simple view)
- The WHERE clause of the view must not contain predicates without references to columns (such as constant predicates like 1 = 1), ROWNUM predicates, subqueries, or ORs.

The merging essentially corresponds to adding an (+) operator to both the columns in the WHERE clause and the restrictions reflect cases where this would be impossible or illegal.

In the above case the execution plan would be something like:

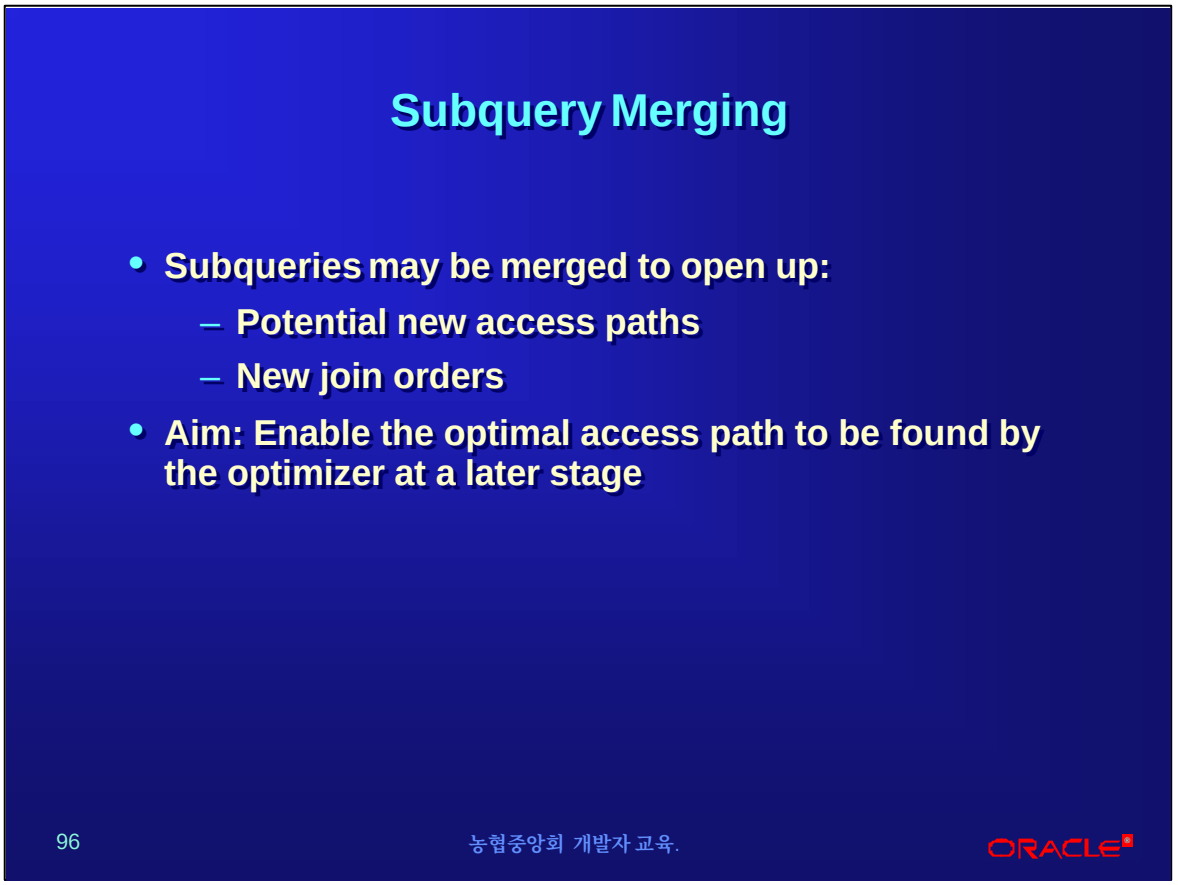
```
SELECT STATEMENT
  HASH JOIN (OUTER)
    VIEW OF V1
      NESTED LOOPS
        TABLE ACCESS (FULL) OF A
        TABLE ACCESS (BY INDEX ROWID) OF B
          INDEX (RANGE SCAN) OF B_C
    TABLE ACCESS (FULL) OF T
```

It is important to understand some of the reasons for nonmergeability of views. This will help explain the appearance of VIEW operations within explain plan output.

Nonmergeable View Example: ROWNUM Reference

```
SQL> create view v_courses as
  2  select * from courses
  3  where ROWNUM = 1;
```

```
SQL> select c.short_name, e.last_name
  2  from v_courses c, employees e
  3  where e.emp_id = c.dev_id;
```



Correlated Subqueries

The subquery depends on the outer query for its value. It is executed repeatedly, once for each row that is selected by the outer query.

```
SQL> select e.last_name
  2  from   employees e
  3  where  exists (select 'X'
  4                 from   registrations r
  5                 where  r.stud_id = e.emp_id
  6                 and    r.status = 'PEND');
```

The statement executes the subquery for every row retrieved from the employees table. A correlated subquery is a nested query that refers to a column from the outer query. In some cases they can perform poorly because the inner result set must be constructed for every single row that is a candidate for inclusion in the outer result set. This is especially true if both the inner and outer result sets are large.

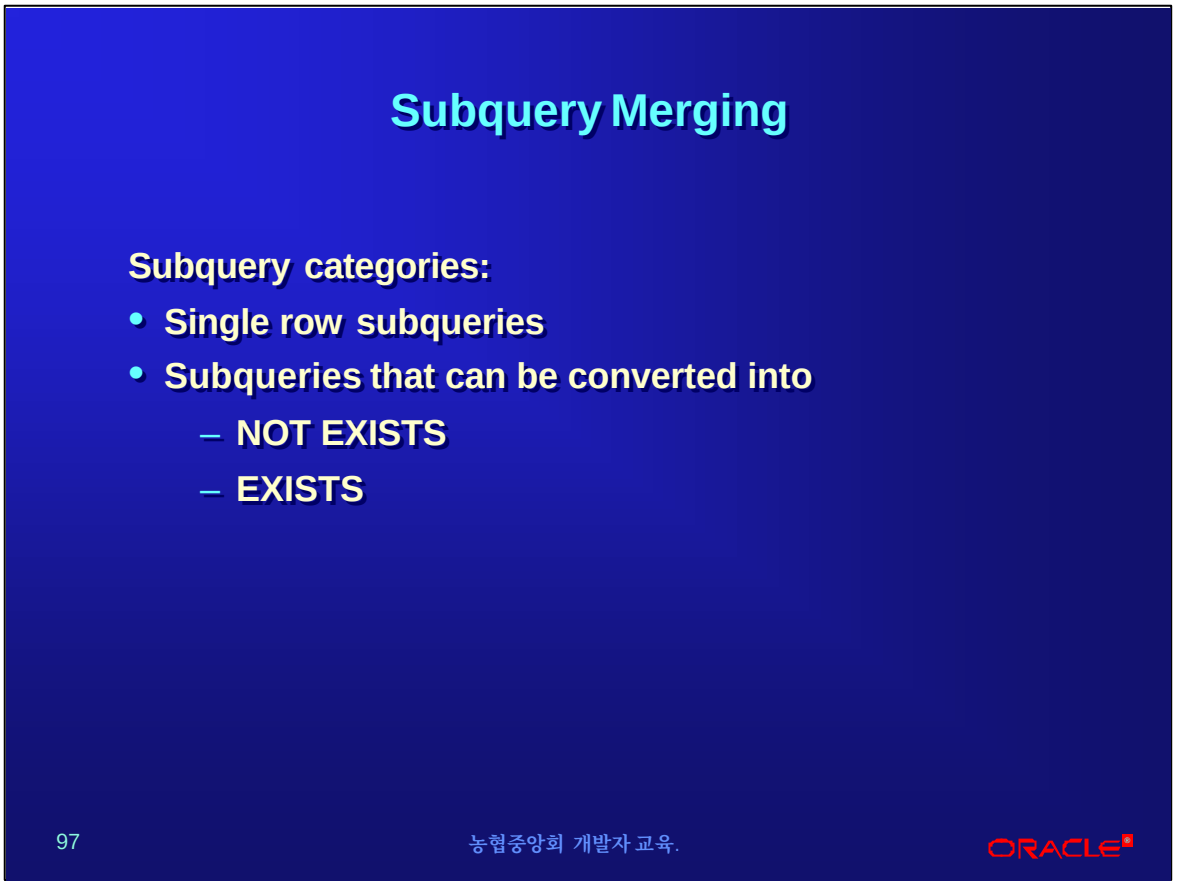
Noncorrelated Subqueries

A noncorrelated subquery does not depend on the outer query.

```
SQL> select e.*
  2  from   employees e
  3  where  e.emp_id in (select r.stud_id
  4                      from   registrations r
  5                      where  r.reg_date > sysdate);
```

Semijoins and Antijoins

A *semijoin* is a subquery that uses the EXISTS predicate, whereas a subquery with NOT IN that conforms to a set of special rules is called an *antijoin*.



Single Row Subqueries

A single row subquery is a subquery that evaluates to on one row:

```
SQL> select ...
  2  from   employees e
  3  where  e.emp_id = (select c.instr_id
  4                     from   classes c
  5                     where  c.class_id = 12501);
```

ISO/ANSI rules say that subqueries should be executed once for each row returned by the outer query. Oracle's approach is to evaluate the subquery and store the result. It uses the estimated cost of executing the subquery in join order calculations. This can be seen in 10053 output.

```
SQL> select ...
  2  from   employees e
  3  where  e.emp_id = <evaluated_value>;
```

Note: The query is not actually rewritten as shown above.

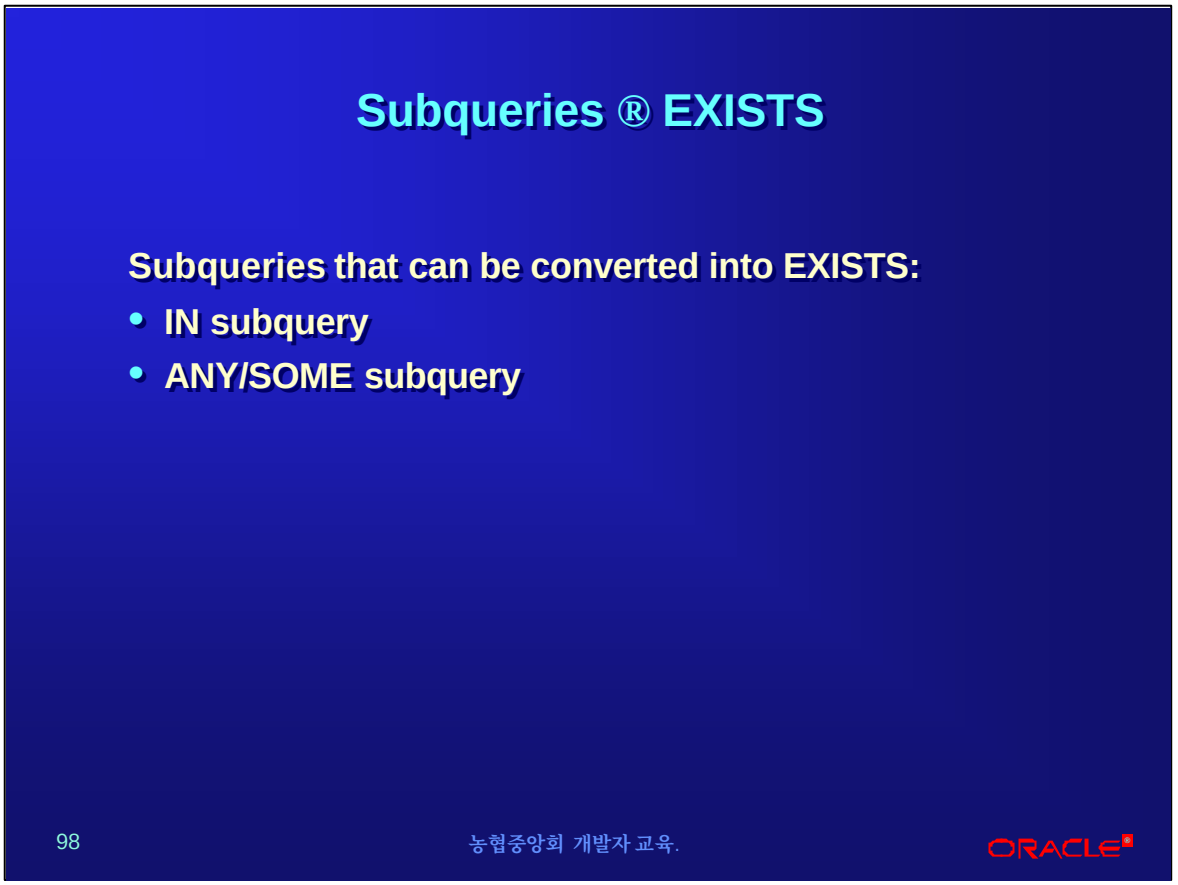
In this example the = is forcing a single row to be returned. If more than one row is desired then different constructs such as in needs to be used.

If the subquery returns more than one value then you receive an error such as:

ORA-1427: "single-row subquery returns more than one row"

Subquery Merging

The Oracle server can convert some subqueries to NOT EXISTS and EXISTS.



IN or = ANY Subquery

The Oracle server first tries to flatten (convert the subquery to a join) an IN into a join under the same rules that apply to view merging.

If any of these conditions are not met and the subquery is not correlated, then the Oracle server transforms it into an in-line view adding a distinct clause to the select list. Transforming into an in-line view causes the subquery to be optimized separately; this appears as a view or filter step in the explain plan output.

Otherwise, it will be transformed into an EXISTS

Example 1

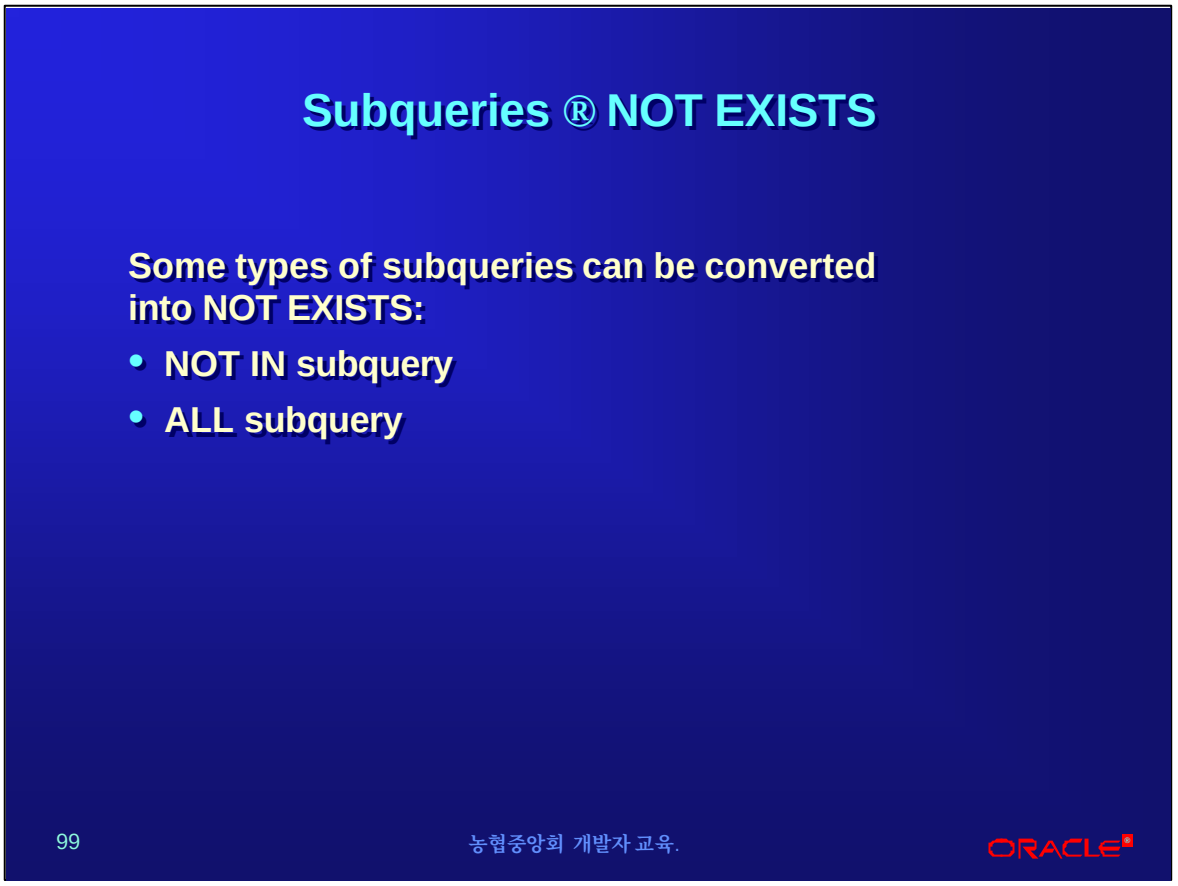
The following IN subquery is flattened (converted to a join):

```
SQL> select count(*)
  2  from   courses
  3  where  dev_id in (select emp_id from employees)

Execution Plan
----------------------------------------
0    SELECT STATEMENT
1  0  SORT (AGGREGATE)
2  1   NESTED LOOPS
3  2    TABLE ACCESS (FULL) OF 'COURSES'
4  2     INDEX (UNIQUE SCAN) OF 'EMP_PK'
```

Example 2

The following IN subquery is transformed into an in-line view:

```
SQL> select count(*)
  2  from   courses
  3  where  dev_id in (select instr_id from classes)

Execution Plan
----------------------------------------
0    SELECT STATEMENT
1  0  SORT (AGGREGATE)
2  1   HASH JOIN
3  2    VIEW OF (VW_NSO_1)
4  3     SORT (UNIQUE)
5  4      TABLE ACCESS (FULL) OF 'CLASSES'
4  2    TABLE ACCESS (FULL) OF
```



NOT IN Subquery

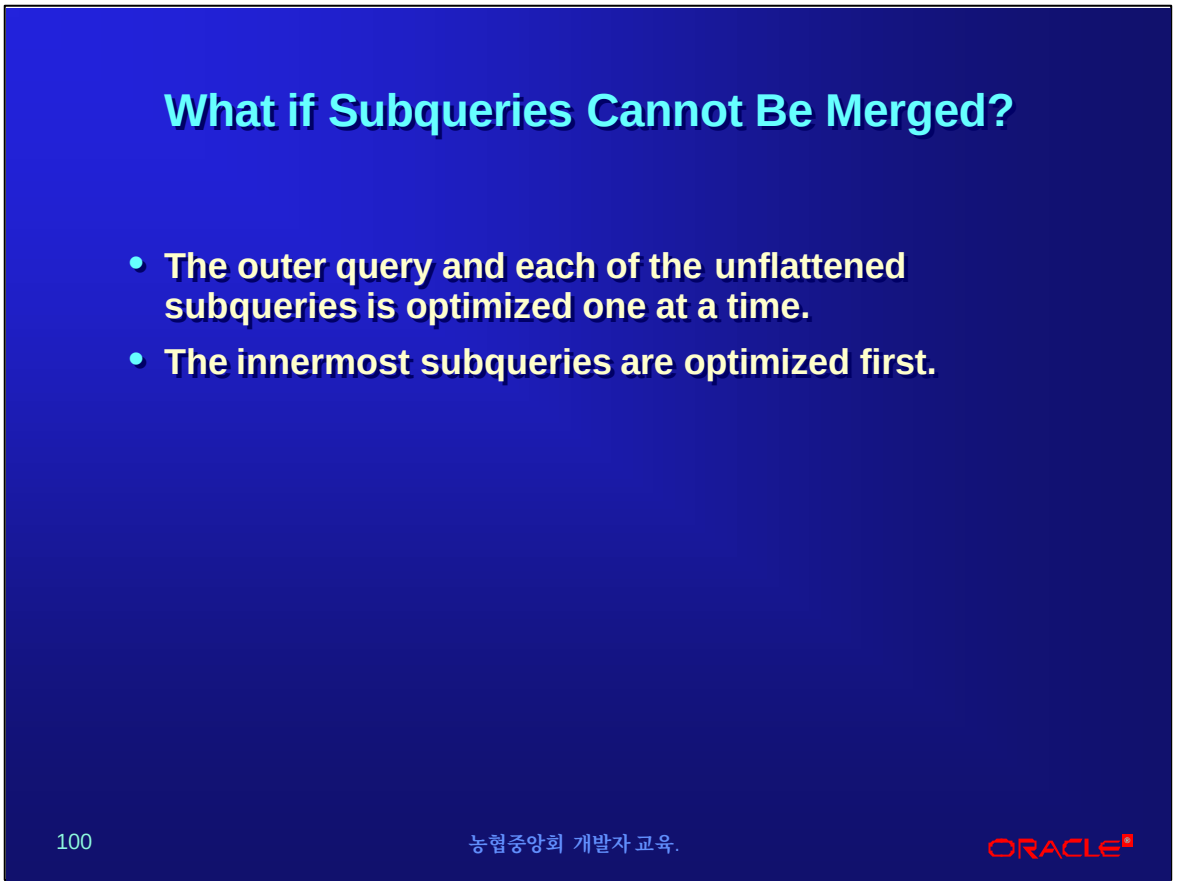
The Oracle server first tries to convert the NOT IN subquery into an antijoin if the following conditions are met:

1. All columns referenced in the subqueries are not null
2. Subqueries are not correlated
3. The WHERE clause of the outer query block does not have any ORs in it

If these conditions are not met, then the NOT IN subquery is transformed into a NOT EXISTS (with a FILTER operation).

ALL Subquery

An ALL subquery is always transformed into a NOT EXISTS.



What if Subqueries Cannot Be Merged?

If a subquery cannot be merged, the Oracle server evaluates the subquery separately and then attaches this to the outer query.

Noncorrelated subqueries are evaluated and the resultant value substituted into the WHERE clause.

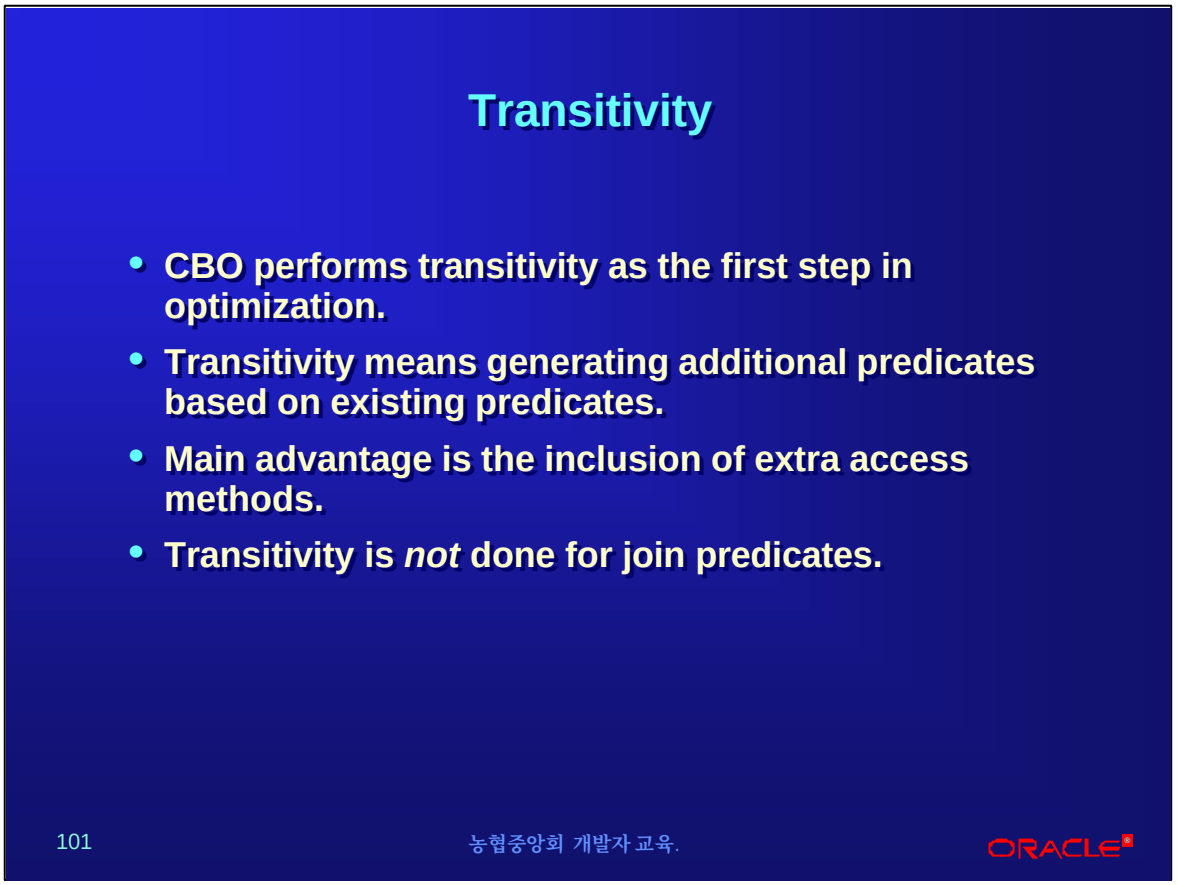
Innermost refers to the most deeply nested subqueries.

If the Oracle server is unable to merge a subquery it is likely that you will get worse performance.

The main problem with merging subqueries is the possibility of introducing duplicates. If the Oracle server is unable to guarantee that duplicates will not be returned then the subquery cannot be merged. For example:

```
IN (a,a,b,c)
```

IN only returns one row for a. If this is transformed into a join, then two rows are returned. In order to merge it must be ensured that only one row is output. Usually this is achieved by using distinct or a unique index.



Transitivity

Assume A, B, and C are columns. If:

```
A=5 and A=B
```

Then it can also be concluded that:

```
B=5
```

This can be applied to queries as shown in the following slide.

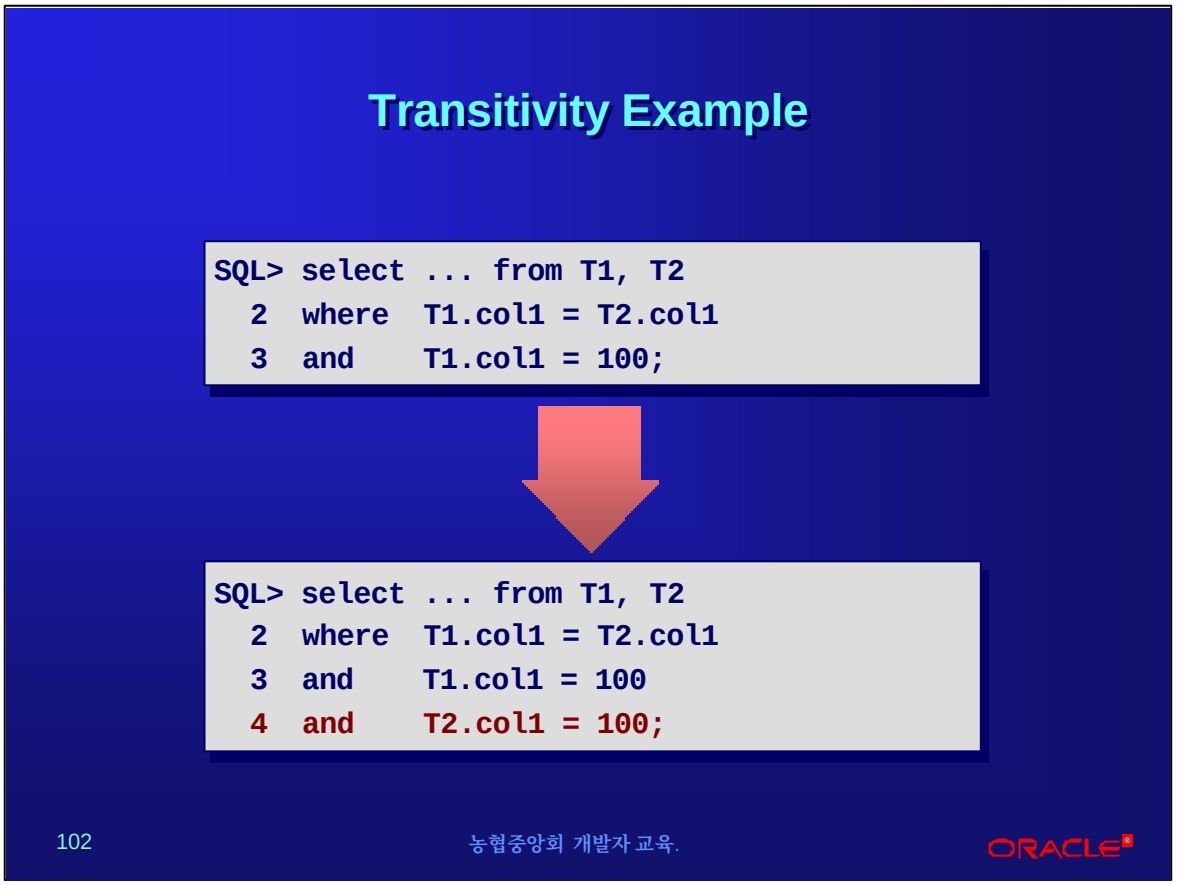
However, in terms of joins, if:

```
A=B and B=C
```

It is *not* concluded that:

```
A=C
```

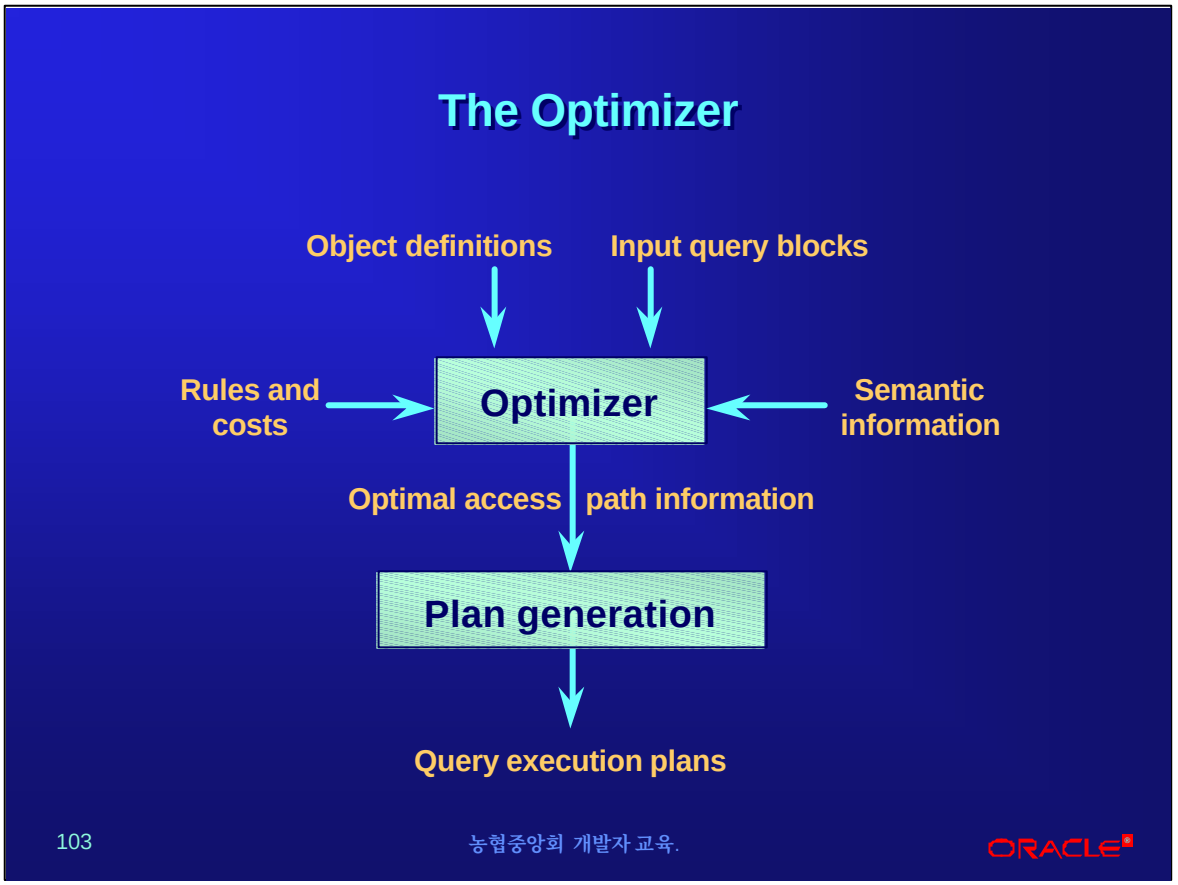
In other words, the Oracle server does not generate transitive closures for join predicates.



Transitivity Example

In the above query, assume there is an index on T2(Col1).

In the first SELECT statement the Oracle server is unable to drive off this index on T2 unless it performs a full index scan because it does not have a value for the leading column. Once the transitive predicate has been introduced it makes driving off T2 feasible because the leading column is now filled.

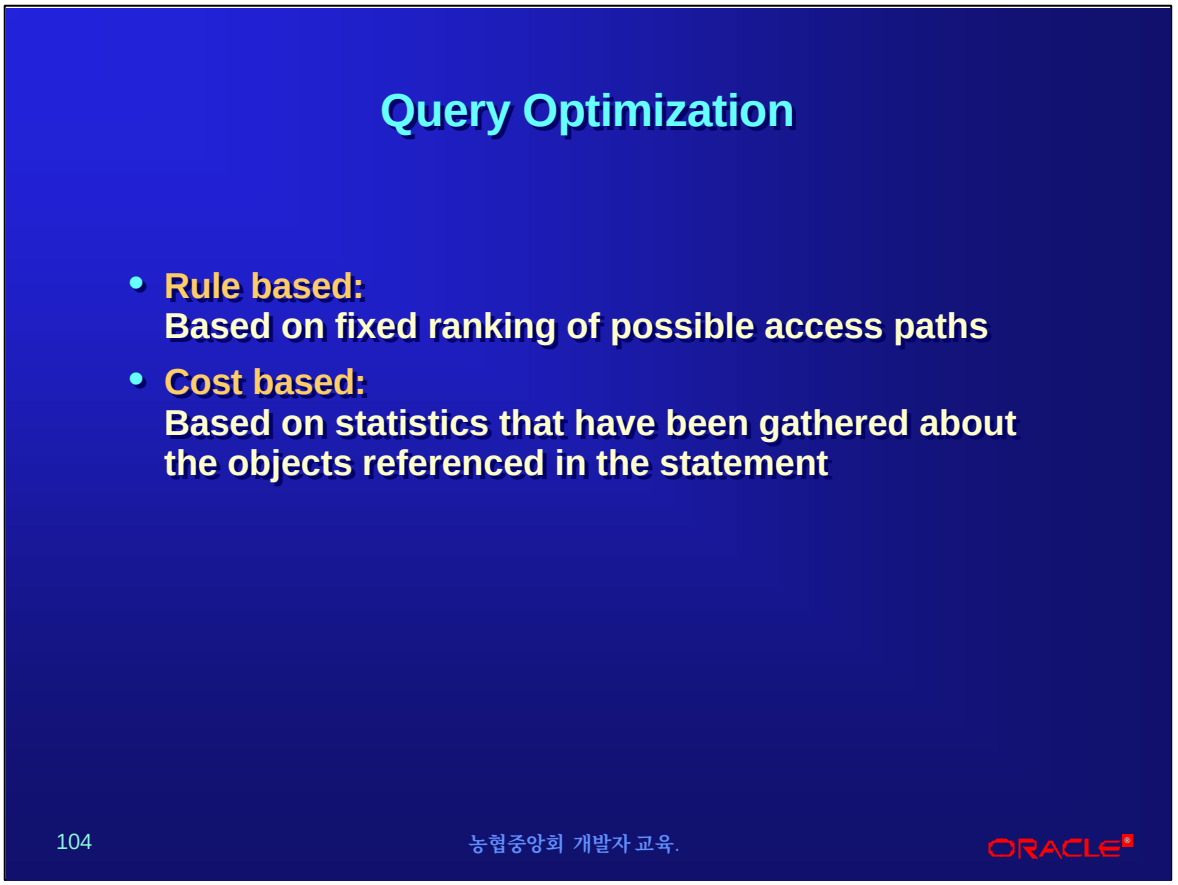


The Optimizer

At this stage the optimizer has a query block (or blocks) passed to it. It uses this along with other information to produce the optimal access path for the query.

When the query enters the optimizer it may contain multiple query blocks; for example, it may contain nonmergeable view query blocks after the view merging phase.

The following slides and the following lessons investigate the contents of the optimizer box.



Query Optimization

Oracle7, Oracle8, and Oracle8*i* have two possibilities for query optimization: RBO and CBO.

Rule-Based Approach (RBO)

- Uses a documented set of rules to determine access path for queries
- Rules are optimized for OLTP type queries
- Problems:
  - New features not available
  - Difficult to tune queries
  - Query structure may affect access path

105 농협중앙회 개발자 교육. ORACLE

Rule-Based Access Path Ranking

1. Single row by rowid
2. Single row by cluster join
3. Single row by hash cluster key with unique or primary key
4. Single row by unique or primary key
5. Cluster join
6. Hash cluster key
7. Indexed cluster key
8. Composite key (entire key)
9. Single-column indexes (single index or index merge)
10. Bounded range search on indexed columns (including key prefix)
11. Unbounded range search on indexed columns (including key prefix)
12. Sort-merge join
13. Maximum or minimum of an indexed column
14. Order by on indexed columns
15. Full-table scan

Several features cannot be used with the rule-based optimizer:

Parallelism, reverse index scans, partition view elimination, star joins, hash joins, fast full index scans, function indexes, materialized views, temporary tables, domains, and so on.

Setting the Optimizer Mode (9i 포함)

- 9i 에서 FIRST_ROWS_N Optimizer mode추가 되었음. (N: 1,10,100,1000)
- At the instance level:
  - `optimizer_mode = {Choose|Rule|First_rows|First_rows_n|All_rows}`
- At the session level: (instance level에 우선)
  - `ALTER SESSION SET optimizer_mode = {Choose|Rule|First_rows|First_rows_n|All_rows}`
- At the statement level: Using hints (Instance, Session level에 우선)
- OPTIMIZER_MODE=CHOOSE 일 경우
  - 통계정보가 없다면 기본적으로 RULE base(RBO)로 Plan이 결정
  - 'RULE,'DRIVING_SITE' Hint 이외의 Hint가 왔다면 CBO로 결정
  - Parallel Degree, Partition Table, SAMPLE절, ...등이 있으면 무조건 CBO
- OPTIMIZER_MODE=`First_rows|First_rows_n|All_rows` 일 경우
  - 통계정보가 없다면 Heuristics Value를 이용하여 CBO로 Plan이 결정, PLAN이 비효율적일 수 있음
- 통계정보가 있으나 Optimizer mode가 RULE 일 경우, 다른 hint가 오지 않은 경우와 Parallel Degree, Partition Table, SAMPLE절등이 나오지 않은 경우는 RBO로 처리
- RULE Hint와 다른 Hint가 오는 경우는 CBO로 처리.(Rule규칙이 위반되므로)

"기존에 7.x Version Rule Base Optimizer를 사용하고 있는데 CBO로 가야하는지 ?"

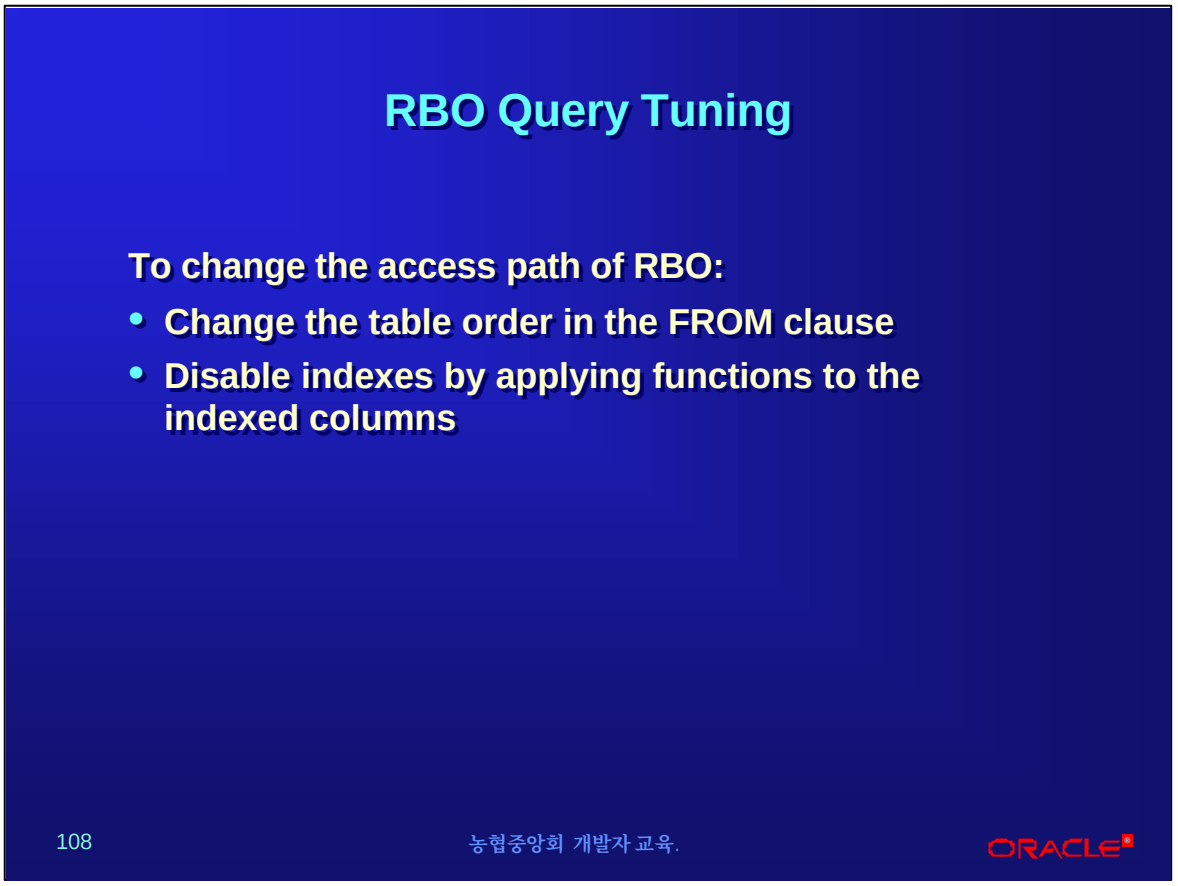
- CBO(CHOOSE, FIRST_ROWS, FIRST_ROWS_n, ALL_ROWS)로 통계정보(DBMS_STATS 또는 Analyze)를 운영할 것을 권장. 향후 10i 부터는 RBO는 not-support됨.

- RULE Base로 그대로 갈 경우 기존의 Plan과 달라질 수 있는 경우는
  . 기존의 보통 Table에서 partition Table로 전환되었거나
  . hint를 주었던 SQL문장들은 CBO로 동작하면서 느슨한 Hint(즉 모든 Access Path및 Join order, Join Method등을 완벽하게 주지 않은 Hint)에 의한 Plan이 바뀌게 되는 경우
  . Dictionary에 object의 순서가 변경되었거나 추가되었을 경우.
  (RBO는 같은 Ranking일 경우 나중에 만든 것이 우선순위가 있음)

- Oracle에서의 Rule Base Optimizer 기능은 앞으로도 추가적인 보완/발전은 없으며 10i 부터 not-support됨 (Note : 189702.1 참조). 그러므로 VLDB와 같은 Partition Table, Hash Join,... 같은 향상된 기능을 효과적으로 운영하기엔 Rule base는 적절치 않으며, Oracle의 향상된 Optimizer의 기능은 사용할 수 없기 때문. 또한 Rule Base Optimizer는 무조건 index만 있으면 전체 Table의 90% 이상을 Select하더라도 index를 타는 것과 같은 비 효율성이 있습니다. 그리고 SQL문장의 From절의 Table순서 및 Where절의 조건의 순서에 의해 Plan이 결정되므로 SQL Syntax에 상당히 신경을 써야 함.

- 물론 RULE Base Optimizer에서 Cost Base Optimizer로 전환된 Site의 공통된 문제점이 Index Scan이 Full Table Scan 형태의 Sort Merge Join 또는 Hash Join으로 많이 바뀌게 되는 경우가 많음 그러한 부분을 보완하기 위해서
  OPTIMIZER_INDEX_COST_ADJ=(default=100) : 100 보다 작은 수 (0에 가까울 수록 Index Scan, 10000에 가까울수록 Full Table Scan)
  OPTIMIZER_INDEX_CACHING=(default=0) : 100에 가까울 수록 Nested loop를 선호하는 Plan으로 동작

과 같은 Parameter로 CBO에서 Index의 Access부분에 점수를 더 주어 가능한 Index Scan으로 Plan이 결정되도록 조정 하거나 할 수가 있음.



RBO Query Tuning

When optimizing joins the RBO evaluates all possible join orders (not including Cartesian products) and searches the join order with the most single row predicates, fewest full table scans, and fewest sort-merge join operations.

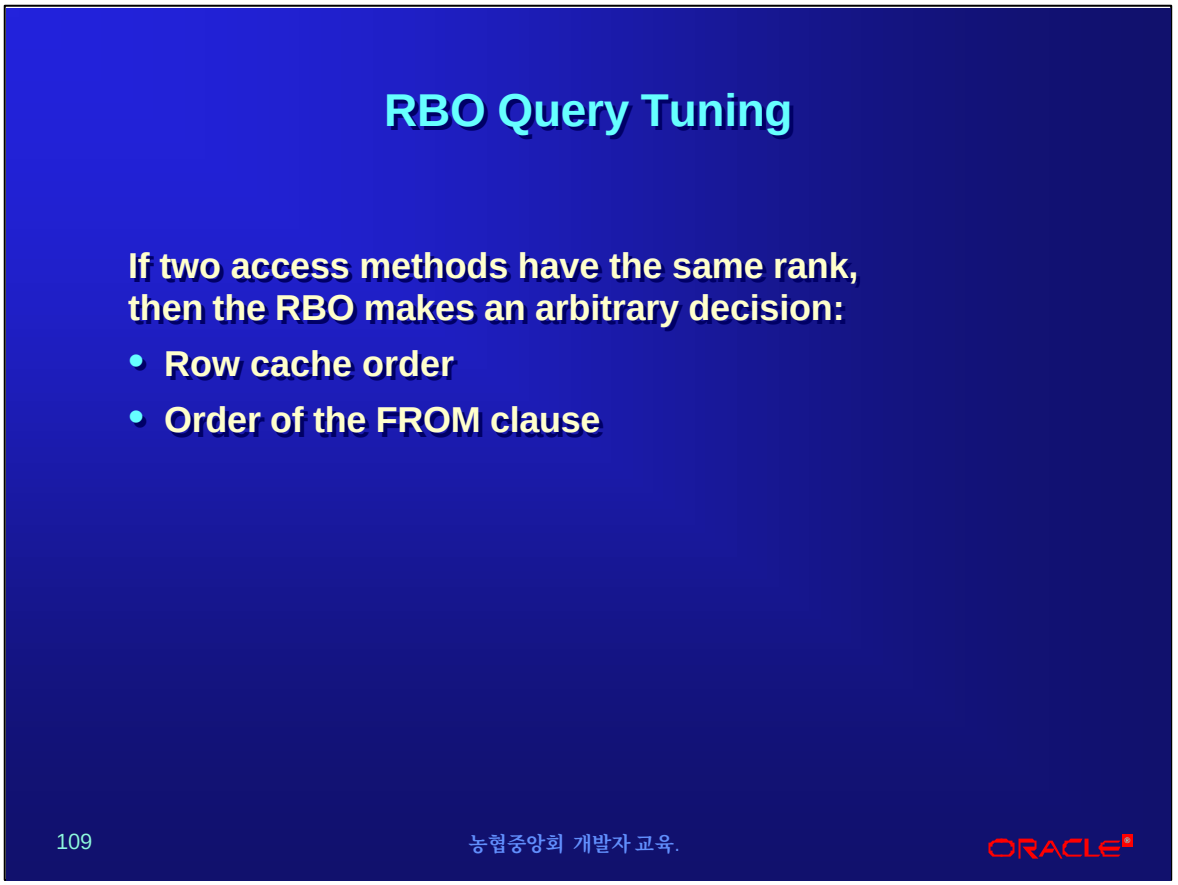
If after evaluating all possible join orders there is still a tie between two or more orders the RBO will select a driving table working from *right* to *left* in the FROM list.

The where predicates will be worked on from bottom to top, but this will only comes into effect if an AND-EQUAL operation can use more than five indexes. In this case the bottom five predicates are used for selecting the five indexes for the AND-EQUAL operation.

Functions applied to columns are applied to every row returned by the query.

This can have a major effect on performance and care should be taken to use lightweight functions (like ||'' for character columns and +0 for numeric and data columns).

By placing the function on the column you do not want to use an index on you can effectively influence the join order according to above comment on joins.



RBO Query Tuning (continued)

If there is a conflict between the ranks of two access paths, the RBO has no choice but to choose the first equally ranked access method in the row cache. This means that the access path of a query potentially can be affected by dropping and recreating indexes or other object reorganisations.

To reproduce with the RBO:

• Create two concatenated indexes; for example, on classes(type, status) and classes(status, type)

• Write a query that specifies all columns in both indexes in the WHERE clause

• Drop the indexes and recreate them in the reversed sequence

• Reexecute the query; the latest created index will be used

This can also happen in the CBO if identical costs are found. This however does not have such an impact as typically it is fairly rare for two objects to have identical cost and also if they do have the same cost then hopefully it costs the same to access them anyway.

RBO → CBO 전환 예 (7.3.4 → 9.0.x)

- Optimizer관련 Parameter 예
- Parameter설정 Focus :
  - OLTP는 OLTP의 특징인 Index위주의 Scan으로 처리
  - Batch job을 CBO의 향상된 기능인 Hash Join의 사용으로 성능의 개선으로 Best Throughput.

NAME	권장값	내용
db_file_multiblock_read_count	32	Full Table Scan시 Read Count
cursor_sharing	EXACT	임의의 상수에 대해서 강제적인 Bind변수로의 변경을 사용하지 않음
hash_area_size	10M	Hash Join발생시 적절한 Memory를 사용하기 위해. workarea_size_policy를 AUTO로 사용할 경우 참조만 될 것임
hash_join_enabled	TRUE	Hash Join Plan을 Optimizer가 고려한다
optimizer_index_caching	80	Index Scan위주로 Plan이 처리되도록 가중치 지정. Index를 이용한 Nested Loop를 선호한다.(0 to 100)
optimizer_index_cost_adj	20	Index Scan위주로 Plan이 처리되도록 가중치 지정 (1 to 10000)
optimizer_mode	CHOOSE	Optimizer Mode를 CHOOSE로
parallel_max_servers	6	최대 MAX parallel server 수. CPU * 2 * 2
sort_area_size	4M	Sort Area Size. Workarea_size_policy를 AUTO로 사용하면 적절히 설정하지만 실제 사용 안됨.
_sort_multiblock_read_count	32	2 -> 32. Temporary쪽에 I/O가 발생했을 때 I/O의 Read단위
pga_aggregate_target	500M	Instance내에서 PGA Target. SQL Work 영역으로 사용토록. (OS memory ? SGA) * 30%정도
workarea_size_policy	AUTO	SQL Work Area(SORT,HASH)를 자동설정 기능 사용.

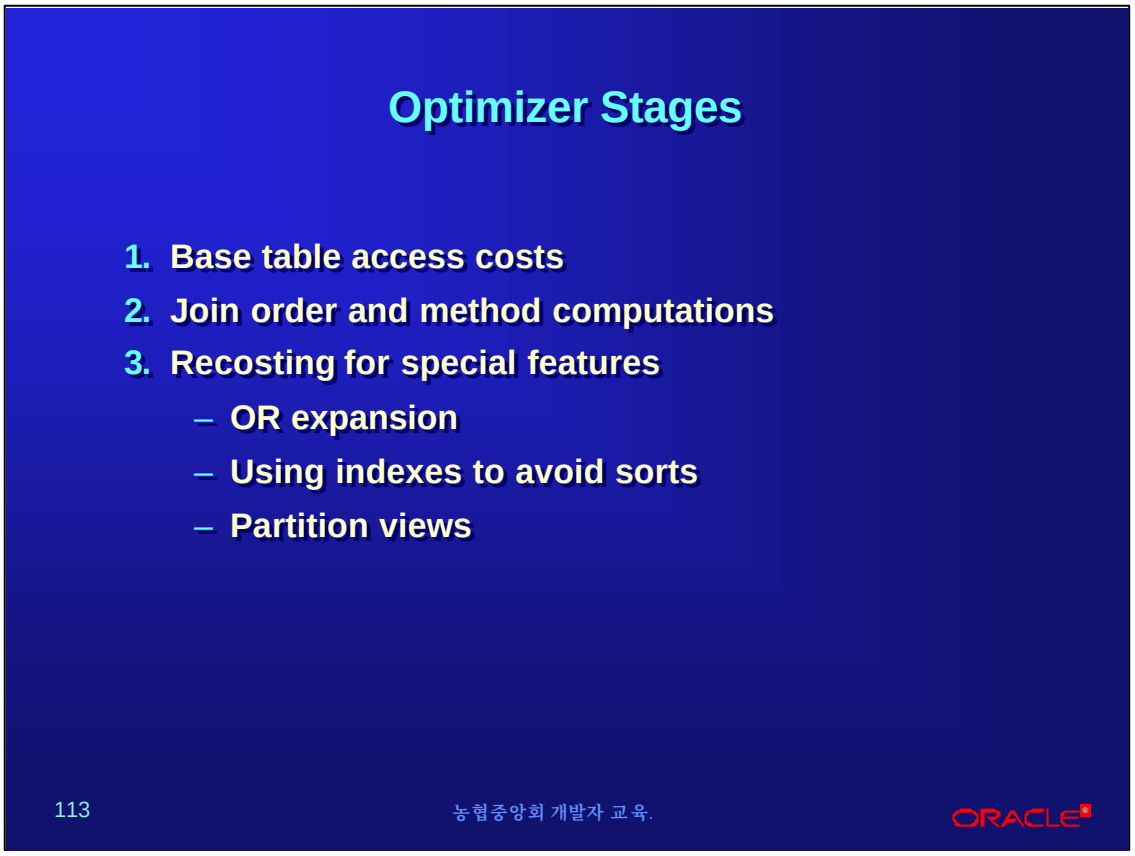
Features that Require the CBO)

- Features that Require the CBO

  Partitioned tables (*)
  Index-organized tables
  Reverse key indexes
  Function-based indexes
  SAMPLE clauses in a SELECT statement (*)
  Parallel execution and parallel DML
  Star transformations
  Star joins
  Extensible optimizer
  Query rewrite (materialized views)
  Progress meter
  Hash joins
  Bitmap indexes
  Partition views (release 7.3)
  Hint (*)
  Parallel DEGREE & INSTANCES – 'DEFAULT'도 해당(*)

Cost-Based Optimizer

- CBO makes optimization resource-driven instead of rule-driven.
- CBO meets the requirements of both DSS and OLTP systems.
- CBO optimization is based on:
  - Costs, based on gathered statistics
  - Hints
  - Parameters, to a lesser degree



Base Table Access Cost

This defines the best access method for each table involved in a query on a cost basis. Access cost and predicted (computed) cardinality are produced based on the supplied predicates.

Join Order and Method Computations

Each valid join order is computed and within this each valid join method is costed. The lowest cost method is chosen.

Recosting for Special Features

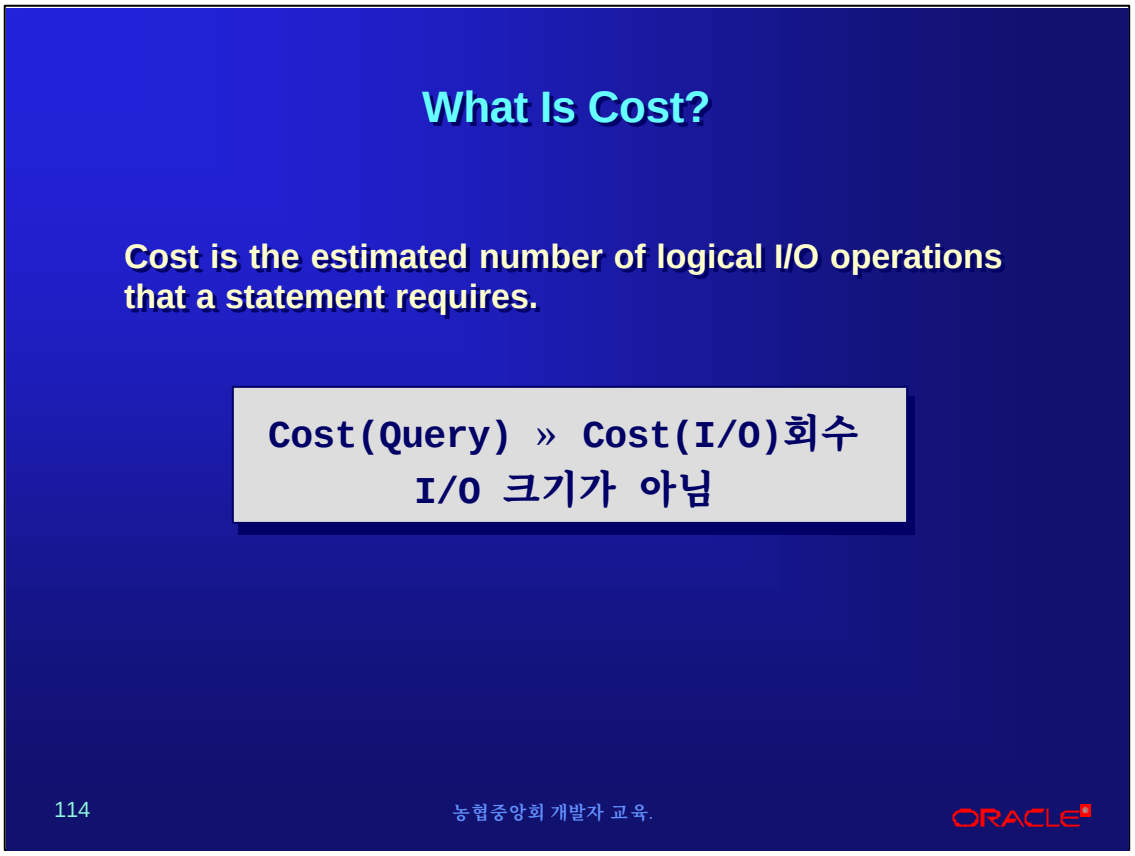
When certain features have been used in the query, the optimizer considers different costing possibilities.

OR expansion:
The CBO uses costs to decide if it should expand inlists and OR statements as a recosting phase after it has determined the base join costs for each join order. This is covered later.

ORDER BY using indexes to avoid sorts:
Recompute the costs, using indexes instead of sorting.

Partition views:
With Oracle7.3 partition views a base cost for the whole partition is computed as if it were a table; then each individual table is recosted to find the best access path for each table in the view.

Note: These three stages show up in event 10053 output, which is discussed in one of the following lessons.



What Is Cost?

The CBO cost model includes only the number of logical I/Os with minor adjustments to compensate for the lack of details regarding CPU and network costs.

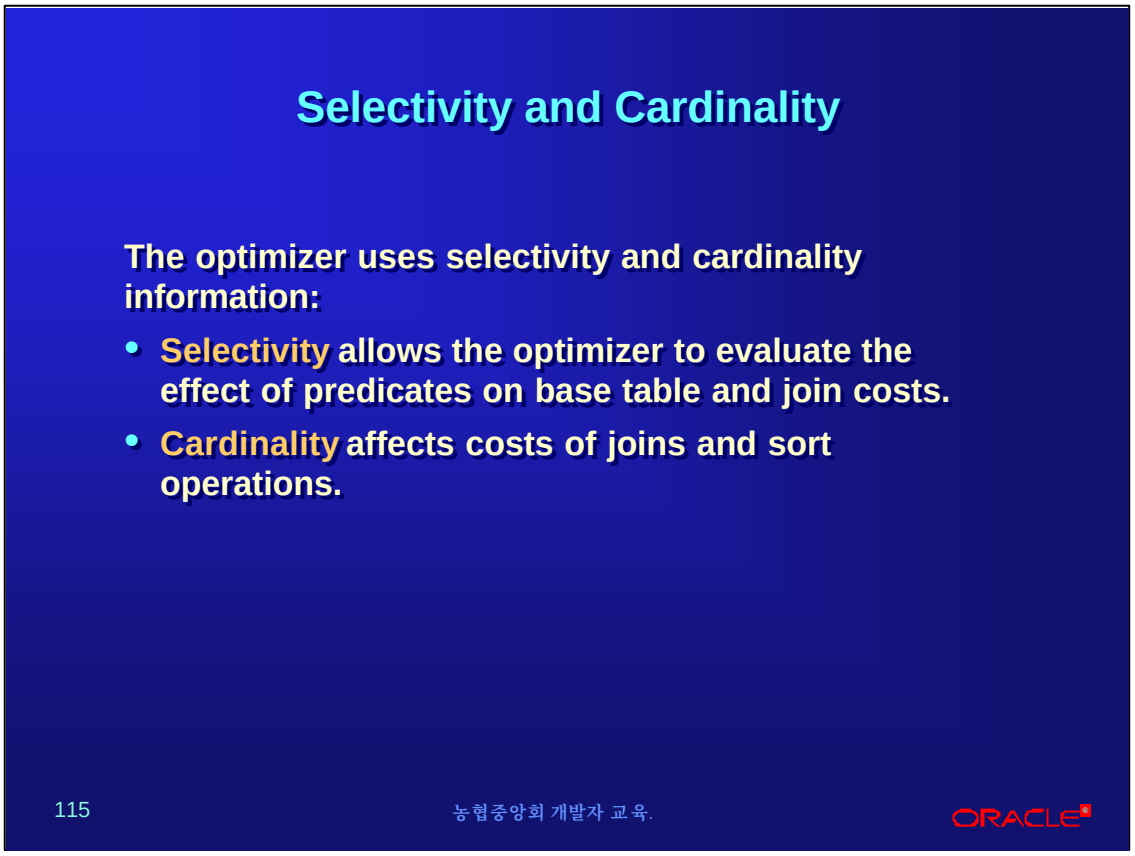
Cost is a comparative measure and as such little can be gained by attempting to relate it back to real I/O costs.

The use of object costs to determine optimal access paths gives more flexibility and more likelihood of finding the best plan.

The number of rows involved can help determine how many times objects will be joined.

The number of blocks provides the size of the object.

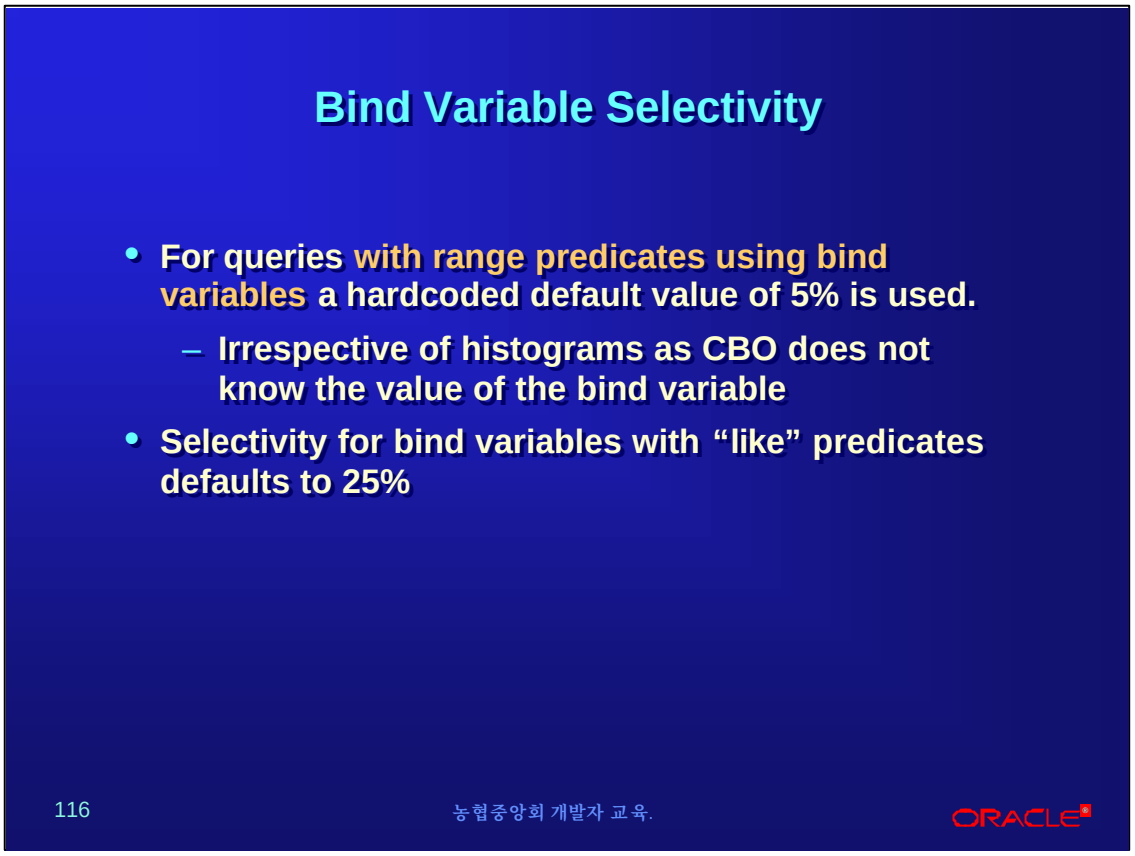
Note: Experience has shown that focusing on the computed cost is nearly useless for tuning purposes.



Selectivity and Cardinality

Selectivity is the means used by the optimizer to generate computed cardinality information from base cardinality.

Note: Selectivity is always a relative value (a percentage); cardinality is an absolute value.



Bind Variable Selectivity

Bind variables are a big headache for the CBO because it cannot determine accurate selectivities for range predicates containing bind variables.

Example:

```
SQL> select last_name from employees
  2  where  emp_id > 9999;
SQL> select last_name from employees
  2  where  emp_id > :b1;
```

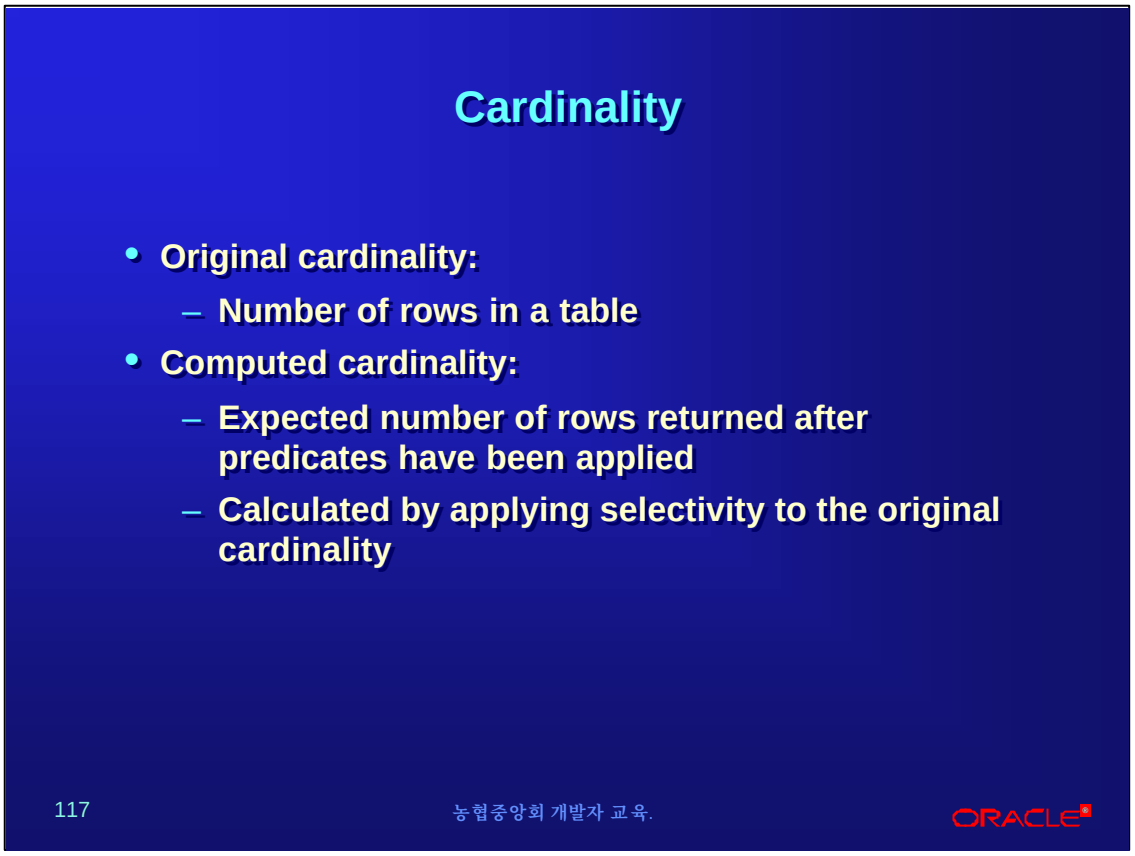
Assuming the table has been analyzed, CBO knows the HIGH and LOW values for emp_id and that the values are evenly distributed between these points.

For the first statement, CBO can determine the selectivity for the where clause; it uses the assumption that values are evenly distributed to enable it to estimate the number of values between the supplied value and the HIGH value.

For the second statement, CBO does not know what the value of :b1 will be, so it is unable to use the same assumption and uses the default selectivity (5%) instead.

Before Oracle7.3.2 the bind variable selectivity was hardcoded to 50% for bounded ranges and 25% for unbounded ranges.

Note: Oracle internal pseudocolumns (like SYSDATE) are treated like bind variables as a statement might be reexecuted a long time after parsing (and optimization).



Cardinality (continued)

Using event 10053 (described later in the course) the optimizer's calculations can be evaluated. Consider the following statement:

```
SQL> select days
  2  from   courses
  3  where  dev_id = 130000;
```

Suppose you have the following statistics:

- Number of distinct values in the DEV_ID column = 203
- Number of rows in table (original cardinality) = 1018

Then the selectivity can be estimated as:

```
Selectivity = 1/203 = 4.926*e-03
```

And the computed cardinality as:

```
Computed Cardinality = 4.926*e-03*1018 = 5.01
```

This is rounded up to the nearest integer = 6.

Join Selectivity and Cardinality

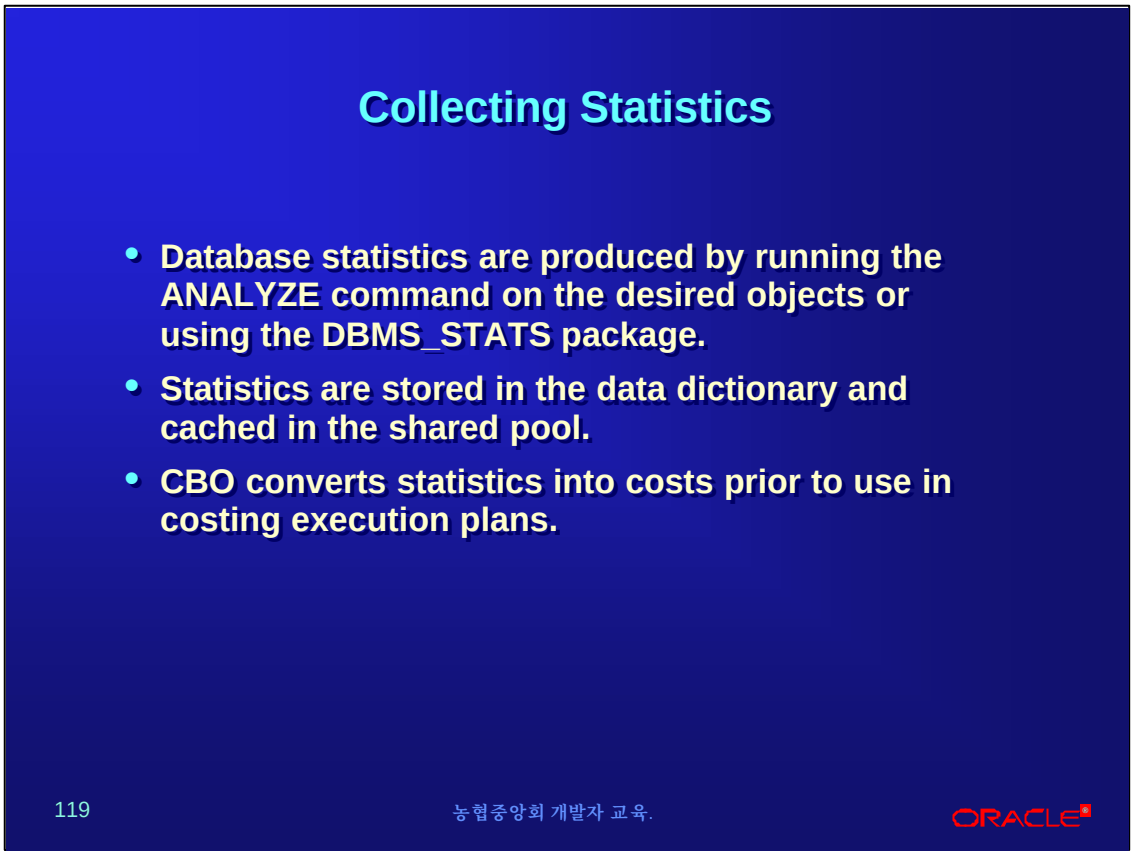
Let JP be a join predicate, defined as T1.c1 = T2.c2. Then the *join selectivity* is calculated as follows**:**

$$Sel(JP) = \frac{1}{max[NDV(T1.c1), NDV(T2.c2)]} * \frac{NN(T1.c1)}{Card(T1)} * \frac{NN(T2.c2)}{Card(T2)}$$

where NDV( ) returns the number of distinct values and NN( ) returns the number of not null values. The join cardinality is defined as follows:

```
Card(JP) = Card(T1) * Card(T2) * Sel(JP)
```

Like all other aspects of the optimizer, the formula for join cardinality assumes uniform column attribute distribution. An example calculation follows in a later lesson.



Collecting Statistics

Starting in Oracle7.2, an ESTIMATE sampling of 5% is usually as accurate as COMPUTE.

Starting in Oracle7.3, statistics for tables, indexes, and table columns can be gathered separately.

Note: The DBMS_STATS package is discussed later in this lesson.

Default Statistics

CBO may use default statistics when statistics are missing or bind variables are used

Selectivity	<7.3.2	>=7.3.2
equality predicate	.125	.01
inequality predicate	.75	.05
all other predicates	.25	.05
table row length	20	20
# of index leaf blocks	25	25
# of distinct values	100	100
table cardinality	100	100
remote table cardinality	2000	2000

ORACLE

Default Statistics

Default statistics can be important if the CBO is forced when insufficient statistics have been gathered or some objects are missing statistics.

With bind variables you often have no other alternative to the use of the defaults due to the lack of other information.

Because the statistics are hard-coded there is a high possibility that they are inaccurate. It is recommended that all objects are sufficiently analyzed prior to using the CBO.

Default Statistics (from kke.h) for Oracle7.3.3 and above:

Selectivity for relations on indexed columns	.009
Selectivity for = on indexed columns	.004
Multiblock read factor	8
Multiblock write factor	8
Remote table average row length	100
# of blocks	100
Scan cost	13
Index levels	1
number leaf blocks/key	1

Statistics (Analyze Command & DBMS_STATS #1)

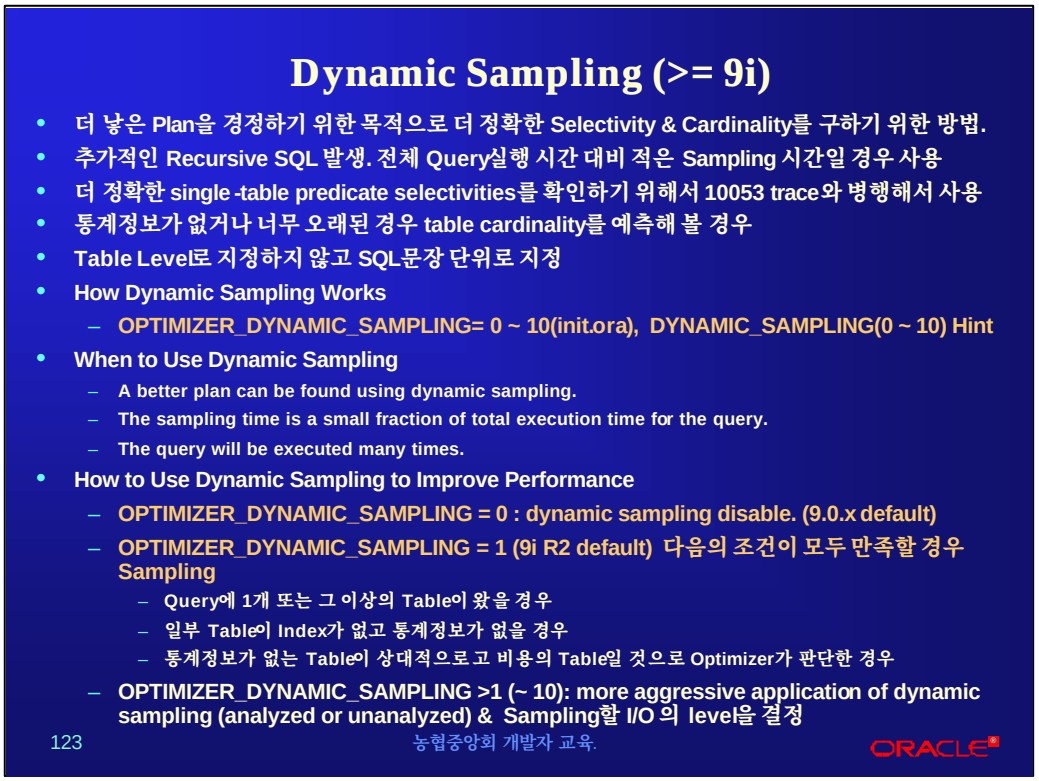
- DBA_TABLES
  NUM_ROWS,BLOCKS,EMPTY_BLOCKS,AVG_SPACE,CHAIN_CNT,
  AVG_ROW_LEN,AVG_SPACE_FREELIST_BLOCKS,NUM_FREELIST_BLOCKS,
  SAMPLE_SIZE,LAST_ANALYZED
- DBA_INDEXES
  BLEVEL,LEAF_BLOCKS,DISTINCT_KEYS,AVG_LEAF_BLOCKS_PER_KEY,
  AVG_DATA_BLOCKS_PER_KEY,CLUSTERING_FACTOR,NUM_ROWS,SAMPLE_SIZE,
  LAST_ANALYZED
- DBA_TAB_COLUMNS
  NUM_DISTINCT,LOW_VALUE,HIGH_VALUE,DENSITY,NUM_NULLS,NUM_BUCKETS,
  LAST_ANALYZED,SAMPLE_SIZE,AVG_COL_LEN
- DBA_TAB_HISTOGRAMS
  TABLE_NAME,COLUMN_NAME,ENDPOINT_NUMBER,ENDPOINT_VALUE,
  ENDPOINT_ACTUAL_VALUE

Statistics (Analyze Command & DBMS_STATS package차이점 #1)

- Analyze Command에만 있는 기능
  - Structural Integrity Check기능

    analyze { index/table/cluster } (schema.){ index/table/cluster } validate structure (cascade) (into schema.table);
  - Chained Rows 수집 기능

    ANALYZE TABLE order_hist LIST CHAINED ROWS INTO <user_tab>;
- Analyze Command & DBMS_STATS의 차이점
  - Analyze는 Serial Statistics Gathering기능만 있다.
  - DBMS_STATS은 parallel Gathering기능이 있다.(Index는 parallel불가)
  - Analyze는 Partition의 Statistics를 각 Partition table과 Index에 대해서 수집하고, Global Statistics는 Partition정보를 가지고 계산하므로, 비정확 할 수 있다. 그러므로 DBMS_STATS사용 권장.
  - DBMS_STATS은 전체 Cluster에 대해서는 Statistics를 수집하지 않는다.
  - DBMS_STATS은 CBO와 관련된 Statistics정보만을 수집한다. 즉 EMPTY_BLOCKS,AVG_SPACE,CHAIN_CNT,...등은 수집되지 않는다.
  - DBMS_STATS은 user의 Statistics table에 수집된 Statistics를 저장할 수 있고, Dictionary로 각 Column,Table,Index,Schema등을 반영할 수 있다.
  - DBMS_STATS은 IMPORT/EXPORT기능 및 추가적인 기능이 많다.(manual참조)



>>> DYNAMIC_SAMPLING Hint의 10053 TRACE

QUERY

select /*+ dynamic_sampling(7) */ deptno from emp where sal *5/8>300

.....

.....

***** 2003-05-28 18:06:58.000**

** Performing dynamic sampling initial checks. **

** Dynamic sampling initial checks returning TRUE (level = 7).

*** 2003-05-28 18:06:58.000

** Generated dynamic sampling query:

query text :

SELECT /*+ ALL_ROWS IGNORE_WHERE_CLAUSE */ NVL(SUM(C1),0), NVL(SUM(C2),0)

FROM (SELECT /*+ IGNORE_WHERE_CLAUSE NOPARALLEL("EMP") */ 1 AS C1,

CASE WHEN "EMP"."SAL"*5/8>300 THEN 1 ELSE 0 END AS C2

FROM "EMP" "EMP") SAMPLESUB

*** 2003-05-28 18:06:58.000

** Executed dynamic sampling query:

level : 7

sample pct. : 100.000000

actual sample size : 14

filtered sample card. : 14

orig. card. : 14

block cnt. : 1

max. sample block cnt. : 256

sample block cnt. : 1 <<<<<<<<< _OPTIMIZER_DYN_SMP_BLKS 와

OPTIMIZER_DYNAMIC_SAMPLING의 level에 의해 Sampling Block수가 결정됨

min. sel. est. : 0.0500

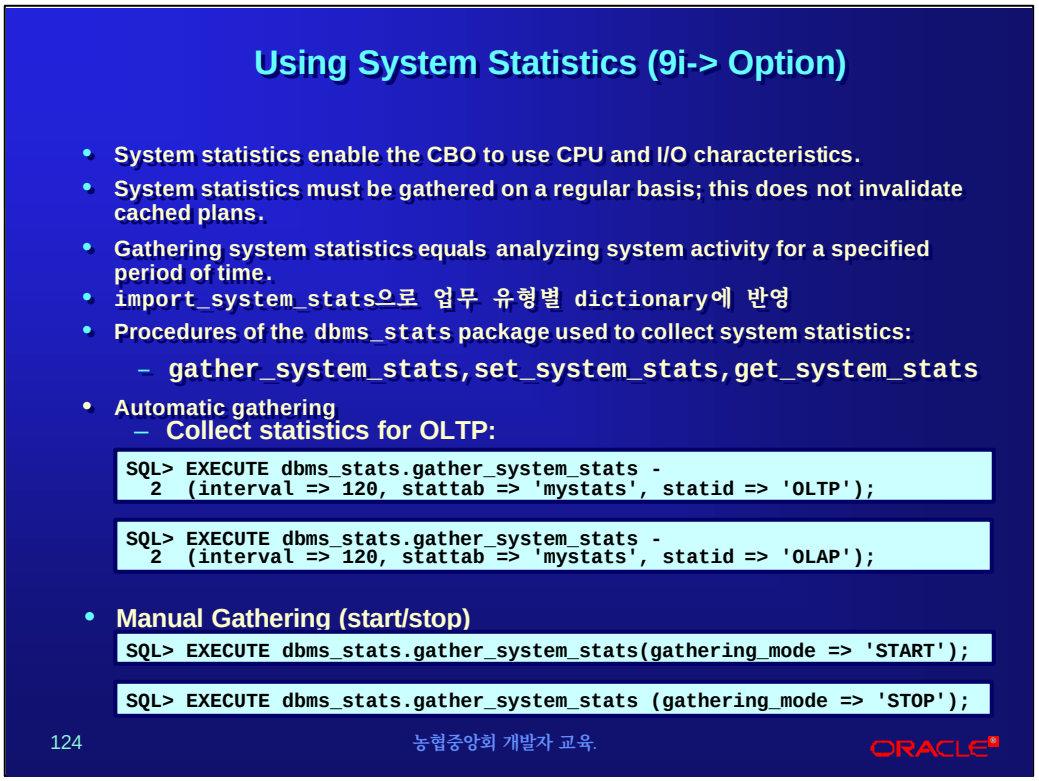
**** Using dynamic sel. est. : 1.00000000**

TABLE: EMP ORIG CDN: 14 ROUNDED CDN: 14 CMPTD CDN: 14

Access path: tsc Resc: 2 Resp: 2

BEST_CST: 2.00 PATH: 2 Degree: 1

.




```
SQL> EXECUTE dbms_stats.gather_system_stats -
>  (gathering_mode => 'START');

PL/SQL procedure successfully completed.

SQL>
SQL> select * from aux_stats$ ;

SNAME                 PNAME                     PVAL1 PVAL2
--------------------- ------------------------- --------- ------------------
SYSSTATS_INFO         STATUS
MANUALGATHERING
SYSSTATS_INFO         DSTART                     05-29-2003 17:08
SYSSTATS_INFO         DSTOP                      05-29-2003 17:08
SYSSTATS_INFO         FLAGS                    1
SYSSTATS_TEMP          SBLKRDS                  1044
SYSSTATS_TEMP          SBLKRDTIM                9000
SYSSTATS_TEMP          MBLKRDS                   205
SYSSTATS_TEMP          MBLKRDTIM                2740
SYSSTATS_TEMP          CPUCYCLES              285852
SYSSTATS_TEMP          CPUTIM                2095618
SYSSTATS_TEMP          JOB                      0
SYSSTATS_TEMP          MBRTOTAL                  3067

12 rows selected.


SQL> select * from table(dbms_xplan.display);

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

--------------------------------------------------------------------------
| Id  | Operation          |  Name       | Rows  | Bytes | Cost  |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             | 24591 |   768K|    43 |
|* 1 |  TABLE ACCESS FULL  | TESTEMP10   | 24591 |   768K|    43 |
--------------------------------------------------------------------------

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

   1 - filter("TESTEMP10"."DEPTNO"=10)

Note: cpu costing is off        <<<<<<<<<< System Stat을 STOP하기 전까지는
                                   사용안함

14 rows selected.


SQL> EXECUTE dbms_stats.gather_system_stats -
>   (gathering_mode => 'STOP');

PL/SQL procedure successfully completed.


SQL> explain plan for
  2  select * from testemp10 where deptno = 10;

Explained.


SQL>  select * from table(dbms_xplan.display);

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

--------------------------------------------------------------------------
| Id  | Operation          |  Name       | Rows  | Bytes | Cost (%CPU)|
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             | 24591 |   768K|    52  (18)|
|* 1 |  TABLE ACCESS FULL  | TESTEMP10   | 24591 |   768K|    52
(18)|
--------------------------------------------------------------------------

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

   1 - filter("TESTEMP10"."DEPTNO"=10)

12 rows selected.
```

Statistics Collection Level (>=9i)

- STATISTICS_LEVEL = {ALL | TYPICAL | BASIC} (Default : TYPICAL)
  - BASIC: No advisories or statistics are collected.
  - TYPICAL: The following advisories or statistics are collected:
    - Buffer cache advisory (V$DB_CACHE_ADVICE)
    - MTTR advisory (V$MTTR_TARGET_ADVICE)
    - Shared Pool sizing advisory (V$SHARED_POOL_ADVICE)
    - Segment level statistics (V$SEGSTAT)
    - PGA target advisory (V$PGA_TARGET_ADVICE)
    - Timed statistics
  - ALL: All of the preceding advisories or statistics are collected, plus the following:
    - Timed operating system statistics
    - Runtime Row Source Statistics - Row source execution statistics (V$SQL_PLAN_STATISTICS)
- V$STATISTICS_LEVEL View
- DB_CACHE_ADVICE, TIMED_STATISTICS, or TIMED_OS_STATISTICS 는 STATISTICS_LEVEL의 값을 override 한다


```
SQL> show parameter statistics_level
NAME_COL_PLUS_SHOW_PARAM                             TYPE    VALUE_COL_PLUS_SHOW_PARAM
---------------------------------------------------------------------- ----------- ------------------------------------------------------------------------------
statistics_level                                     string                  TYPICAL

SQL>  select STATISTICS_NAME ,SESSION_STATUS,SYSTEM_STATUS ,
          ACTIVATION_LEVEL,STATISTICS_VIEW_NAME
           from V$STATISTICS_LEVEL;
STATISTICS_NAME          SESSION_ SYSTEM_S ACTIVAT STATISTICS_VIEW_NAME
------------------------ -------- -------- ------- ------------------------
Buffer Cache Advice      ENABLED  ENABLED  TYPICAL V$DB_CACHE_ADVICE
MTTR Advice              ENABLED  ENABLED  TYPICAL V$MTTR_TARGET_ADVICE
Timed Statistics         ENABLED  ENABLED  TYPICAL
Timed OS Statistics      DISABLED DISABLED ALL
Segment Level Statistics ENABLED  ENABLED  TYPICAL V$SEGSTAT
PGA Advice               ENABLED  ENABLED  TYPICAL V$PGA_TARGET_ADVICE
Plan Execution Statistics DISABLED DISABLED ALL     V$SQL_PLAN_STATISTICS
Shared Pool Advice       ENABLED  ENABLED  TYPICAL V$SHARED_POOL_ADVICE
```

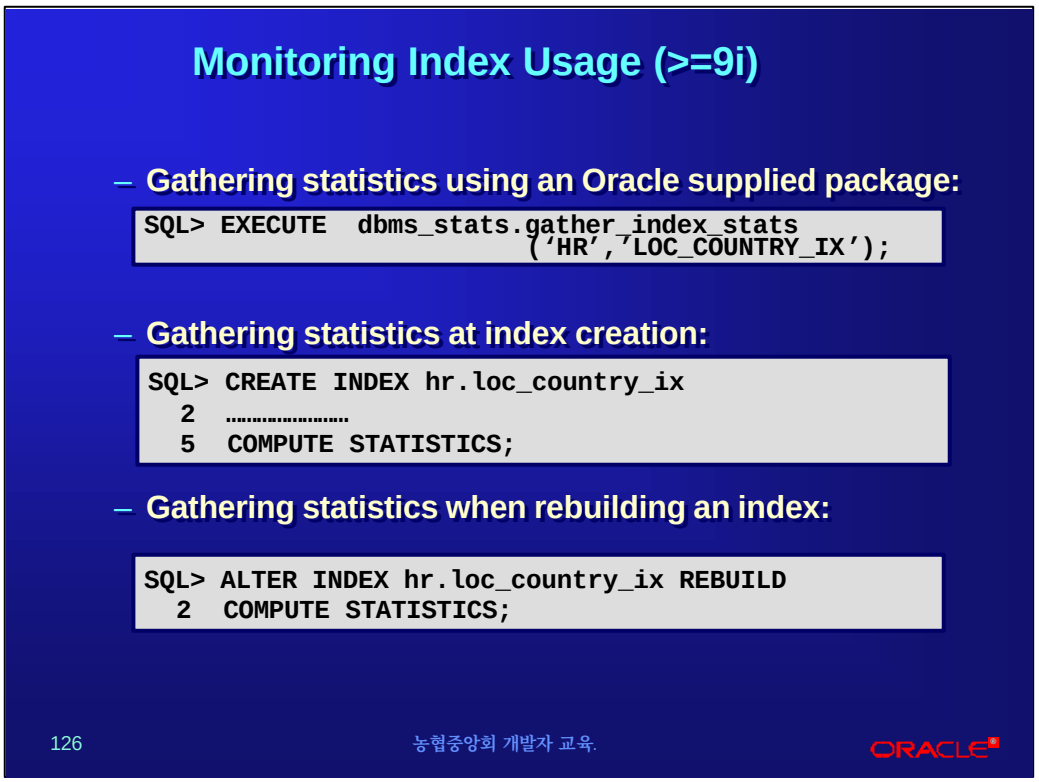
•v$segstat_name: Lists the segment statistics being collected

•v$segstat: Displays the statistic value, statistic name, and other basic information

•v$segment_statistics: Displays the segment owner and tablespace name in addition to all the rows contained in v$segstat

•V$SQL_PLAN_STATISTICS (9iR2) :

•V$SQL_PLAN_STATISTICS_ALL(9iR2) : SQL의 memory usage statistics정보도 포함 (sort or hash-join). This view concatenates information in V$SQL_PLAN with execution statistics from V$SQL_PLAN_STATISTICS and V$SQL_WORKAREA



-- Index에서 Segment Level의 Statistics걸기 (Index,Table다 잡힘) 또한 (V$OBJECT_USAGE에 Index사용 여부 기록)
alter index <index-Name> monitoring usage;
alter index <index-Name> nomonitoring usage;

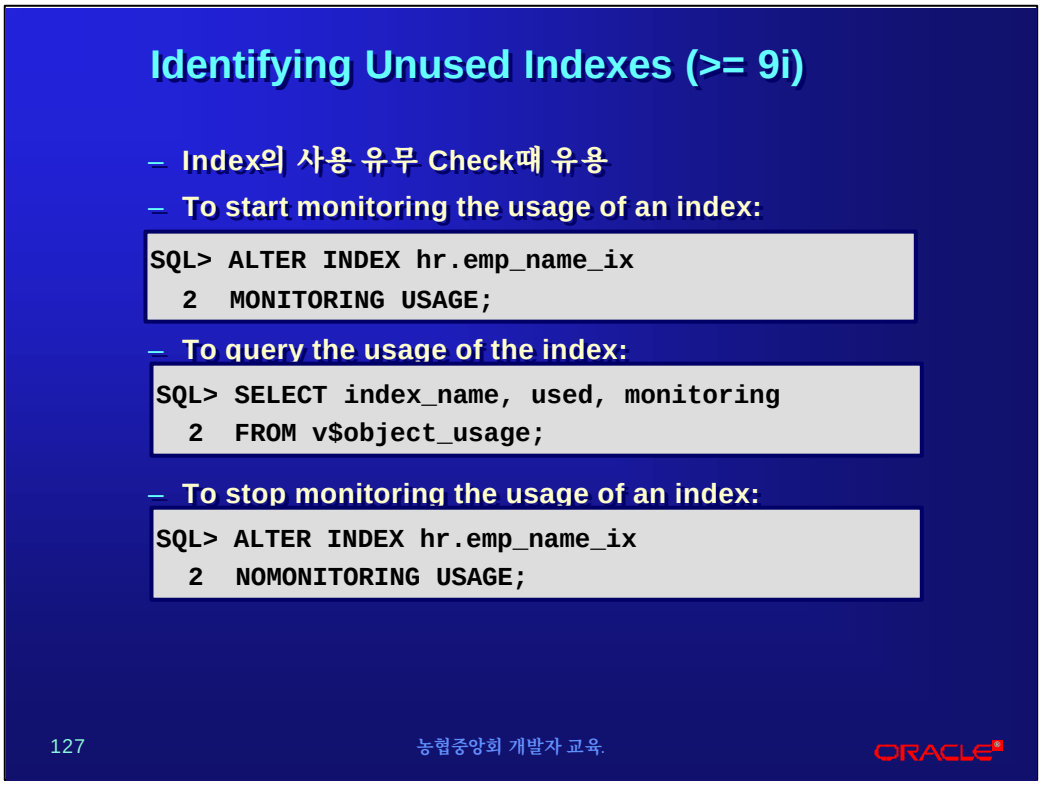
-- Table에서 Segment Level의 Statistics걸기
alter table <table-Name> monitoring;
alter table <table-Name> nomonitoring;

-- 자신의 Schema의 Index사용 여부만 보임(현재 Connect 된 Schema)
select * from v$object_usage;

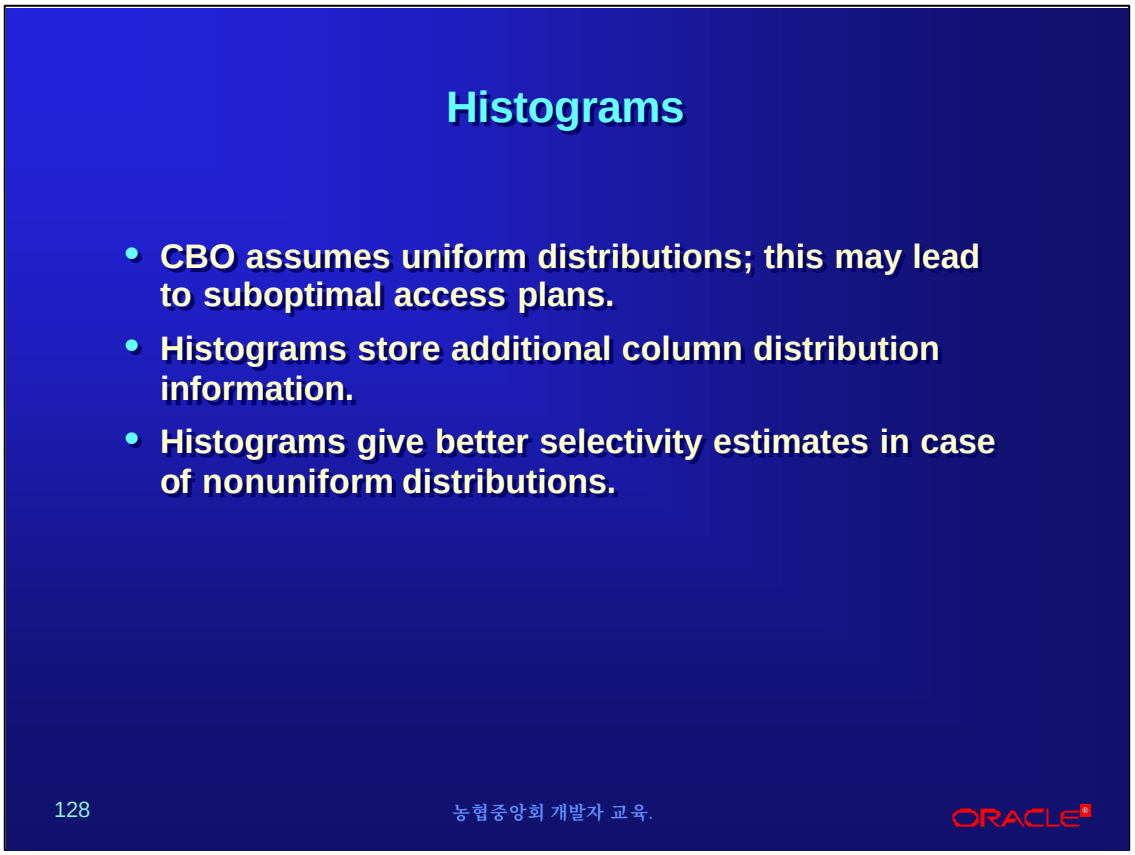
-- segment statistics이 수집되고 있는 Object List보기
select * from v$segment_statistics;

-- segment statistics이 수집되고 있는 Object List보기
select * from v$segstat_name;

-- segment statistics이 수집된 Data
select * from v$segstat;

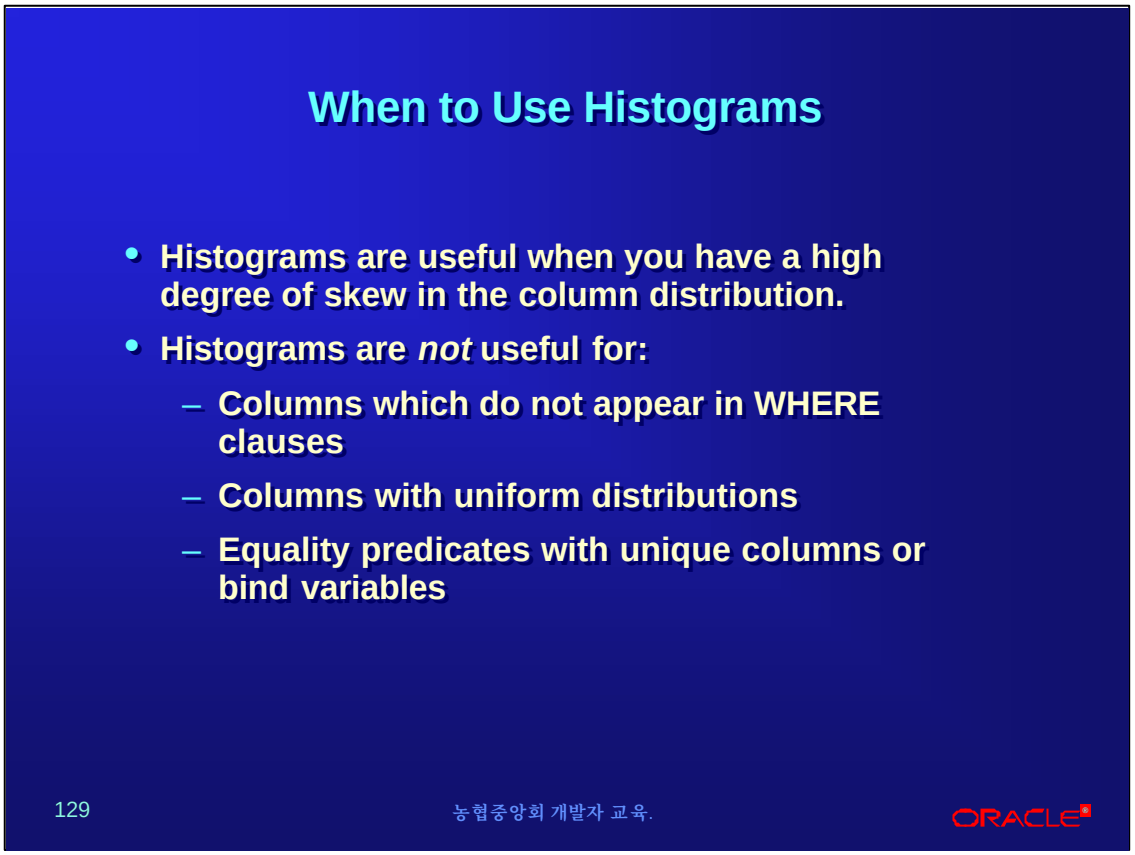



```
SQL> select * from v$object_usage;

no rows selected

SQL> alter index TESTEMP10_DEPTNO    monitoring usage;

Index altered.

SQL> select * from testemp10 where deptno = 20;

    EMPNO ENAME      JOB           MGR HIREDATE       SAL      COMM    DEPTNO
---------- ---------- --------- ---------- --------- ---------- ---------- ----------
     7369 SMITH      CLERK          7902 17-DEC-80        800                    20
....
....
....

20 rows selected.

SQL> select * from v$object_usage;

INDEX_NAME                     TABLE_NAME                     MON USE START_MONITORING    END_MONITORING
------------------------------ ------------------------------ --- --- ------------------- -------------------
TESTEMP10_DEPTNO               TESTEMP10                      YES YES 05/29/2003 17:42:44

SQL> alter index TESTEMP10_DEPTNO    nomonitoring usage;

Index altered.

SQL> select * from v$object_usage;

INDEX_NAME                     TABLE_NAME                     MON USE START_MONITORING    END_MONITORING
------------------------------ ------------------------------ --- --- ------------------- -------------------
TESTEMP10_DEPTNO               TESTEMP10                      NO  YES 05/29/2003 17:42:44 05/29/2003 17:43:38

SQL>
```



Histograms

The histogram approach provides an efficient and compact way to represent the data distributions. CBO will store information about the data distribution of each attribute in the data dictionary.

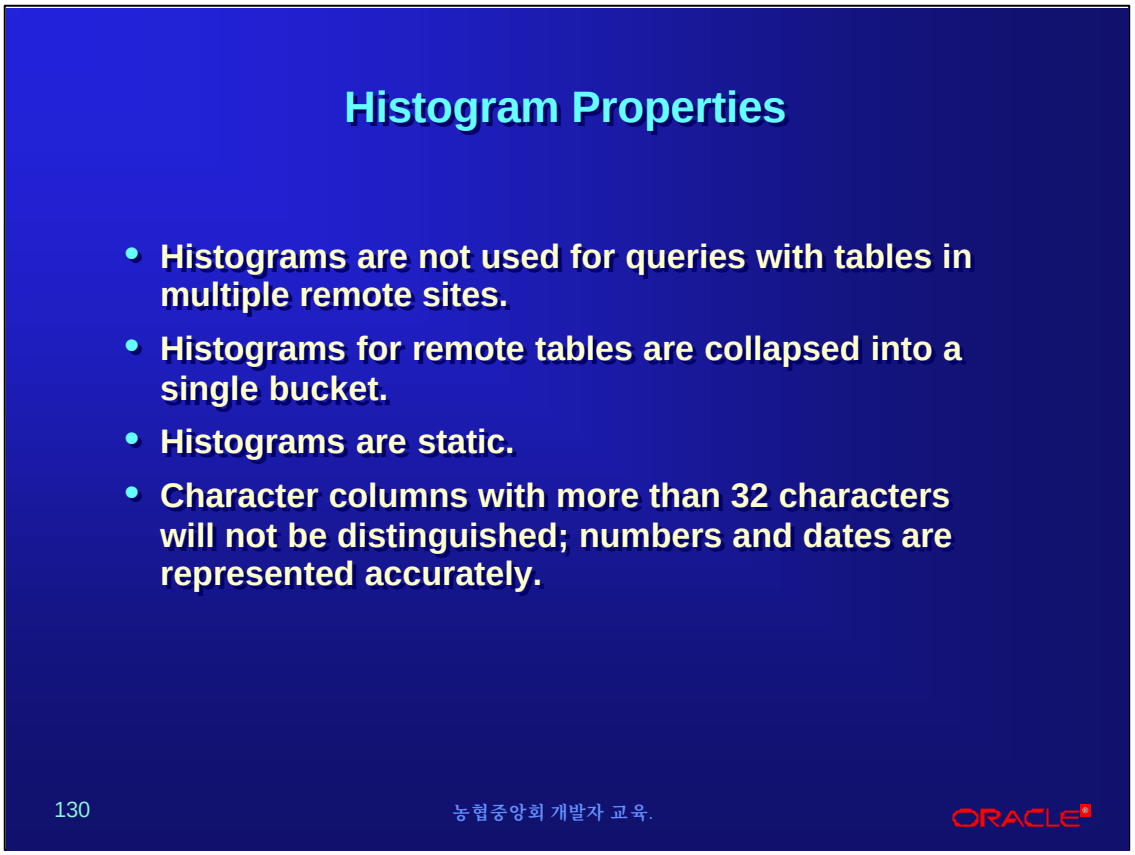


When to Use Histograms

You can analyze different columns with different bucket sizes.

Since Oracle7.3 you can analyze columns of interest. This results in more efficient use of temporary space and analyze operations complete faster because only the required subset of columns are analyzed.

Recompute only those columns whose distribution have changed, or only maintain table statistics.



Histogram Properties

Character columns have some exceptional behavior inasmuch as histogram data is stored only for the first 32 bytes of any string. Any predicates that contain strings greater than 32 characters will not use histogram information and the selectivity will be 1 / DISTCNT.

Note: Before Oracle 8.0.5 this limit was five characters.

Data in histogram endpoints is normalized to double precision floating point arithmetic.

5

Hints and Plan Stability(OUTLINE)

Hints Usage

- Use hints when there is a substantial gap between the assumptions made by the CBO and the actual data distribution and data pattern.
- You can use hints to influence:
  - The optimization approach
  - The access path for a table accessed
  - The join order and method

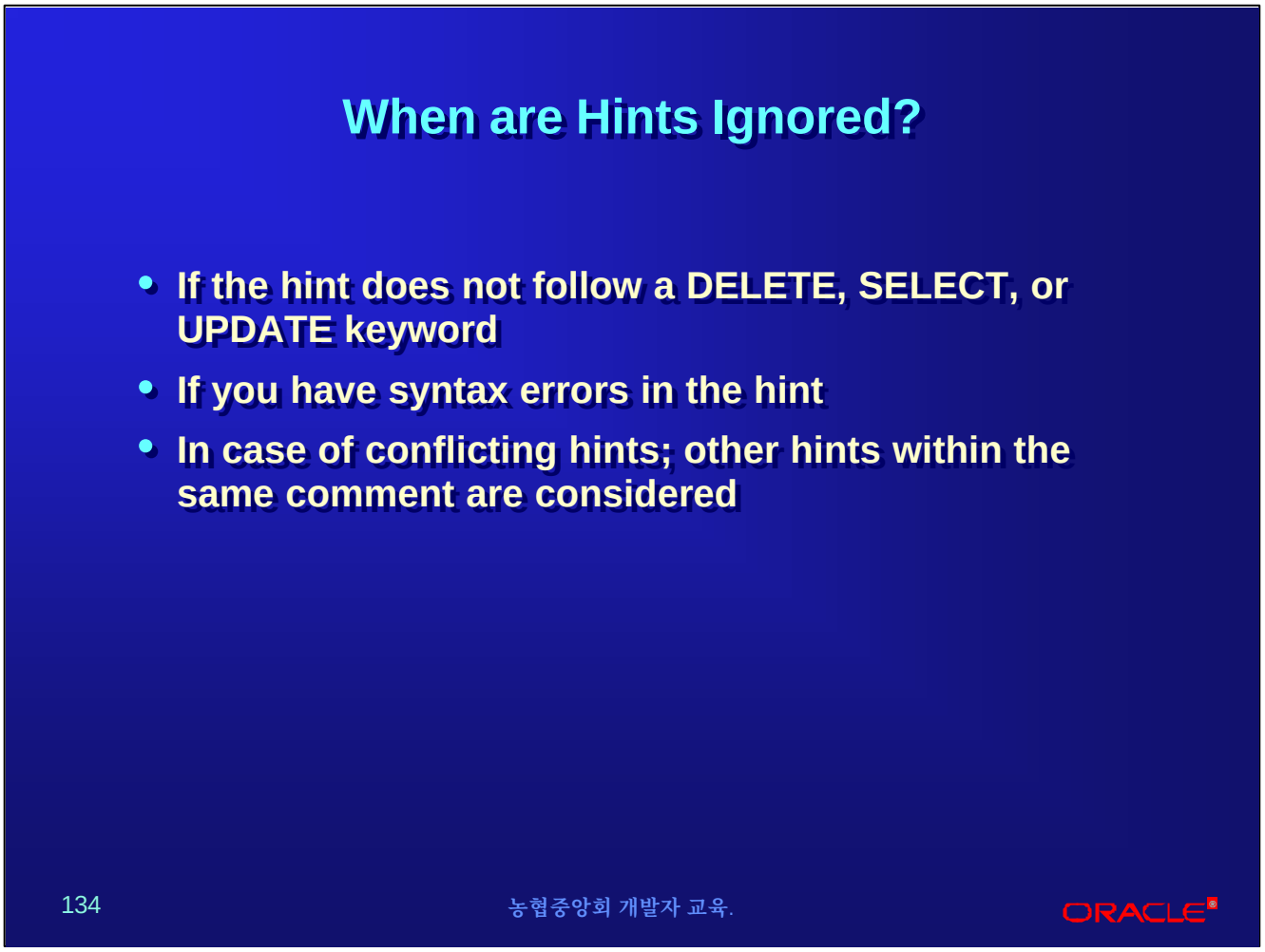
Hints Usage

- Hints can be used in any of the following:
  - Simple SELECT, UPDATE, or DELETE statements
  - Parent statements or subqueries of a complex statement
  - Parts of a compound query
- You can have only one comment containing hints per query block.

Hints

Hints are enclosed in comments to make SQL statements portable and ANSI/ISO compliant.

There is only one hint—the APPEND hint—that is used in conjunction with INSERT statements. You can specify this hint to use the *direct path* method for adding rows to a table.

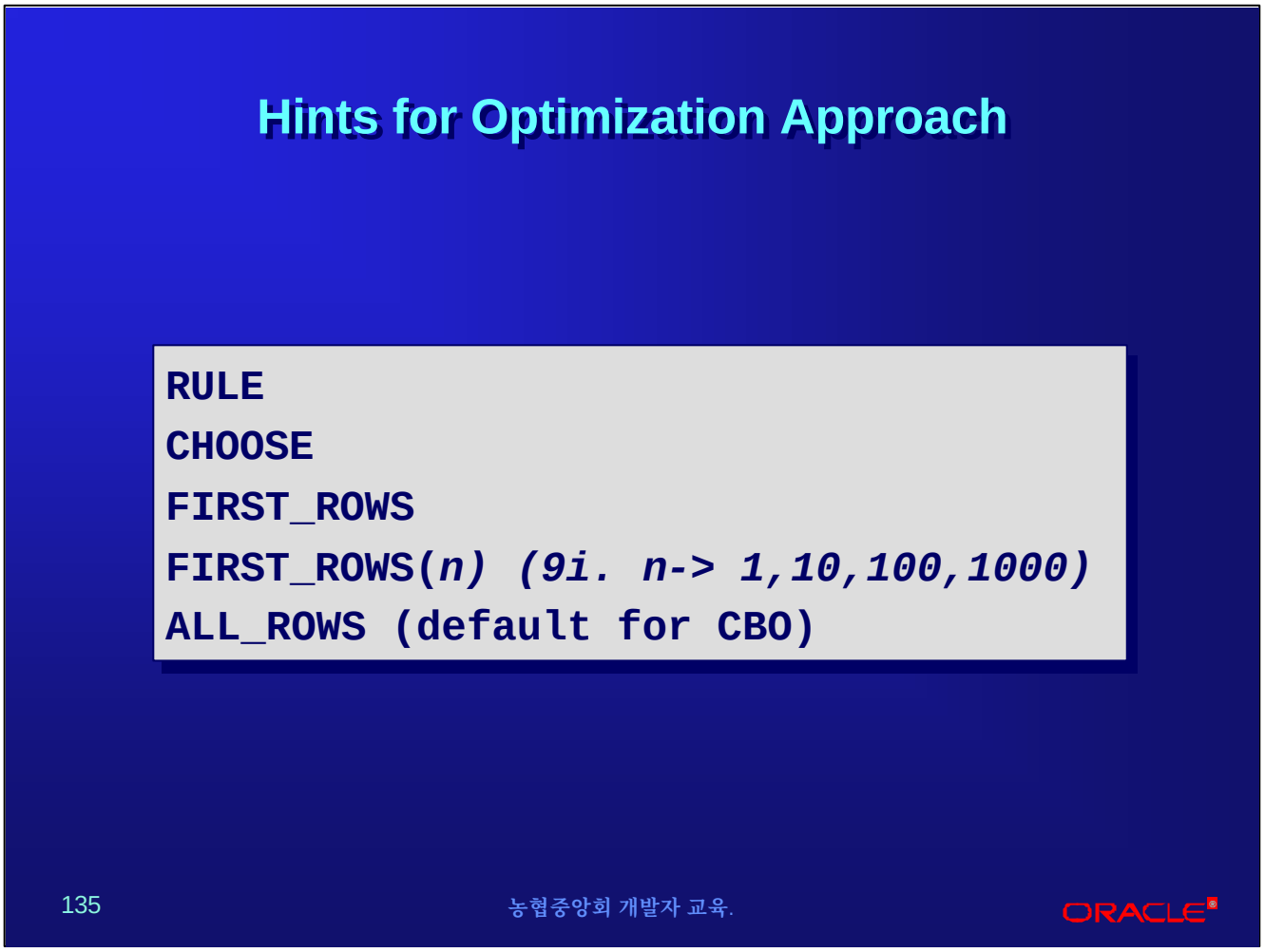


Optimizer Mode Hints and Views

The rules for the optimizer mode hint are as follows:

- If there is a mode hint in the top level query, it is used regardless of any mode hints in any referenced views.
- If there is no top-level 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 view mode hints conflict, all view mode hints are discarded and the session default is used.

The above rules also hold for views defined in terms of subviews and for set operations.



Optimizer Mode Hints

The FIRST_ROWS hint attempts to achieve optimal response time (favor index scans and NLJ). FIRST_ROWS only evaluates NLJ.

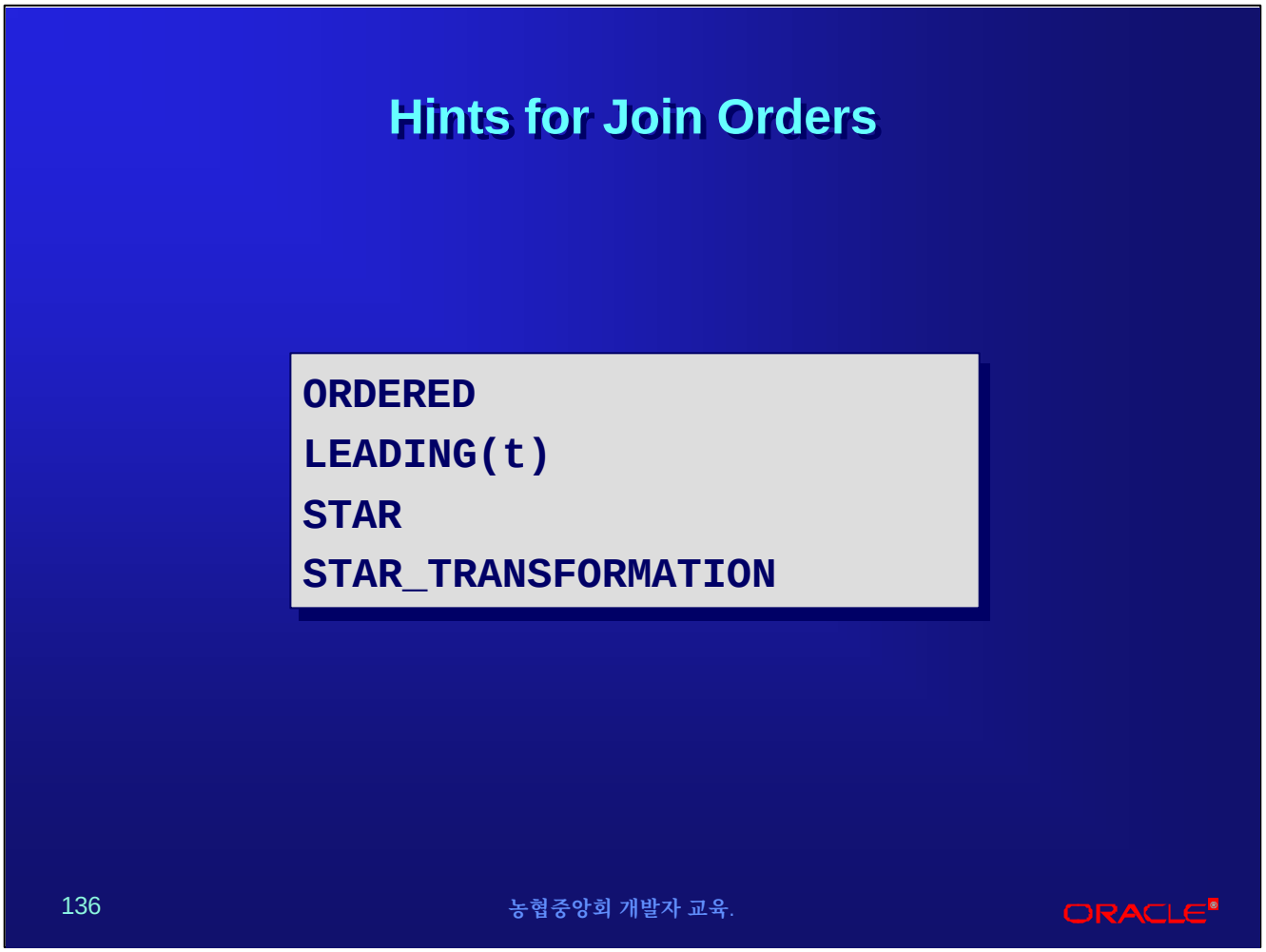
The ALL_ROWS hint attempts to achieve optimal throughput (favor full table scans and SMJ). The FIRST_ROWS hint causes the optimizer to make the following choices:

If an index scan is available:

- The optimizer will choose it over a full table scan
- The optimizer may choose a NLJ over a SMJ whenever the associated table is the potential inner table of the nested loops join
- For an ORDER BY clause, the optimizer may choose it to avoid a sort operation

FIRST_ROWS is ignored when:

- The FIRST_ROWS and USE_MERGE hints are defined as they conflict
- The FIRST_ROWS hint is defined but there is no index on the join column of the inner table
- There are set operators (UNION, INTERSECT, MINUS, UNION ALL)
- The statement contains a GROUP BY clause and group functions
- The statement contains a FOR UPDATE clause
- The statement contains a DISTINCT operator



Hints for Join Orders

You must specify the table to be accessed exactly 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.

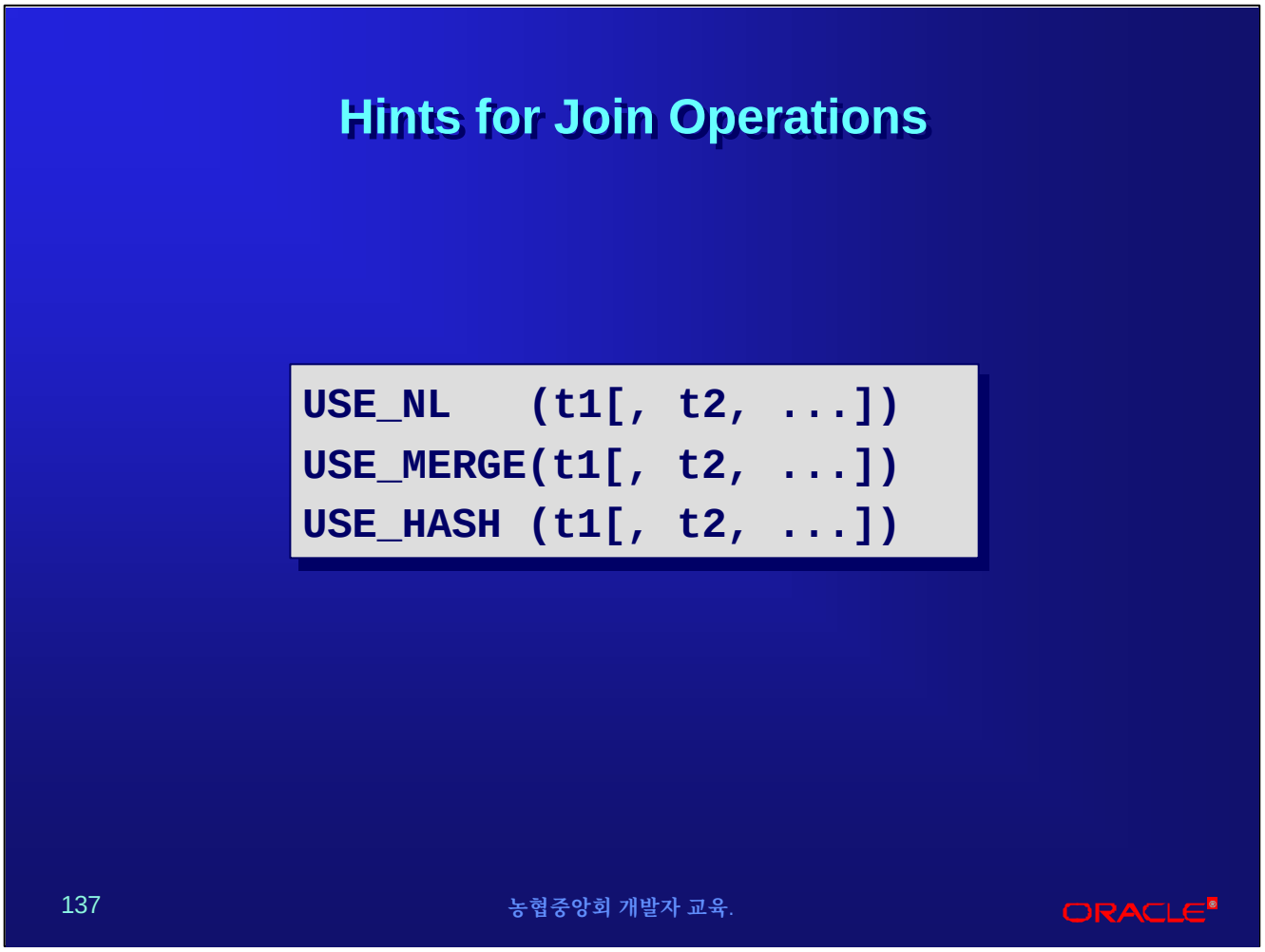
ORDERED and LEADING
The ORDERED hint forces to follow the table order in the FROM clause; the LEADING hint only specifies the driving table.

STAR
Force a star query plan 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 joined together and the large table's concatenated index has at least two columns and there are no conflicting access or join method hints.

STAR_TRANSFORMATION
There are several factors which control when the optimizer is able to perform a star transformation; first of all you need at least three tables in the query (two dimension tables and one fact table).

In the cases where the optimizer chooses to consider the star transformation, it only selects the star transformation plan if it is the cheapest. You can override this with the STAR_TRANSFORMATION hint; this causes the optimizer to select the star transformation regardless of cost.

Star transformation differs from star joins in that it is better suited for star schemas with a “sparse” fact table (only a few of the possible combinations of dimensions can be found in the fact table).



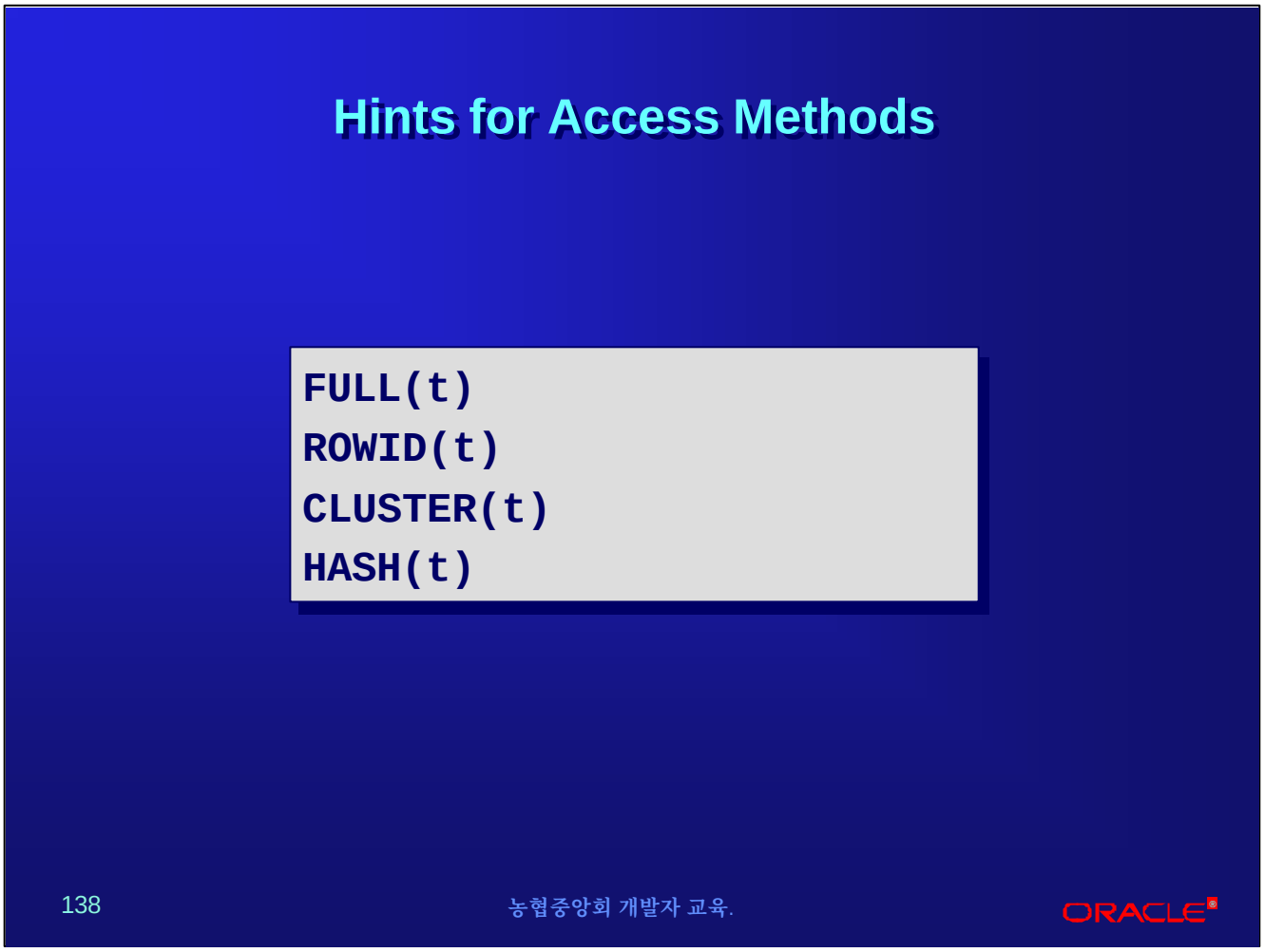
Hints for Join Operations

Hints for join operations must be specified on the *inner* table of the join order in order to be considered.

It is generally recommended to use join operation hints combined with an ORDERED hint. Otherwise the optimizer can choose a join order where the join operation would not make any sense.

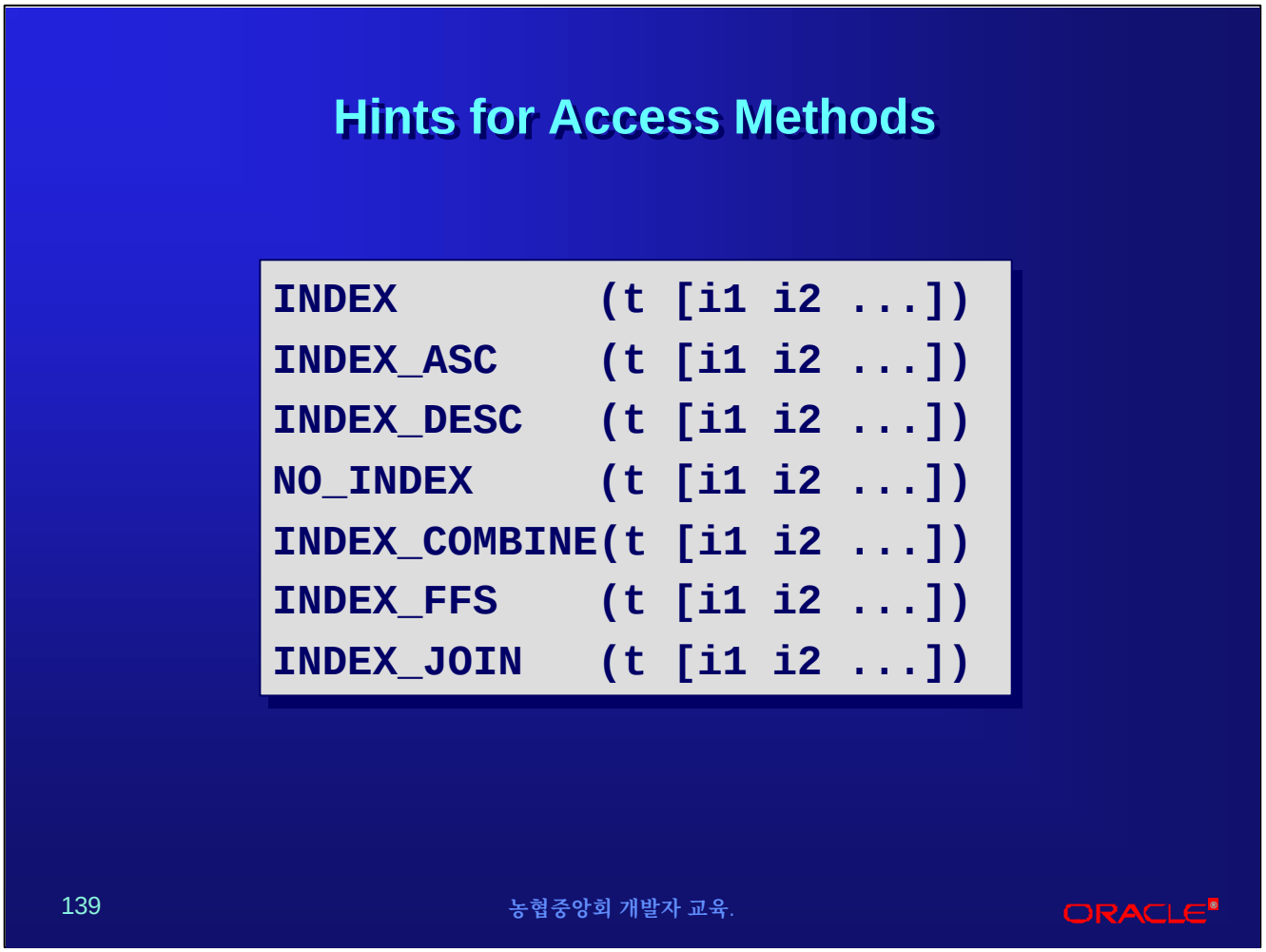
This happens when the optimizer decides that making the table specified in the join operation hint the *driving* table will result in the lowest cost.

This behaviour looks like the optimizer is ignoring the USE_% hints.



Hints for Access Methods

The HASH hint only applies to tables stored in a cluster with the HASHKEYS parameter specified.



INDEX and INDEX_ASC

If these hints specify a single available index, the optimizer performs a scan on this index.

If these hints specify a list of available indexes, the optimize r considers the cost of a scan on each index in the list and then performs the index scan with the lowest cost. The optimizer may also choose to scan multiple indexes from this list and merge the results, if such an access path has the lowest cost.

If these hints specify 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.

INDEX_DESC

The INDEX_DESC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, Oracle scans the index entries in descending order of their indexed values.

This hint has no effect on SQL statements that access more than one table.

NO_INDEX

This is the opposite of the INDEX hint.

INDEX_COMBINE

If no indexes are given as arguments for the INDEX_COMBINE hint, the optimizer will use on the table whatever Boolean combination of bitmap indexes has the best cost estimate. If certain indexes are given as arguments, the optimizer will try to use some Boolean combination of those particular bitmap indexes.

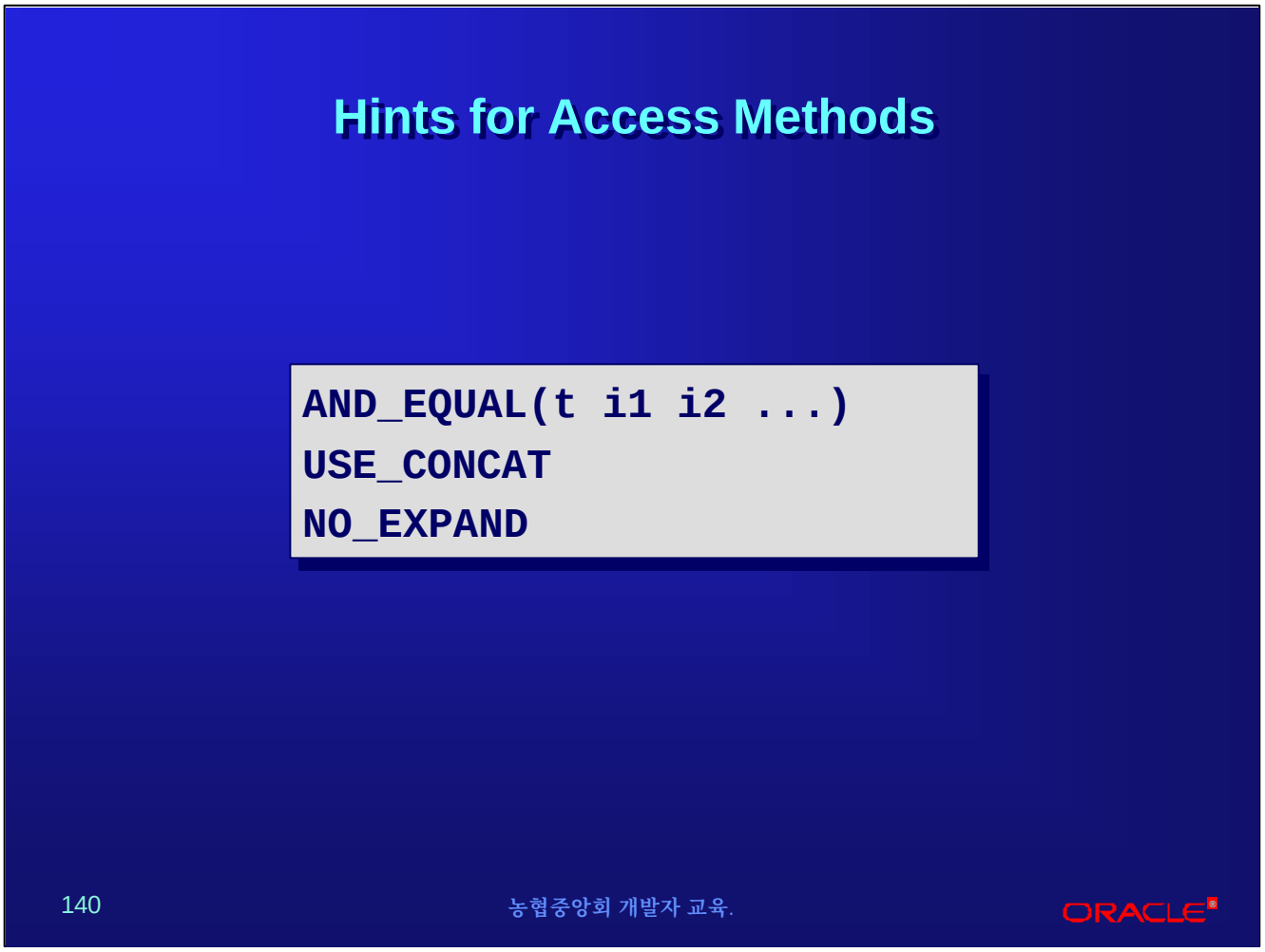
INDEX_FFS hint

If you specify a single index in the hint, the optimizer performs a fast full index scan on the index rather that a full table scan.

If you specify more than one index, it calculates the cost for each index scan and arrives at the index which gives the lowest cost.

If you do not specify an index in the hint, the optimizer considers the cost of index scans associated with the table and arrives at the optimum cost.

If a cluster index is present in the table, CBO overrides all the indexes given in the hint



AND_EQUAL

Choose an access path that merges the scans on several single-column indexes. The maximum number of indexes that can be merged is five.

USE_CONCAT

If the leading predicates of a concatenated index are join predicates and trailing predicates are ORs, the high cost of driving table forces OR optimization to be turned off.

It is not the high cost of the driving table as such, but rather the high cost of *repeatedly* accessing the table. Joins and driving tables have nothing to do with this; you can get this performance problem with a single table too.

Note: OR expansion is only done for branches that have an index driver.

NO_EXPAND

Very long inlists can cause problems for the cost-based optimizer especially when the inlist is expanded into a large number of UNIONed statements. This is because the CBO has to determine the cost for the expanded statement which is time consuming because of the large number of branches.

You can force the CBO not to expand by using the NO_EXPAND hint.

Note: The USE_CONCAT and the NO_EXPAND hints are the opposite of eachother.

Hints for Subqueries

HASH_AJ

MERGE_AJ

HASH_SJ

MERGE_SJ

PUSH_SUBQ

Antijoins (NOT IN Subqueries)

```
SQL> select e.*
  2  from   emp e
  3  where  e.deptno NOT IN (select /*+ HASH_AJ */ d.deptno
  4                          from   dept d
  5                          where  d.loc = 'DALLAS');
```

Semijoins (EXISTS Subqueries)

```
SQL> select d.*
  2  from   dept d
  3  where  EXISTS (select /*+ HASH_SJ */ 'x'
  4                 from   emp e
  5                 where  e.deptno = d.deptno
  6                 and    e.sal > 200000);
```

Antijoins and semijoins can be solved more efficiently by using a hash or merge operation instead of the default nested loops operation (often seen as a FILTER operation in the execution plan).

PUSH_SUBQ

This hint causes subqueries to be evaluated at the earliest possible time in a query block. Normally subqueries are executed as the last operation.

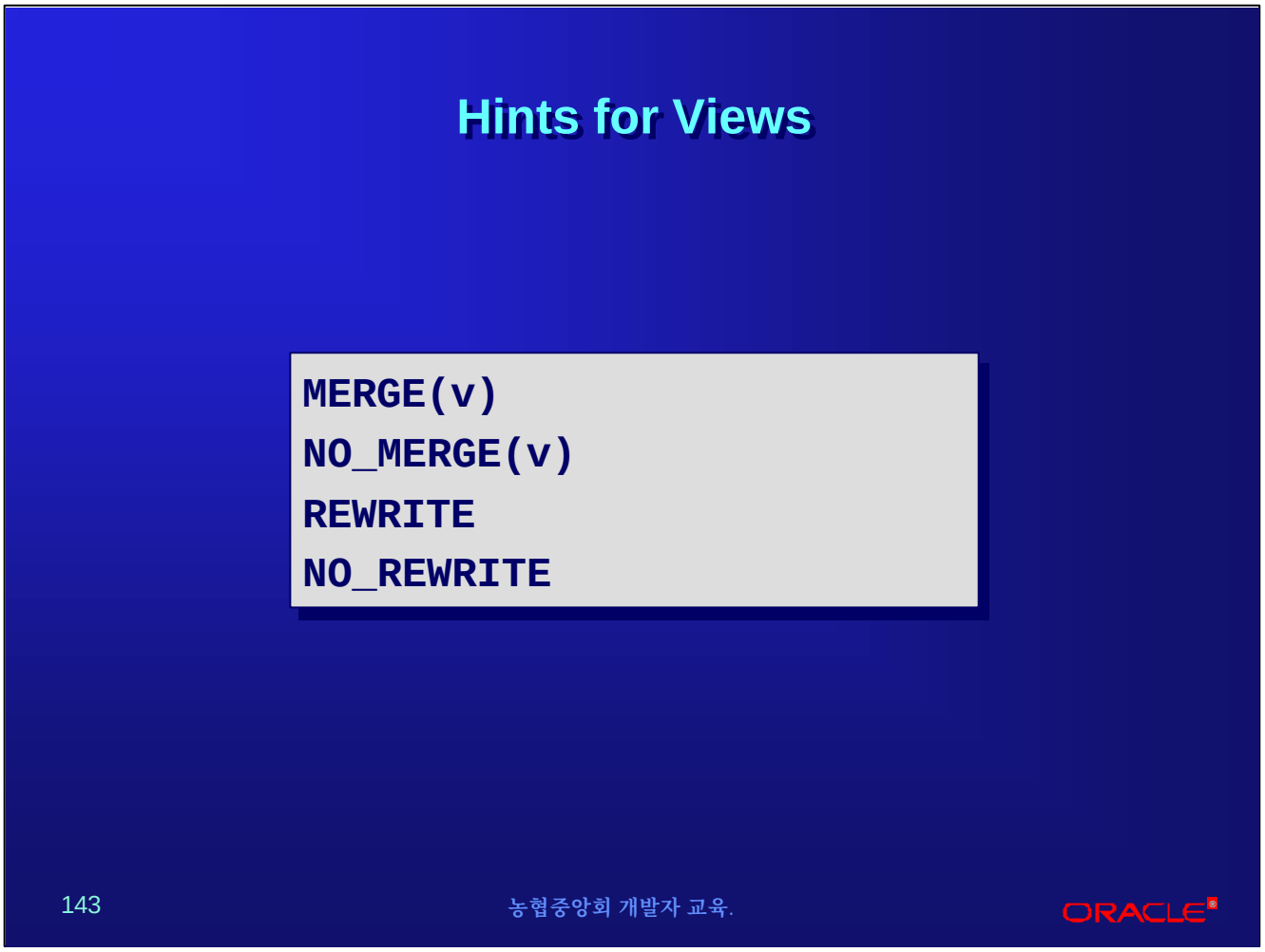
Example: Subquery Without PUSH_SUBQ Hint

```
SQL> select count(*)
  2  from   courses
  3  where  dev_id = (select instr_id
  4                   from   classes
  5                   where  class_id = 12501);
Execution Plan
----------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS
1 0   SORT (AGGREGATE)
2 1    FILTER
3 2      TABLE ACCESS (FULL) OF 'COURSES'
4 2      TABLE ACCESS (BY INDEX ROWID) OF 'CLASSES'
5 4        INDEX (UNIQUE SCAN) OF 'CLS_PK' (UNIQUE)
```

Notice the FILTER operation and that the subquery is evaluated for each row in COURSES.

Same Example With PUSH_SUBQ Hint

```
SQL> select /*+ push_subq */ count(*)
  2  from   courses
  3  where  dev_id = (select instr_id
  4                   from   classes
  5                   where  class_id = 12501);
Execution Plan
----------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS
1 0   SORT (AGGREGATE)
2 1    TABLE ACCESS (FULL) OF 'COURSES'
3 2     TABLE ACCESS (BY INDEX ROWID) OF 'CLASSES'
4 3      INDEX (UNIQUE SCAN) OF 'CLS_PK' (UNIQUE)
```



MERGE

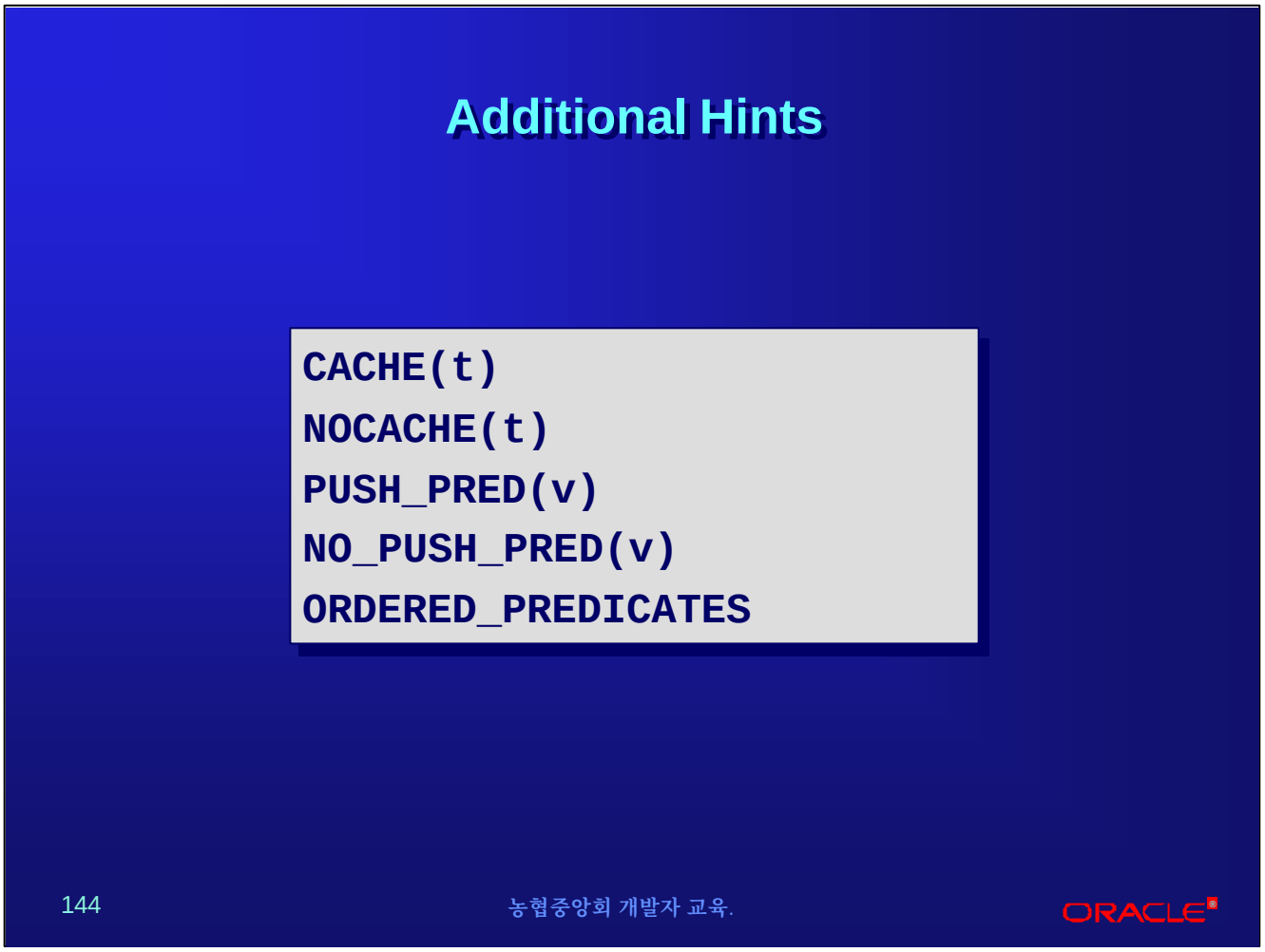
This hint makes the optimizer merge a complex view into the statement before optimization.

NO_MERGE

This hint disables view merging and can be used to gain control over how a view is optimized.

REWRITE and NO_REWRITE

Controls whether a query is rewritten to use materialized views.



PUSH_PRED and NO_PUSH_PRED

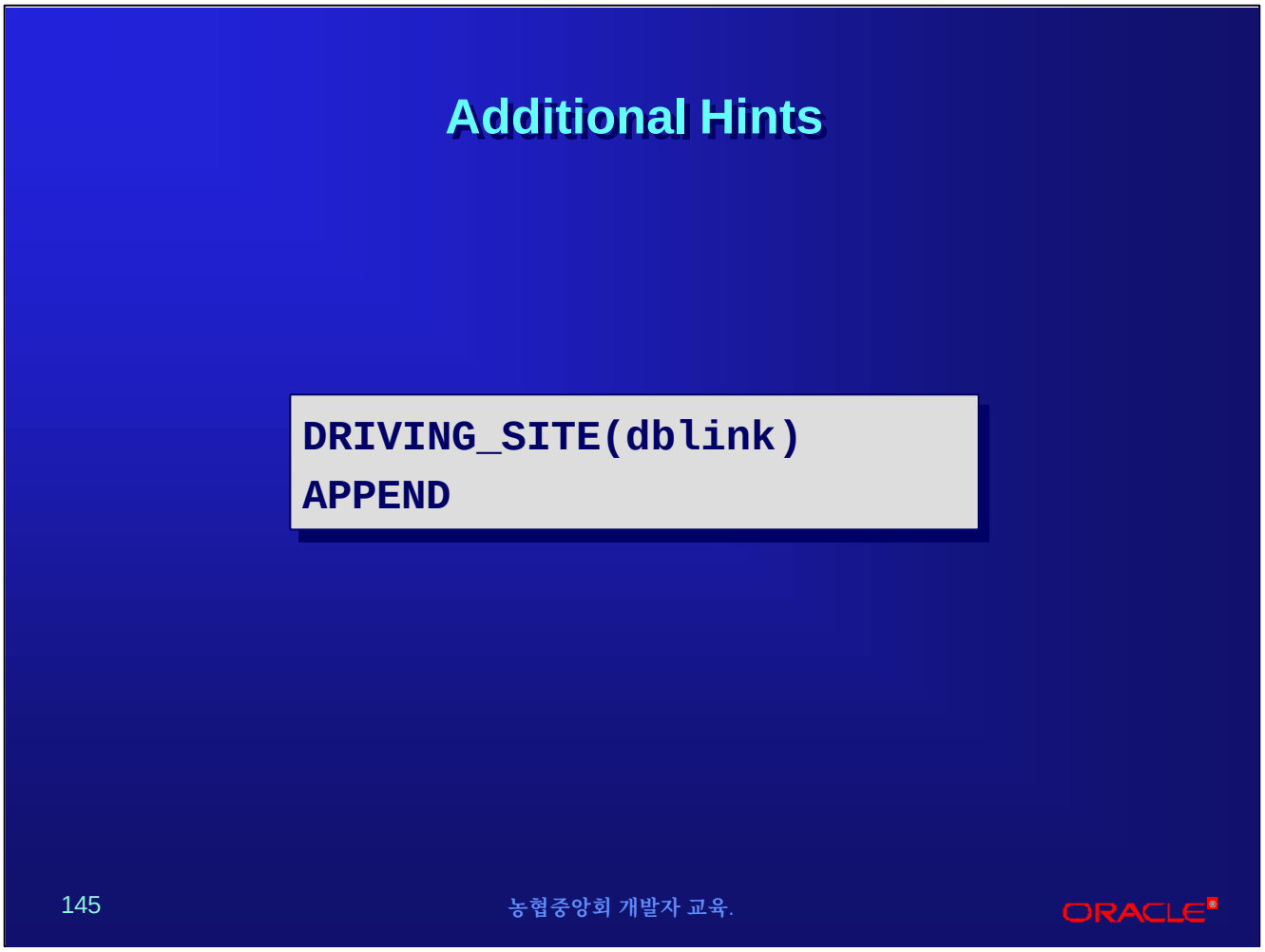
Respectively forces and prevents join predicates from being pushed into a view.

```
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;
```

ORDERED_PREDICATES

This hint forces the optimizer to preserve the order of predicate evaluation, except for predicates used as index keys. If you do not use the ORDERED_PREDICATES hint, then Oracle evaluates all predicates in the order specified by the following rules. Predicates:

- Without user-defined functions, type methods, or subqueries are evaluated first, in the order specified in the WHERE clause.
- With user-defined functions and type methods that have user-computed costs are evaluated next, in increasing order of their cost.
- With user-defined functions and type methods without user-computed costs are evaluated next, in the order specified in the WHERE clause.
- Not specified in the WHERE clause (for example, predicates transitively generated by the optimizer) are evaluated next.
- With subqueries are evaluated last in the order specified in the WHERE clause.



DRIVING_SITE

Specifies on which node in a distributed query the result set should be initially built before being returned to the node issuing the query.

APPEND

This is the only hint not involved in query management. It specifies that an INSERT ... SELECT statement should use the direct path method for writing data blocks:

```
SQL> insert /*+ APPEND */
  2  into   employees
  3  select *
  4  from   old_employees;
```

Global and Local Hints

- Extended hint syntax allows for specifying (global) hints through views.
- Use a dot notation to reference a table.

```
SQL> create view instructors as
  2  select *
  3  from   employees e
  4  where  job like '%INSTRUCTOR%';

SQL> select /*+ index(i.e emp_sal_idx) */
  2         i.last_name, i.salary
  3  from   instructors i
  4  where  i.salary > 5000;
```


Extended Hint Syntax

Note that the *Oracle8i Designing and Tuning for Performance Release 2 (8.1.6)*, chapter 7 states that the global hints also apply for subqueries.

This only applies for subqueries in FROM clauses—also known as inline views—and not for subqueries in the WHERE clause; this confusion comes from the ANSI definition of a "subquery".

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

Hints and Optimizer Mode

- All hints force the cost-based optimizer to be activated
- The only exceptions are:
  - The RULE hint
  - The DRIVING_SITE hint

Hints and Optimizer Mode

This means that just specifying a simple index hint in a complex join will change the optimizer from RULE to the ALL_ROWS goal using hard-coded defaults for tables not analyzed.

Plan Stability

- Plan stability forces the optimizer to use the same unvarying execution plan for a statement.
- This is implemented using stored outlines in the OUTLN schema.
- Procedure:
  - Create stored outlines.
  - Run your application with activated category.

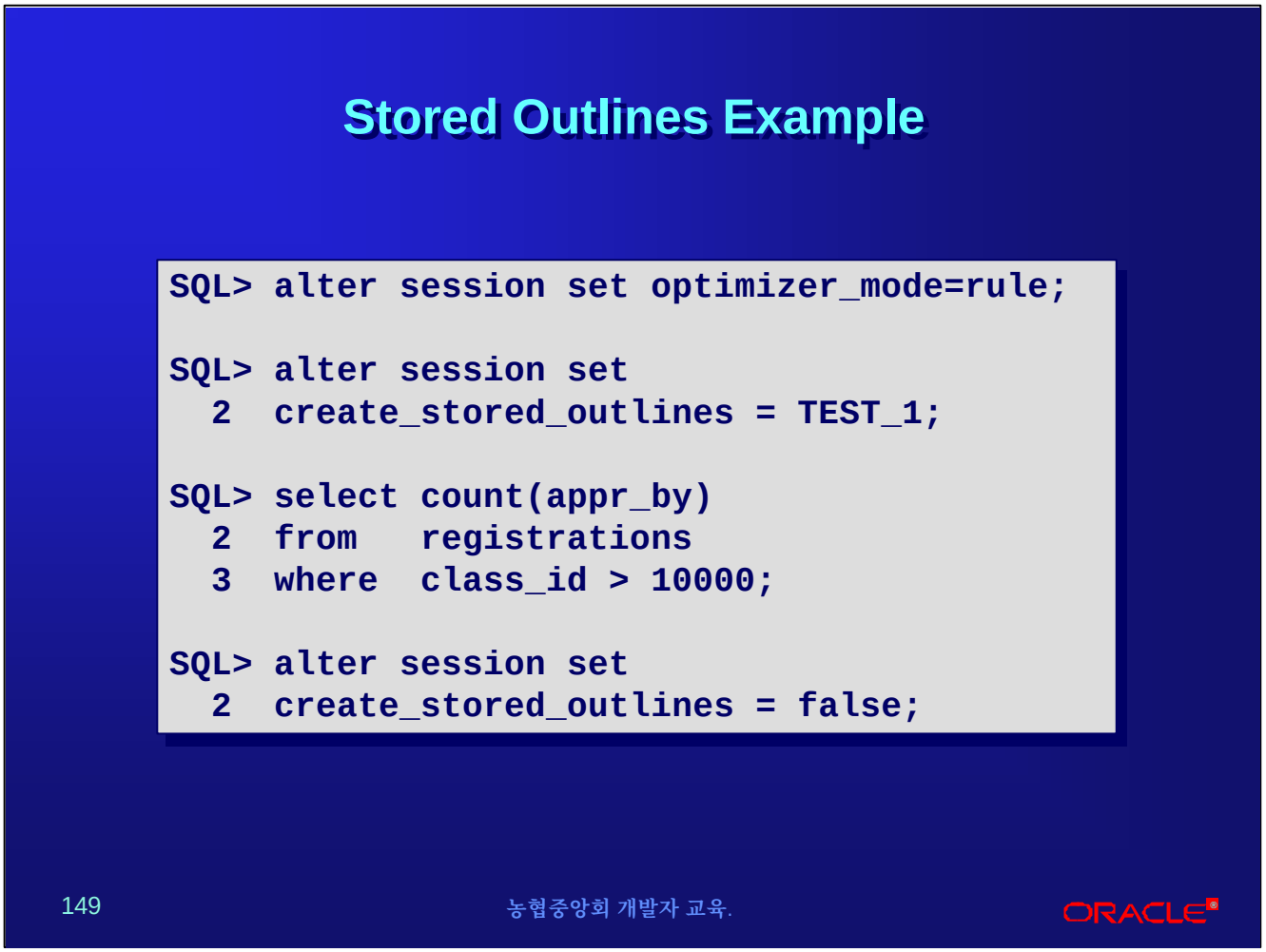
Plan Stability

A stored outline is a representation of the execution plan for a statement. You can group stored outlines in categories. To use the execution plans of your stored outlines your session (or the instance) must have the corresponding category activated.

Stored outlines can prevent changes in optimizer mode, statistics, data volumes, data distribution, and creation of new indexes from changing the execution plan of SQL statements.

Stored outlines cannot handle differences in several initialization parameters, especially some of those ending with *_enabled*, such as *query_rewrite_enabled*, *star_transformation_enabled*, and *optimizer_features_enabled*.

This means that these parameters must be consistent across execution environments for stored outlines to function properly.



Stored Outlines Example

Set the OPTIMIZER_MODER to RULE, and enable the creation of stored outlines for your session. Suppose the query on the REGISTRATIONS table has the following execution plan:

```
Execution Plan
----------------------------------------------------
SELECT STATEMENT Optimizer=RULE
  SORT (AGGREGATE)
    TABLE ACCESS (BY INDEX ROWID) OF 'REGISTRATIONS'
      INDEX (RANGE SCAN) OF 'REG_PK' (UNIQUE)
```

Disable the creation of stored outlines, and change your session environment:

```
SQL> alter session set optimizer_mode=all_rows;
```

If you run the same SQL statement again, you probably see the following execution plan:

```
Execution Plan
----------------------------------------
SELECT STATEMENT Optimizer=ALL_ROWS
  (Cost=43 Card=1 Bytes=11)
  SORT (AGGREGATE)
    TABLE ACCESS (FULL) OF 'REGISTRATIONS'
```

Stored Outlines Example (continued)

Now activate the stored outline category TEST_1:

```
SQL> alter session set use_stored_outlines = TEST_1;
```

Run the same SQL statement again:

```
Execution Plan
----------------------------------------------------------
--
SELECT STATEMENT Optimizer=ALL_ROWS (Cost=11779 Card=1
Bytes=11)
  SORT (AGGREGATE)
    TABLE ACCESS (BY INDEX ROWID) OF 'REGISTRATIONS'
      INDEX (RANGE SCAN) OF 'REG_PK' (UNIQUE)
```

As you can see, the execution plan from the RBO environment reappears even if the normal indicators of the CBO (costs in the execution plan) are present.

Storing Outlines

Stored outlines can be generated for a single statement or at the session or the instance level

```
ALTER SESSION set create_stored_outlines
            = {TRUE|category|FALSE}
```

```
ALTER SYSTEM set create_stored_outlines
            = {TRUE|category|FALSE}
              [NOOVERRIDE]
```

```
CREATE OUTLINE outline
[FOR CATEGORY category] ON statement
```


CREATE_STORED_OUTLINES

By setting this parameter to TRUE or a category name, all subsequent SQL statements in your session (or the instance) generate stored outlines. If this parameter is set to an existing category, then new outlines will be added to that category.

If no category name is specified, the stored outlines are added to the DEFAULT category.

If you do not specify NOOVERRIDE, this system setting takes effect in all sessions.

Setting this parameter to FALSE stops the generation of outlines.

Existing outlines in the category are *not* updated if these should happen to be executed again with a different execution plan during the period of storing outlines in the category.

CREATE OUTLINE

This command creates (or replaces) the stored outline for the specified statement without executing the statement.

Note: Outline names must be unique across the database.

Data Dictionary

- Stored outlines are stored as a set of hints necessary to reproduce the execution plan.
- Data dictionary views:

```
{USER|ALL|DBA}_OUTLINES
{USER|ALL|DBA}_OUTLINE_HINTS
```


Columns in USER_OUTLINES

NAME	Outline name (SYS_OUTLINE_*nnn* if not specified with CREATE OUTLINE)
CATEGORY	Category name (DEFAULT if not specified)
USED	USED/UNUSED depending whether the stored outline has ever been used
VERSION	Oracle version
SQL_TEXT	The statement

Columns in USER_OUTLINE_HINTS

NAME	Outline name (SYS_OUTLINE_*nnn* if not specified with CREATE OUTLINE)
NODE	Query or subquery ID to which the hint applies. The top-level query is labeled 1; subqueries are assigned sequentially numbered labels, starting with 2
STAGE	Outline hints can be applied at three different stages during the compilation process; this column indicates at which stage this hint was applied
JOIN_POS	Position of the table in the join order; the value is 0 for all hints except access method hints, which identify a table to which the hint and the join position apply
HINT	Text of the hint

Hints for Stored Outlines

- Outline hints are not limited to the documented hints:
  - %_OUTLINE_HINTS
- Outline hints cannot be edited

153

농협중앙회 개발자 교육.

ORACLE

Hints Used In Stored Outlines

Note that the normal, documented hints will not be able to reproduce all execution plans.

To illustrate this, consider the following two examples:

1. If the statement is a join, it is necessary to be able to specify the join order *without* changing the sequence of tables in the FROM clause; the ORDERED hint will not suffice in this case as it simply ties the join order to the sequence of tables in the FROM clause.
2. It is impossible to specify hints into a subquery embedded in a view.

To solve these problems stored outlines not only employ hints not supported as normal hints (for example, NO_FACT) but they also use information on where in the statement the hints apply (the NODE column), in what sequence tables should be joined, and at what stage in the statement compilation the hint should be applied.

For reasons of complexity it should therefore be obvious that direct editing of the contents in the underlying table OUTLN.OL$HINTS is not supported nor encouraged.

However, it can be disputed if copying an entire set of hints for one outline into another is recommended or not. For further information, see Note:92202.1 which describes a method for doing this.

Stored Outlines Usage

Stored outlines are used when:

- **An outline category is enabled**

```
alter session set use_stored_outlines
      = {category|TRUE}
alter system set use_stored_outlines
      = {category|TRUE} [NOOVERRIDE]
```

- **The executed statement can be found as a direct match of a statement in the outline category.**

Stored Outlines

When setting *use_stored_outlines* to TRUE, the category named DEFAULT will be used.

For matching statements the same strict rules apply as for state ment matching in the shared pool with the exception of optimizer mode.

Stored Outlines Usage

- Stored outlines are not used when:
  - CURSOR_SHARING = FORCE
  - A hint in the stored outline becomes invalid.
- V$SQL.OUTLINE_CATEGORY
  - Contains outline category if used

CURSOR_SHARING and Stored Outlines

CURSOR_SHARING = FORCE disables usage of stored outlines.

This is a new Oracle8*i* Release 2 feature that internally replaces literals with bind variables, thus reducing the consequence of running an application that parses many different statements due to the usage of literals (see the lesson “Query Execution Overview”).

In the first versions of Oracle8*i* (before production) the two technologies could be used together, but it resulted in unacceptable response times and was for technical reasons changed to the deactivation of stored outlines when CURSOR_SHARING = FORCE.

Tests have shown that in contrast to normal hints where if one hint fails—for example, an index has been removed—the rest of the hints are still considered, stored outlines will not be used if not all hints are valid.

To test whether a statement did indeed use a stored outline, query the column OUTLINE_CATEGORY in V$SQL (does not exist in V$SQLAREA); if this column is NULL the statement did not use a stored outline.

OUTLN_PKG

For maintenance of categories the OUTLN_PKG package is supplied, with the following procedures:

- DROP_UNUSED
- DROP_BY_CAT (*category*)
- UPDATE_BY_CAT(*old_category*, *new_category*)

DROP_UNUSED

Drops all stored outlines that have never been used (USED column in %_OUTLINES).

DROP_BY_CAT

Drops all stored outlines within the specified category.

UPDATE_BY_CAT

Changes all the outlines in a category to another category. If an outline is defined for a SQL statement which already has an outline defined in the new category, this outline will not be moved from the old category.

Exporting and Importing Outlines

- Export and import of the tables used for stored outlines is supported.
- %_OUTLINES is based on OUTLN.OL$
- %_OUTLINE_HINTS is based on OUTLN.OL$HINTS
- This enables portability of stored outlines between databases.

Exporting With a Condition

By using the QUERY parameter of the export utility it is possible to export a particular outline or category to later import into another database.

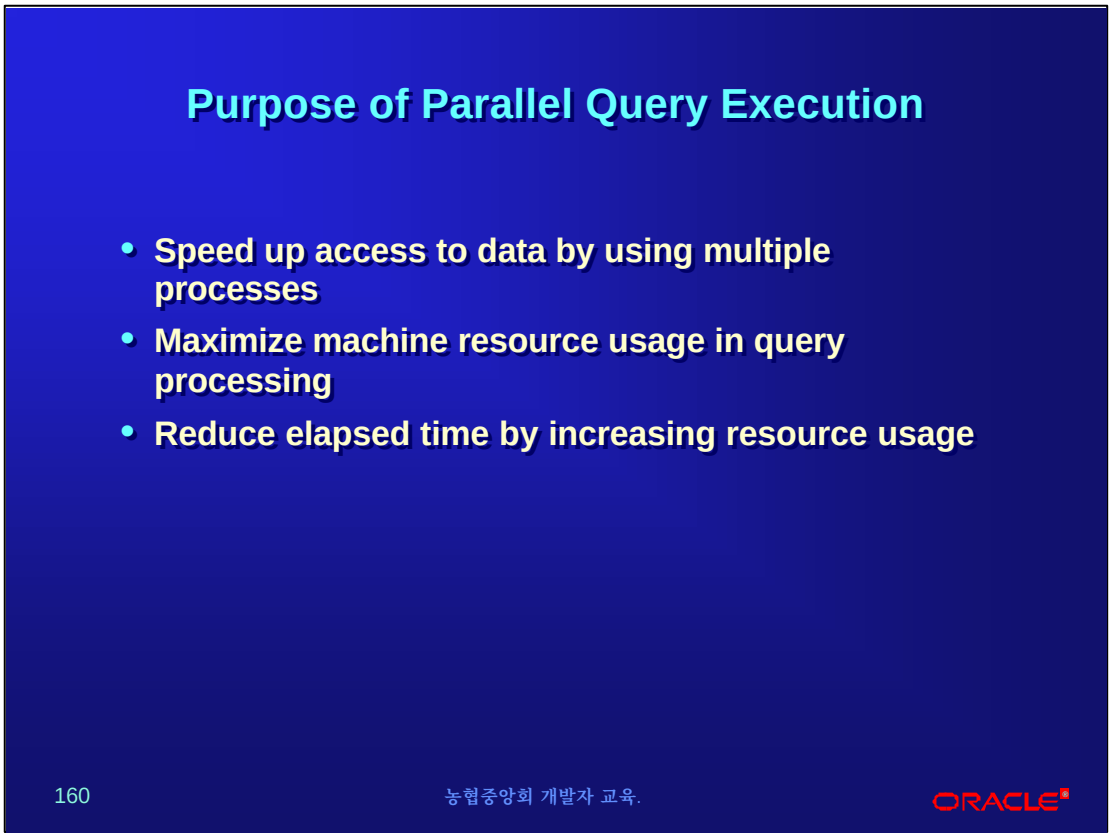
Windows NT example:

```
exp outln/outln
    FILE=exportfile
    TABLES='OL$' 'OL$HINTS'
    QUERY=\"WHERE CATEGORY='category'\"
```

Plan Stability (Stored Outline #1)

```
coenect sys/manager
grant create any outline to scott;
connect scott/tiger
alter session set
CREATE_STORED_OUTLINES=TESTOUTLINE;
select * from emp,dept where emp.deptno = dept.deptno;
alter session set CREATE_STORED_OUTLINES=false;

connect outln/outln
select * from OL$;
sleect * from OL$HINTS;


connect scott/tiger
alter session set USE_STORED_OUTLINE=TESTOUTLINE;
select * from emp,dept where emp.deptno = dept.deptno;
Select * from emp,dept where emp.deptno = dept.deptno;

connect sys/manager
select * from V$SQL;
```

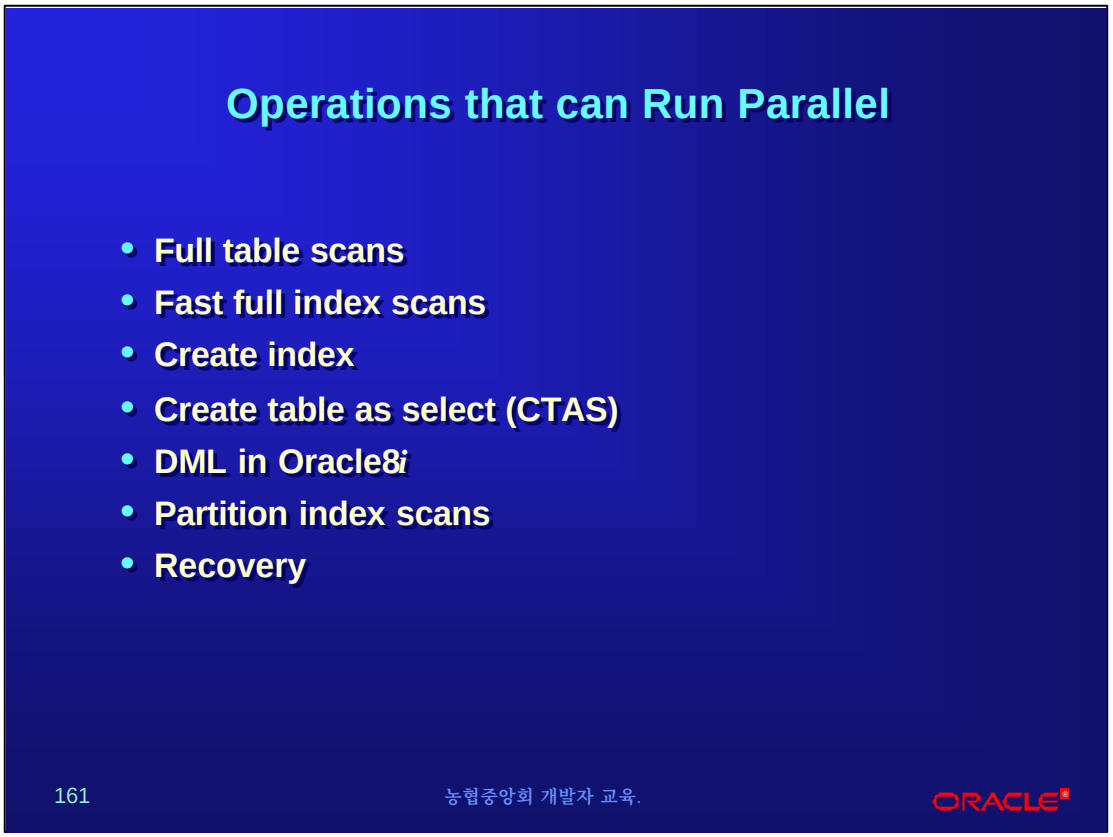
Schedule:	Timing	Topic
	60 minutes	Lecture
	60 minutes	Total



Purpose of Parallel Query Execution

In order for PQ to increase performance the server will need spare CPU capacity.

PQ can make statements run slower if there are insufficient resources. This can occur in any parallel implementation due to overheads such as process startup and interprocess communication.

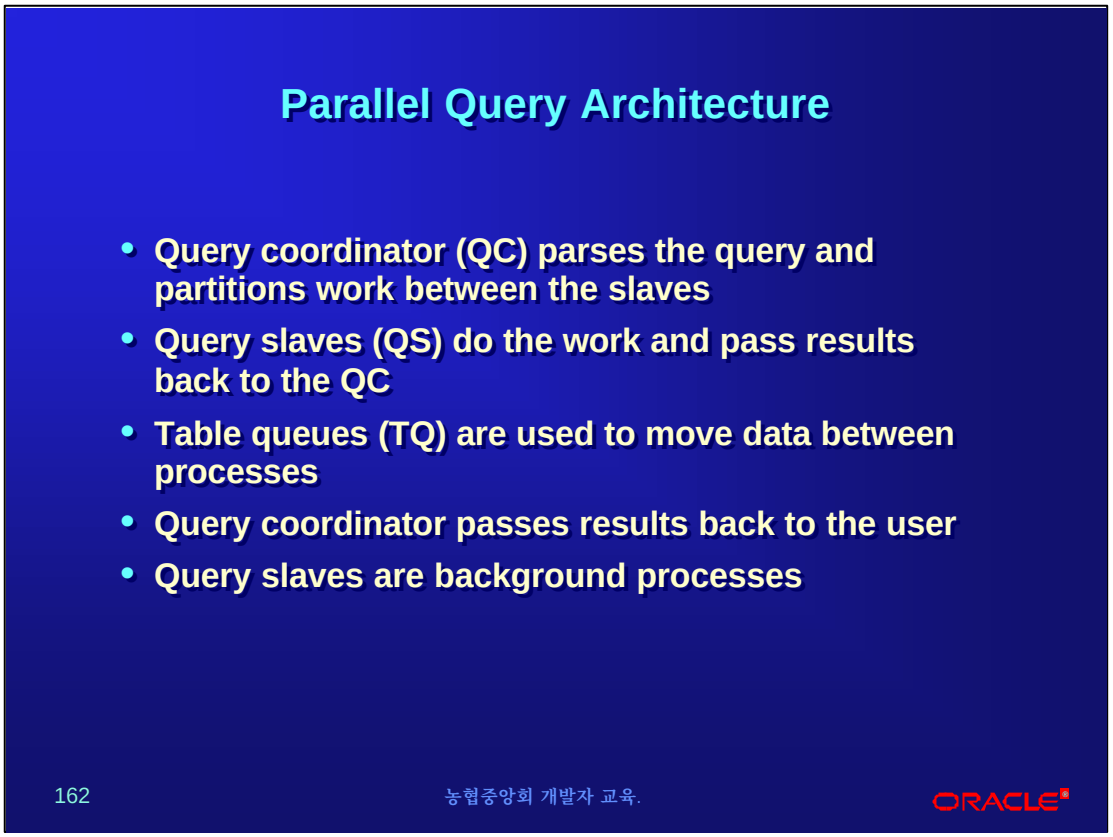


Operations That Can Run Parallel

Once a query has started in parallel then subsequent steps may also run in parallel, for example GROUP BY and JOIN steps that are driven by the first full table scan. Of course these steps may still run serially.

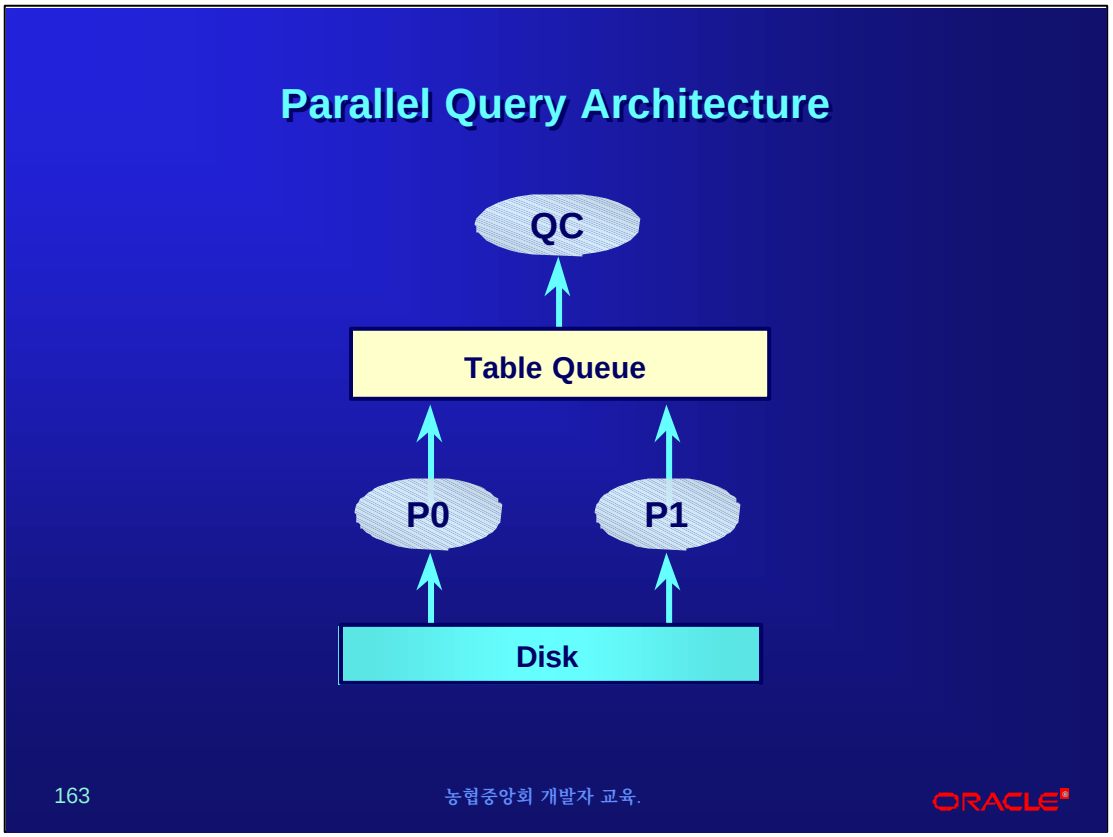
Normal Index Scans

The leaf blocks for a particular key or key range will not necessarily be physically located next to each other. Also it is often not possible to determine which blocks will be read until the read has started.



Parallel Query Architecture

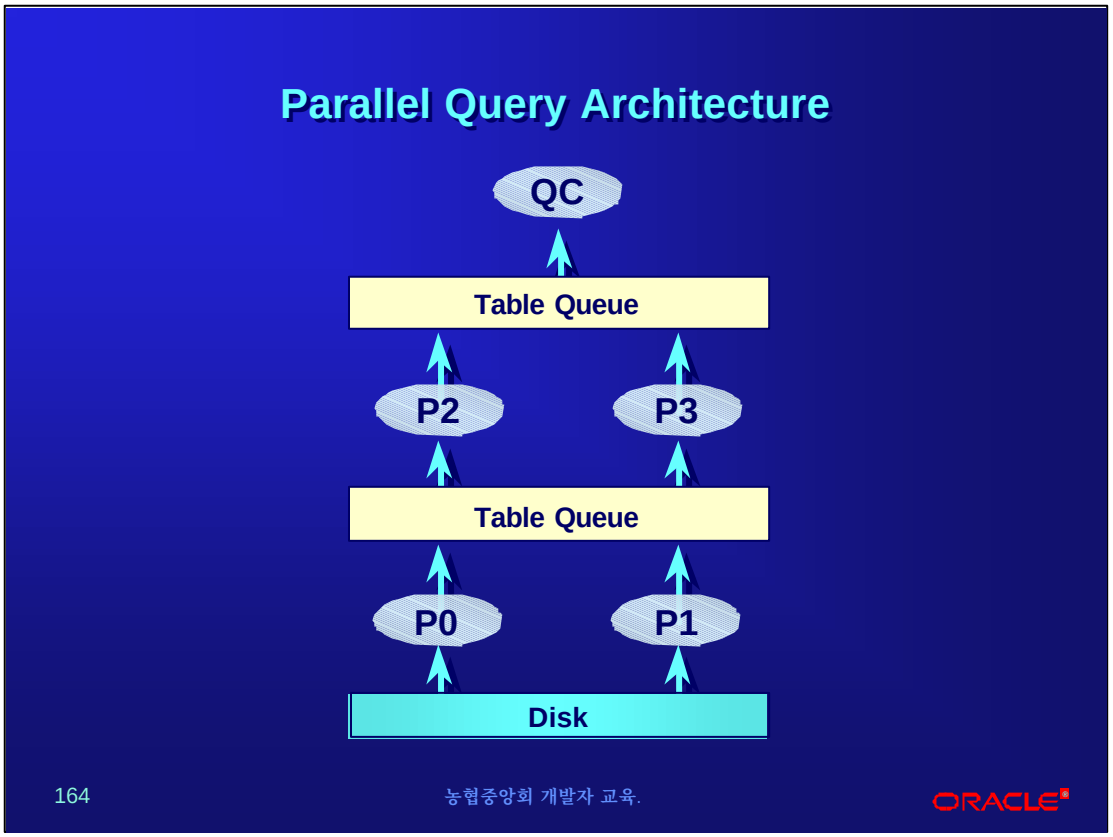
The following slides contain more details on each of these parts of the PQ architecture.



Parallel Query Architecture

The example above has one slave set.

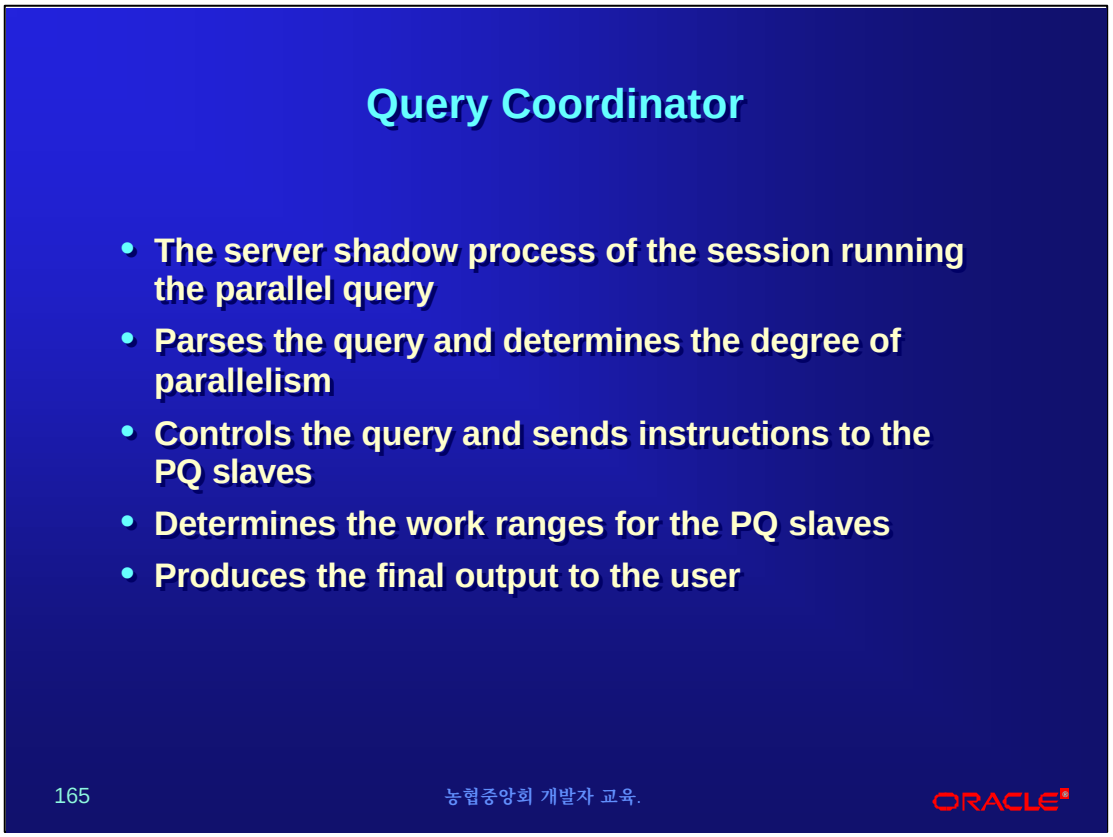
The slave set processes (P0 and P1) are reading data from disk. They send this data to the query coordinator (QC) using a table queue (TQ); the query coordinator presents the results to the client program.



Parallel Query Architecture (continued)

The example above has two slave sets.

Slave set one (P0 and P1) are reading data from disk. They send this data to slave set two (P2 and P3) using a table queue. Finally the query coordinator (QC) reads the results from a second table queue before presenting them to the client program.

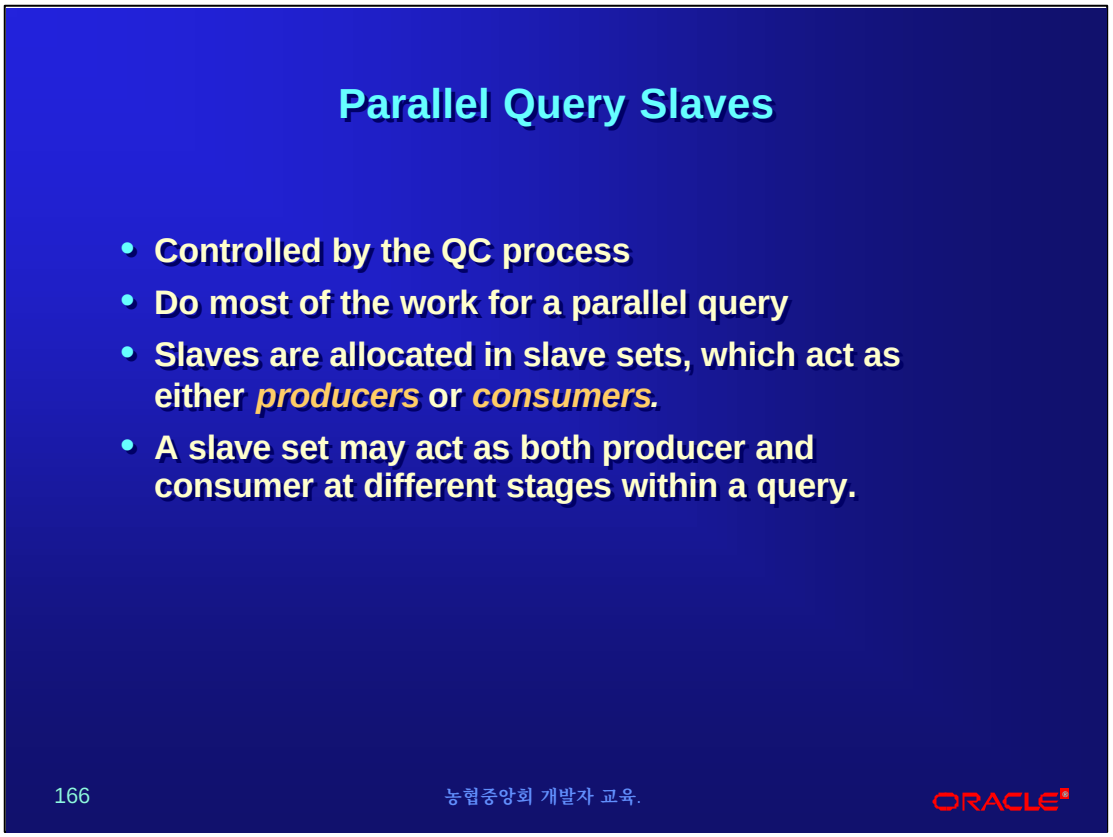


Query Coordinator (QC)

The QC parses the SQL statement; both serial and parallel plans are prepared. It then attempts to get the number of parallel slaves it wants to run the query. If it cannot get sufficient slaves the query may run serially.

Scan Work Ranges
In Oracle7 the work is split for scan slaves by ROWID range. Oracle8*i* may divide the work by ROWID range or partitions.

Sort Work Ranges
The ranges for sort slaves are determined based on the data returned by the scan slaves.

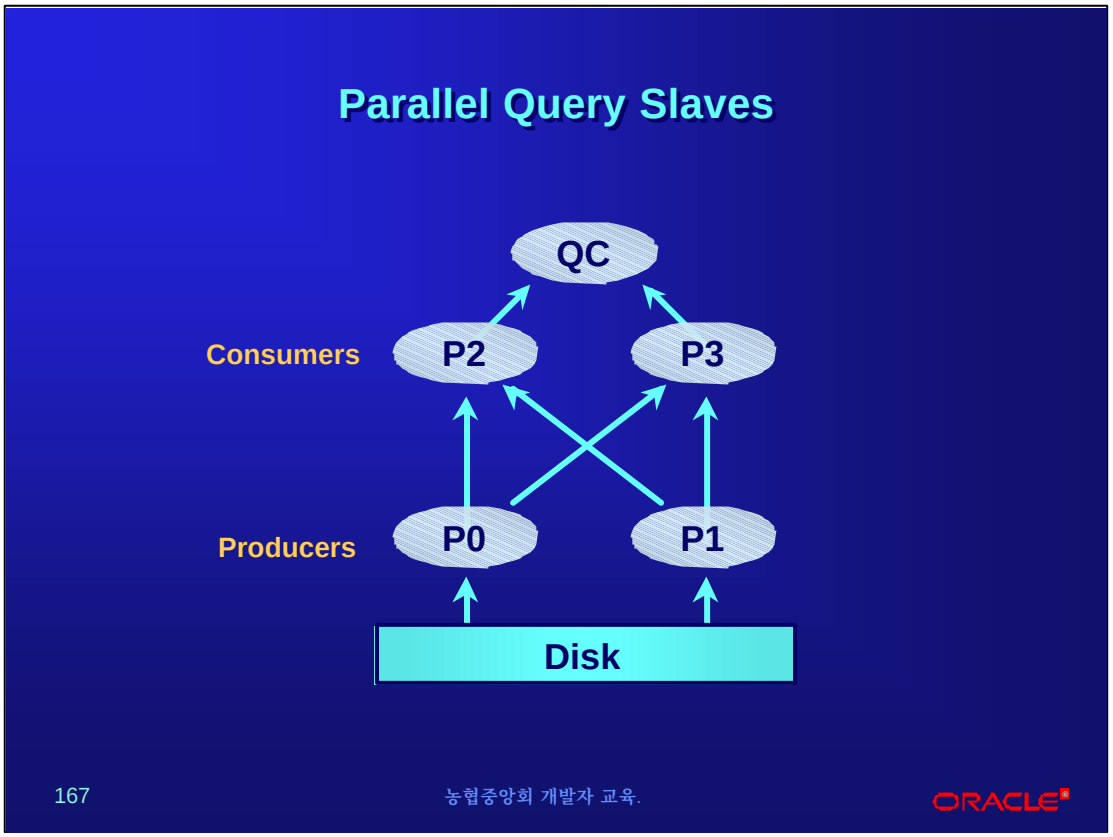


Producers and Consumers

It is possible to have one or two slave sets depending on the complexity of the query. The simplest example is SELECT * FROM EMPLOYEES. If this statement is run in parallel the query will use only one slave set, just to scan the table. This set of slaves will act as *producers*.

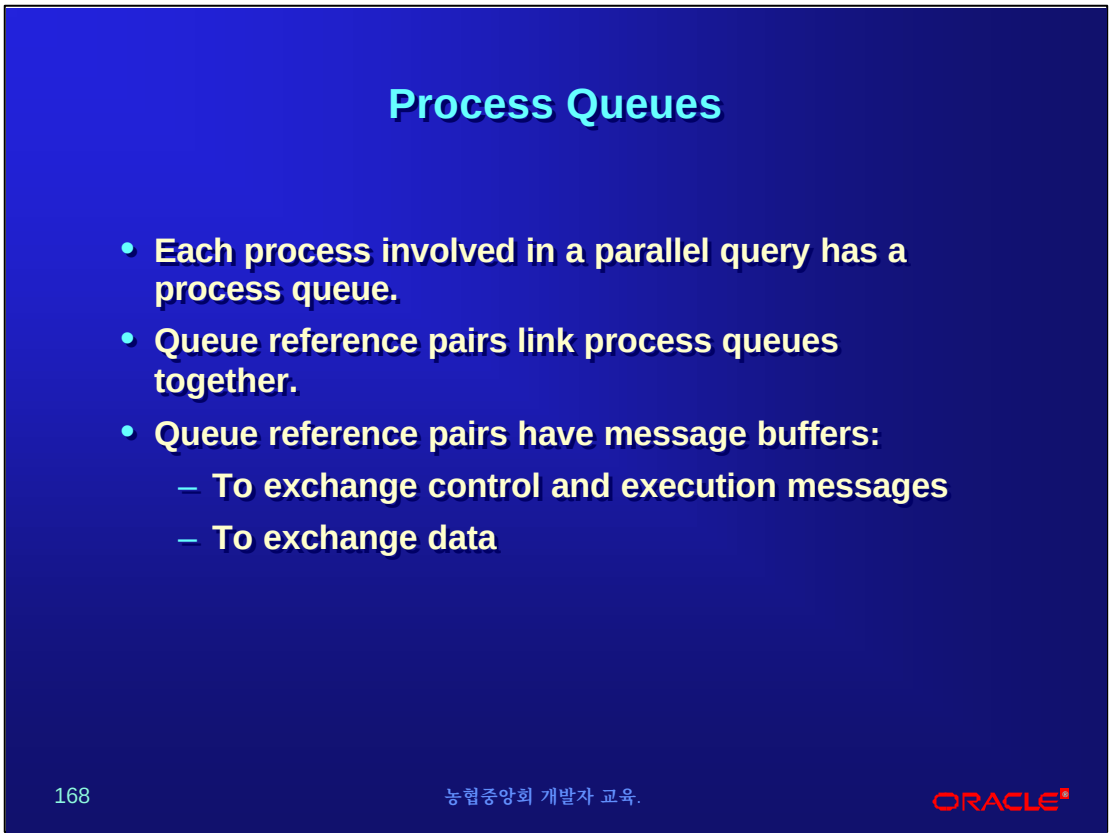
In a more complicated query you have at least two separate slave operations. When acting as producers, slaves are making data available to the next step (through table queues). When acting as *consumers*, slaves are taking data from a previous table queue and performing operations on it.

Consumers are normally used for sorting operations; if a query has no sort step then it is likely it will have only one slave set.



Parallel Query Slaves

In the above example query slaves P0 and P1 are acting as producers (slave set 1). They produce data which is consumed and processed by query slaves P2 and P3 (slave set 2).



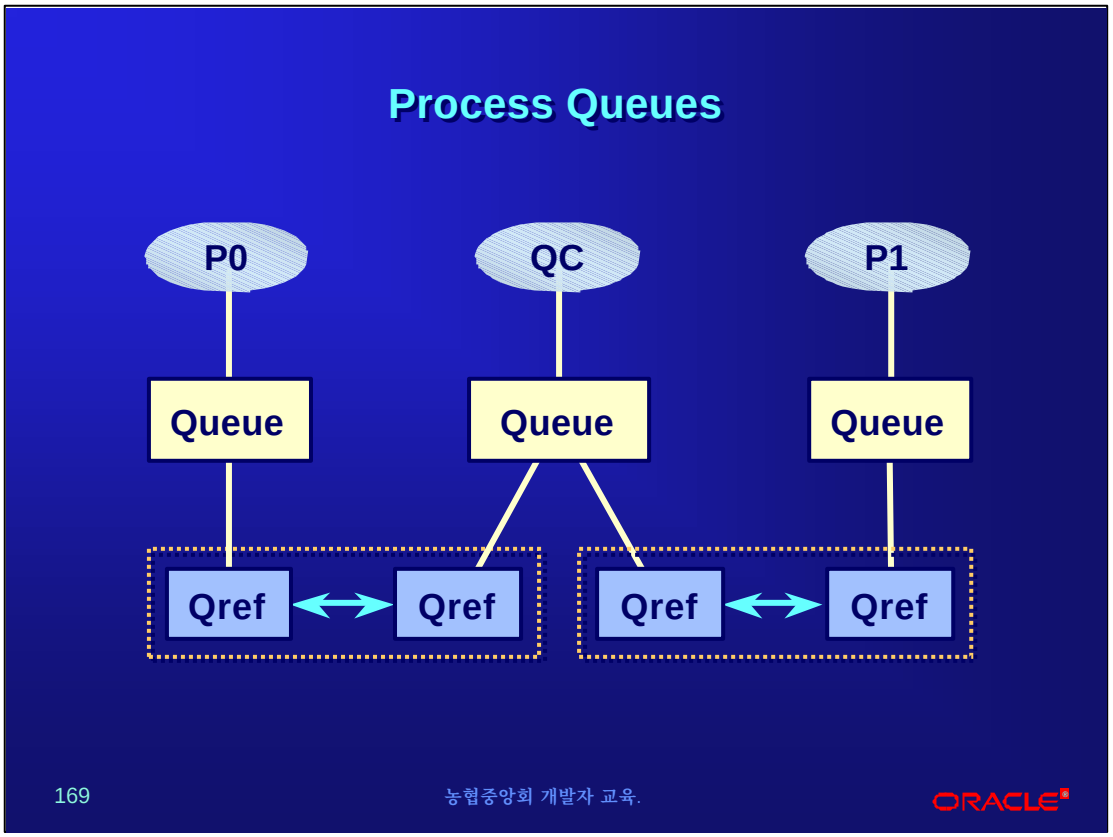
Process Queues

A process queue is the mechanism by which the QC coordinates with PS processes and the PS processes communicate with each other. This communication is handled using queue references.

Queue References
A queue reference is a representation of a link between two process queues. Queue references are always organized in pairs; one for the process at each end of the link.

Message Buffers
Each queue reference pair has three message buffers (four message buffers for internode parallel query) which are used to communicate between the processes. Every parallel operation is given a unique serial number. This is used by all the processes involved as a sanity check on incoming messages, as all messages carry this number.

The “parallel query dequeue wait” event is signalled when a process is waiting to read from one of its queues.



Process Queues (continued)

In the above example the QC is controlling two PS processes. This would be the situation where a query had a single slave set consisting of two slaves.

For the more complicated example of two slave sets (P0 and P1 in set one, P2 and P3 in set two) then there would be queue references as follows:

```
QC <--> P0
QC <--> P1
QC <--> P2
QC <--> P3
```

QC controls slaves and reads end results

```
P0 <--> P2
P0 <--> P3
P1 <--> P2
P1 <--> P3
```

Slave set 1 sends data to slave set 2

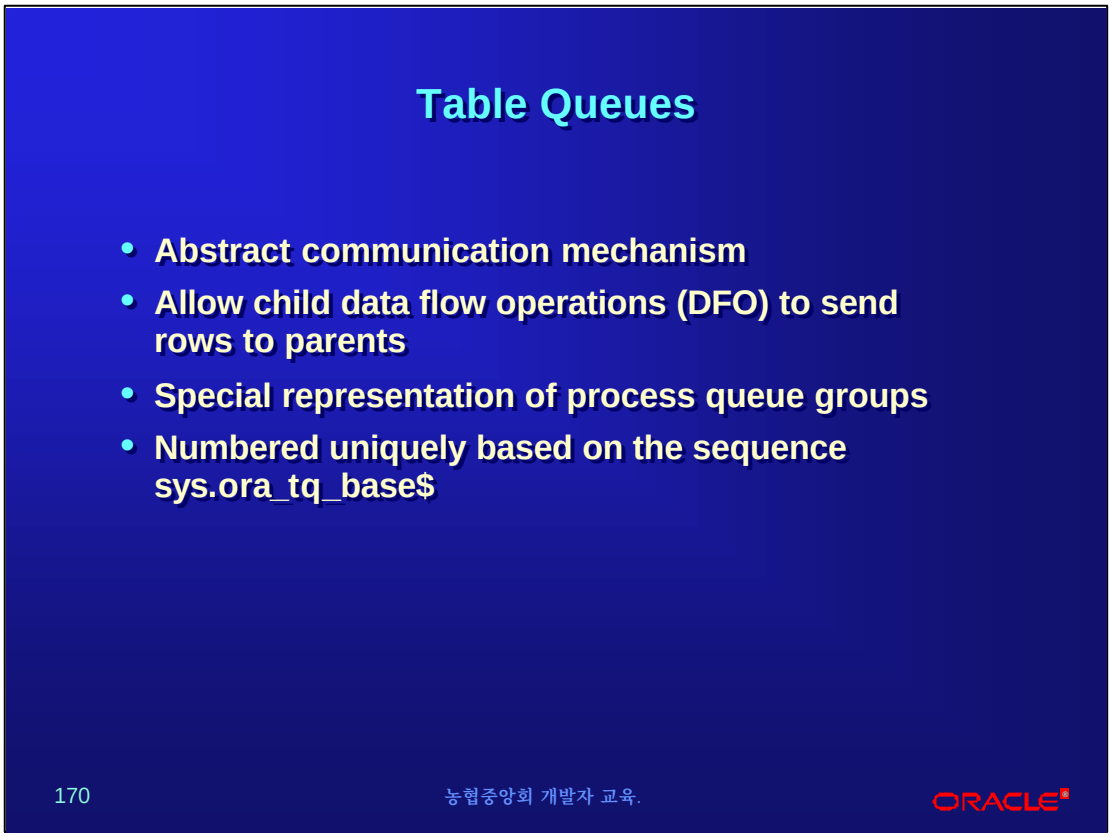


Table Queues (TQ)

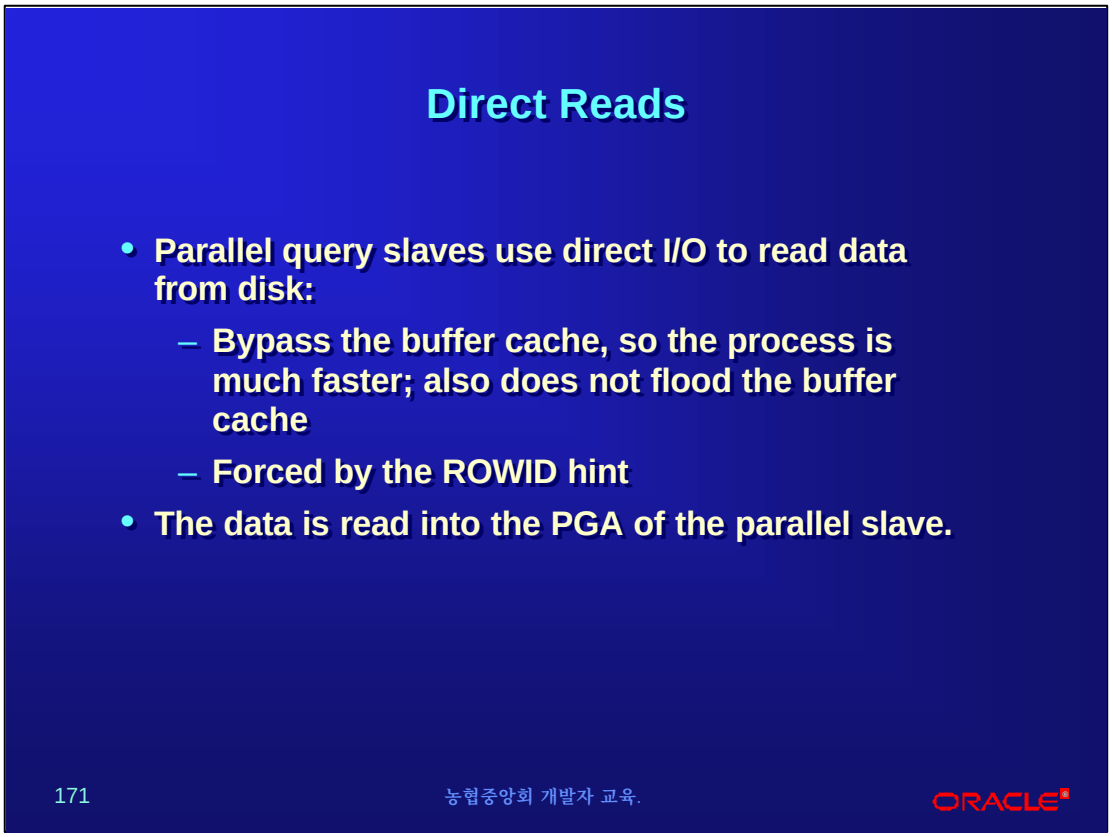
Using the example from the previous slide of two slave sets (P0 and P1 in set one, P2 and P3 in set two) then the table queues would be as follows:

```
P0 <--> P2    P0 produces data for slave set 2
P0 <--> P3

P1 <--> P2    P1 produces data for slave set 2
P1 <--> P3

P2 <--> QC    The QC reads data from slave set 2
P3 <--> QC
```

- P0 and P1 (slave set one) act as producers for slave set two
- P2 and P3 (slave set two) act as consumers from slave set one
- P2 and P3 (slave set two) act as producers for the QC
- QC acts as a consumer from slave set two

This is a trace file fragment showing the use of the sequence ora_tq_base$:

```
PARSING IN CURSOR #10 len=37 dep=1 uid=0 oct=3 lid=0
tim=263039 hv=1908503079 ad='4a6a2f0'
SELECT ORA_TQ_BASE$.NEXTVAL FROM DUAL
END OF STMT
PARSE #10:c=57,e=58,p=0,cr=6,cu=0,mis=1,r=0,dep=1,og=4,tim=263048
EXEC  #10:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=1,og=4,tim=263051
```

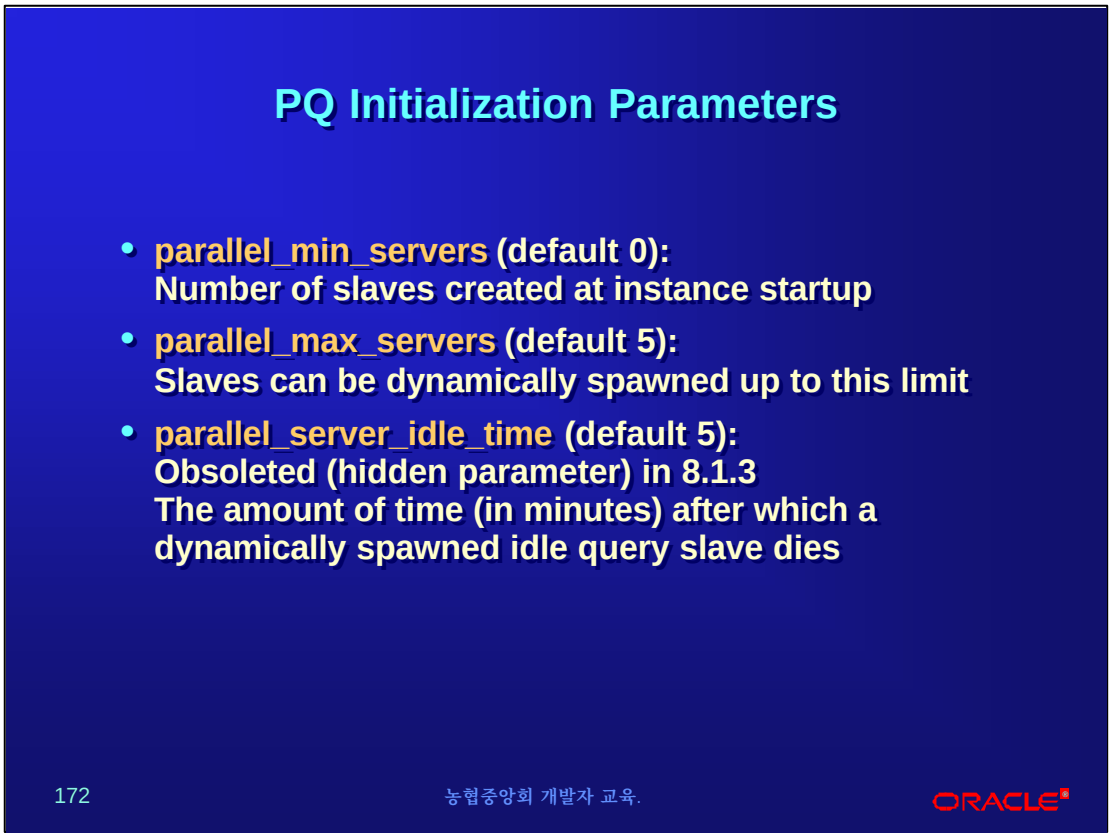


Direct Reads

If direct reads are *not* used (for example, because *compatible* is not set) then the data will be read from the buffer cache in the SGA as normal.

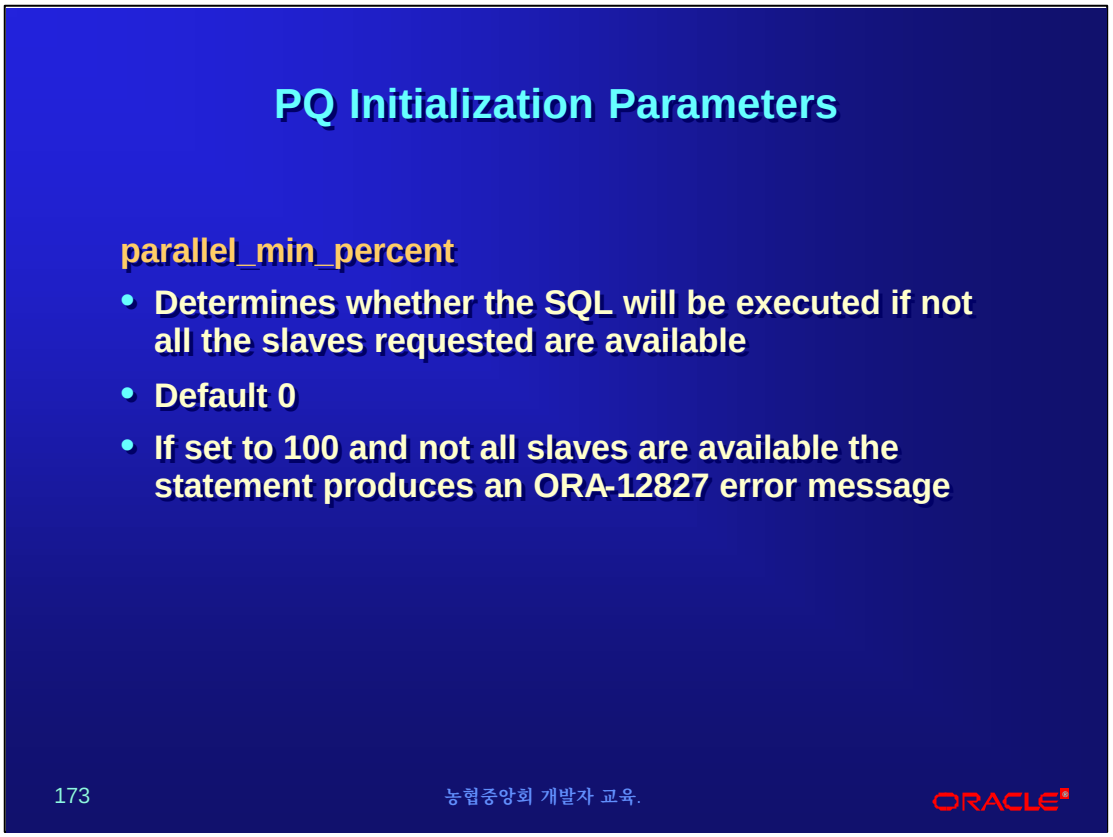
Effects of ROWID Hint

The ROWID hint forces a checkpoint of the object being read to ensure the latest versions of blocks are used.



PQ Initialization Parameters

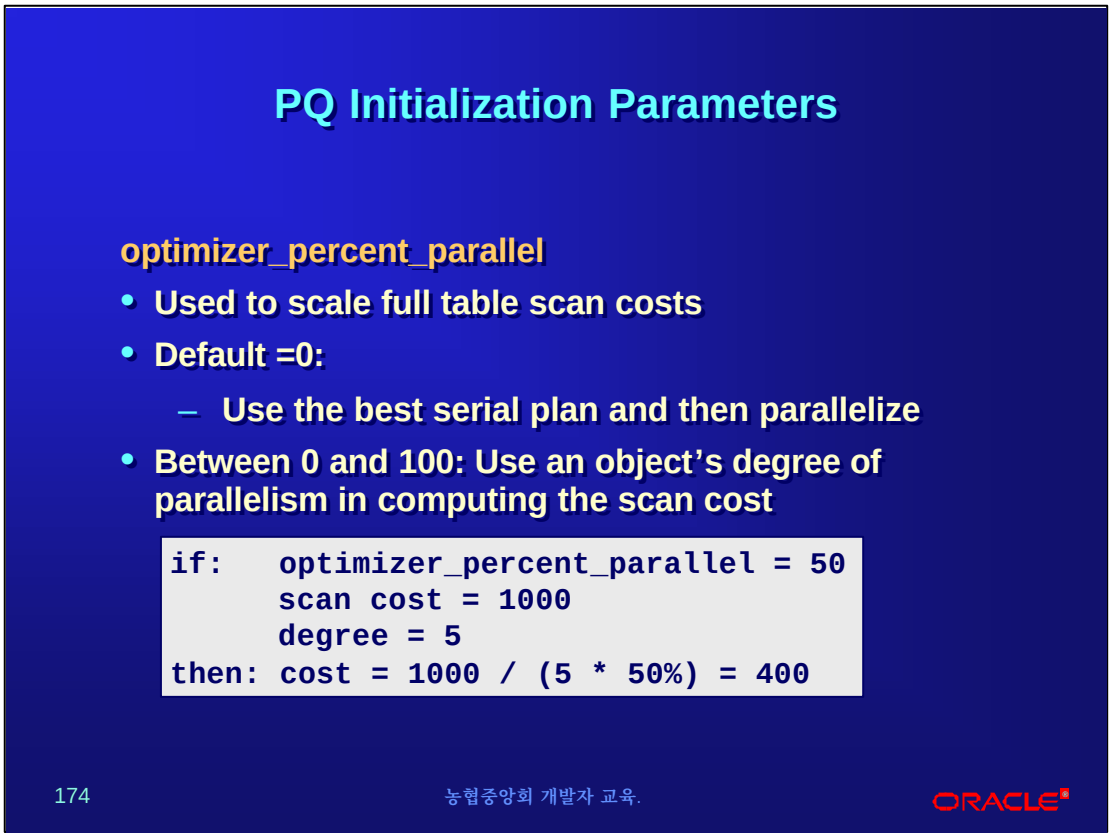
Setting *parallel_min_servers = parallel_max_servers* avoids some of the problems associated with process startup. This is useful for debugging.



PQ Initialization Parameters (continued)

Setting *parallel_min_percent* = 100 is useful to show whether queries are being executed in serial because there is a lack of slave resource, or because they have a nonparallel plan.

Note: *parallel_min_percent* can be set at the session level. In Oracle8.0 there was a bug which prevented this, but it is fixed in 8.0.6.



PQ Initialization Parameters (continued)

Some bugs have been reported with the *optimizer_percent_parallel* parameter in Oracle7.3. The main symptoms are parsing taking an excessive amount of time (in some cases several hours), and excessive memory usage.

Note: *optimizer_percent_parallel* can be set at the session level.

Example: optimizer_percent_parallel

Suppose the registrations table has parallel degree 20 and you have an index on CLASS_ID.

```
SQL> select count(stud_id)
  2  from   registrations
  3  where  class_id > 100000;
```

First the plan with *optimizer_percent_parallel* = 0:

```
Execution Plan
----------------------------------------------------------
0 SELECT STATEMENT Optimizer=CHOOSE (Cost=15)
1   SORT (AGGREGATE)
2     INDEX (RANGE SCAN) OF 'I_REG_CLASS_ID' (NON-UNIQUE)
```

Now the plan with *optimizer_percent_parallel* = 100:

```
Execution Plan
----------------------------------------------------

0 SELECT STATEMENT Optimizer=CHOOSE (Cost=10)
1   SORT (AGGREGATE)
2     TABLE ACCESS* (FULL) OF 'REGISTRATIONS'
```

And the following slave SQL:

```
2 :Q10000 PARALLEL_TO_SERIAL
SELECT /*+ ROWID(A1) PIV_SSF */ COUNT(*)
FROM   "REGISTRATIONS" A1
WHERE  ROWID BETWEEN :B1 AND :B2
AND    A1."CLASS_ID">100000
```

Extracts from a 10053 event trace show the scan costs being altered.

OPTIMIZER_PERCENT_PARALLEL = 0

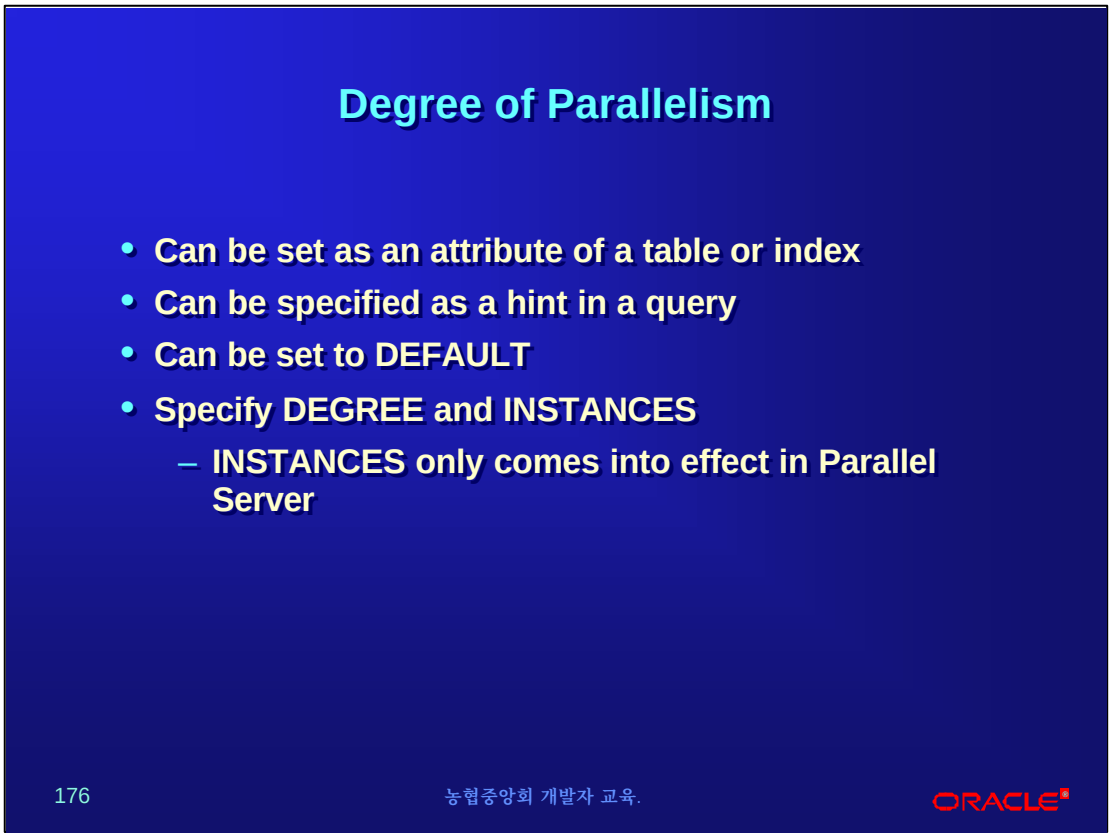
```
************************************************
SINGLE TABLE ACCESS PATH
Column:  CLASS_ID  Col#: 1  Table: REGISTRATIONS
    NDV: 3120  NULLS:0  DENS:3.2051e-004 LO:50288
HI:155801
  TABLE: REGISTRATIONS  ORIG CDN: 56252  CMPTD CDN: 29750
  Access path: tsc  Resc:  177  Resp:  177
  Access path: index (index-only)
      INDEX#: 4431  TABLE: REGISTRATIONS
      CST: 15  IXSEL:  5.2885e-001  TBSEL:  5.2885e-001
```

BEST_CST: 15.00 PATH: 4 Degree: 1

OPTIMIZER_PERCENT_PARALLEL = 100

```
************************************************
SINGLE TABLE ACCESS PATH
Column:  CLASS_ID  Col#: 1   Table: REGISTRATIONS
    NDV: 3120  NULLS:0  DENS:3.2051e-004 LO:50288
HI:155801
  TABLE: REGISTRATIONS  ORIG CDN: 56252  CMPTD CDN: 29750
  Access path: tsc  Resc:  177  Resp:  10
  Access path: index (index-only)
      INDEX#: 4431  TABLE: REGISTRATIONS
      CST: 15  IXSEL:  5.2885e-001  TBSEL:  5.2885e-001
  Access path: index (index-only)
```

BEST_CST: 10.00 PATH: 2 Degree: 20



Degree of Parallelism

The number of parallel slaves to user per slave group.

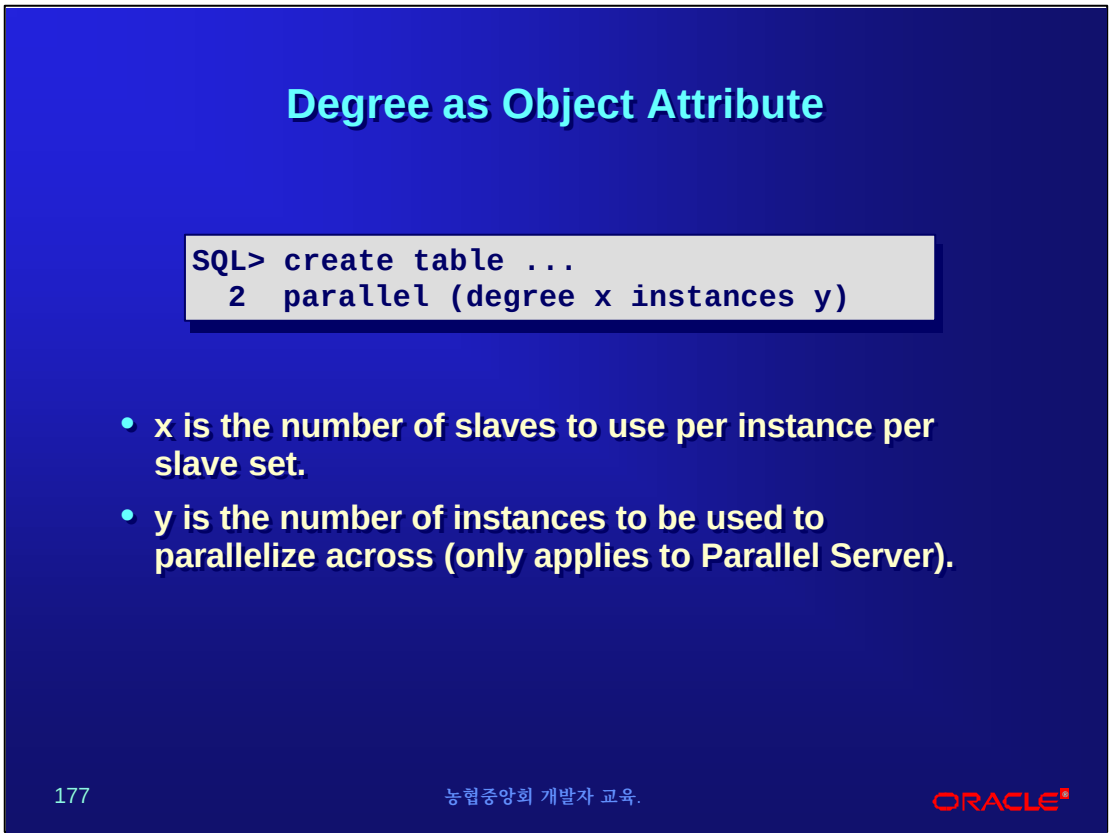
INSTANCES

The number of instances to use in the query; only relevant in a Parallel Server environment.

Setting the Degree of Parallelism to DEFAULT

If you set the degree to DEFAULT the number of slaves to use is calculated based on the number of data files over which the object to be scanned is spread.

Note: Oracle8*i* offers even more flexibility. You can specify the CPU processing power (*parallel_thread_per_cpu*). More details will follow when the *parallel_adaptive_multiuser* and *parallel_automatic_tuning* parameters are discussed.



Degree as Object Attribute

PARALLEL can be set at CREATE or ALTER for tables.

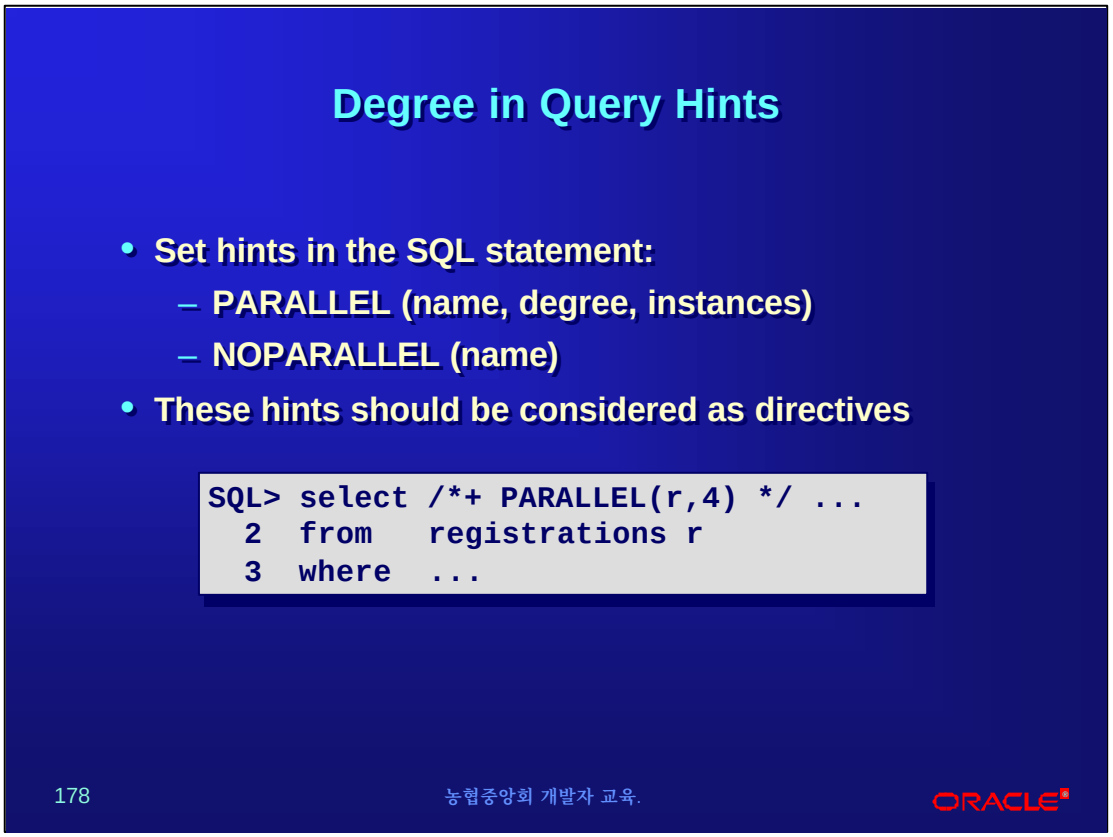
In Oracle7 PARALLEL can be set at CREATE for indexes.

In Oracle8*i* PARALLEL can be set at CREATE or ALTER for indexes.

Note: Setting the degree of parallelism for a table or index will force the use of CBO, even in the complete absence of statistics.

Total Slaves Required

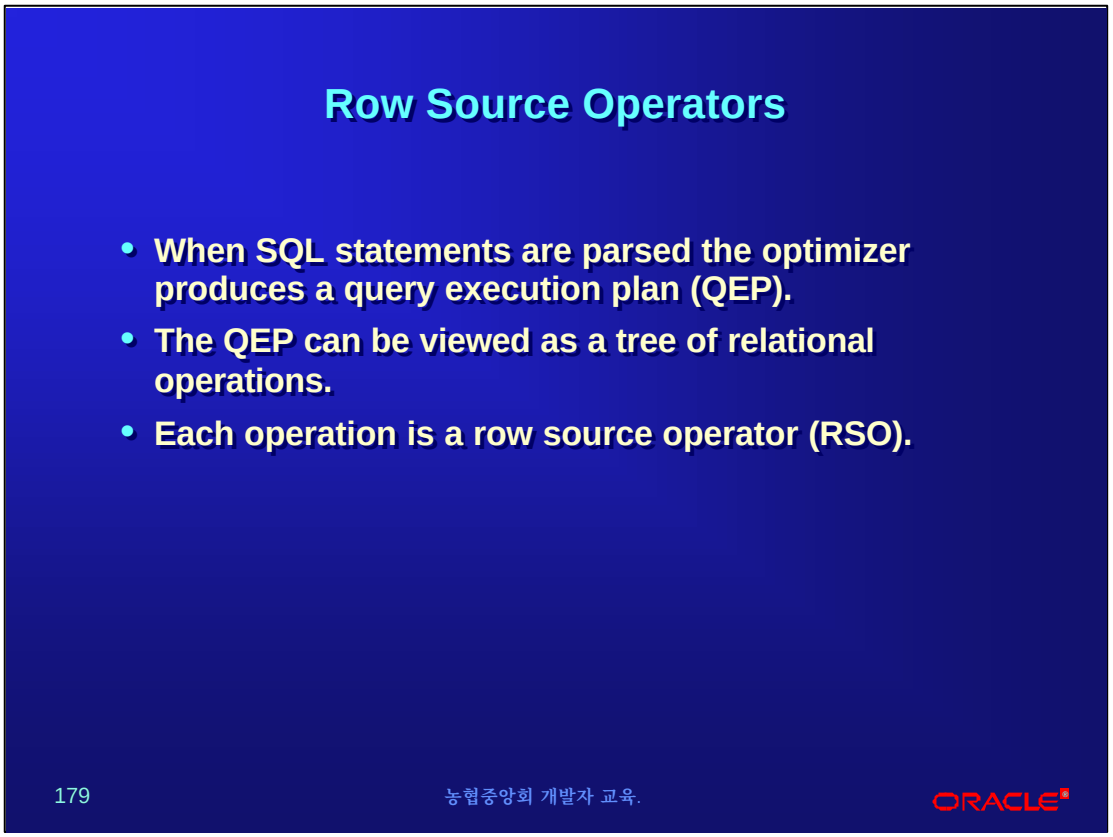
For the above example each slave set gets X slaves per instance. If internode parallel query (IPQ) is in use there will be X * Y slaves per slave set. So if the query being run is sufficiently complex to require both producer and consumer slave sets the actual requirements are 2 * X , or in the IPQ case 2 * X * Y.



Degree in Query Hints

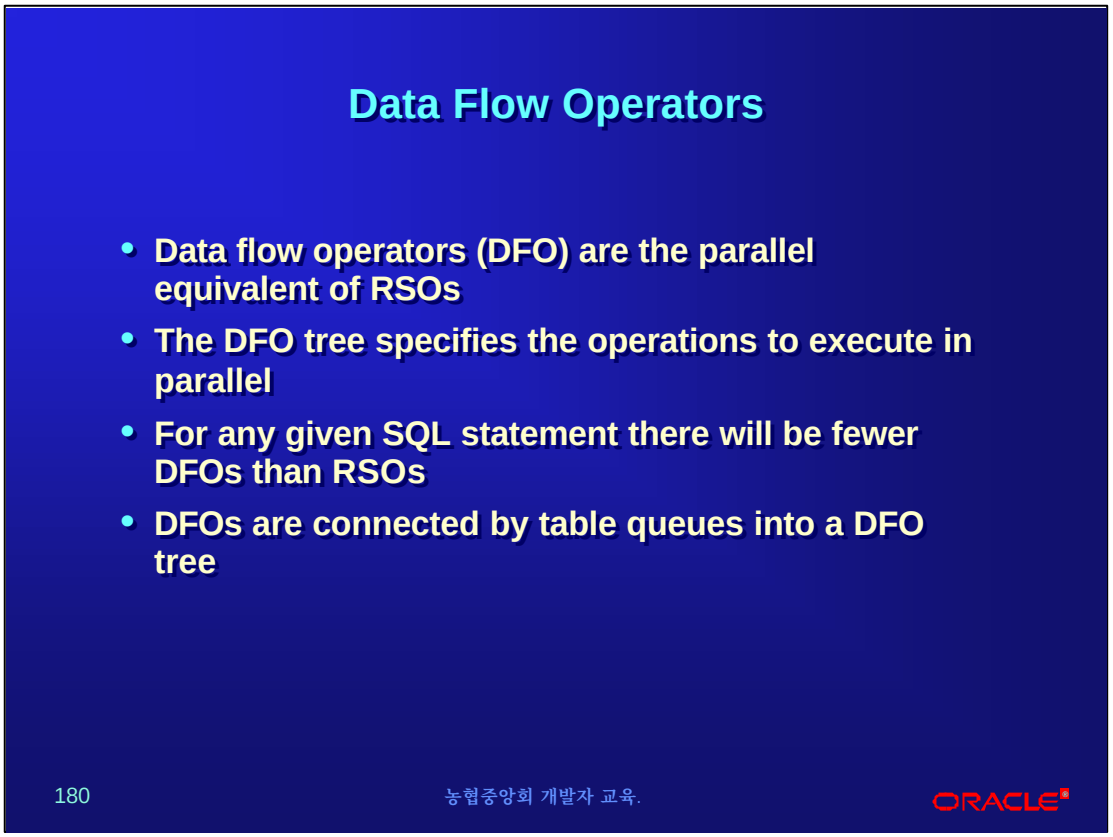
If the PARALLEL or NOPARALLEL hints are specified they will override any parallel attributes set for objects scanned in the query.

If the degree of parallelism requested is not possible then the query may run serially.



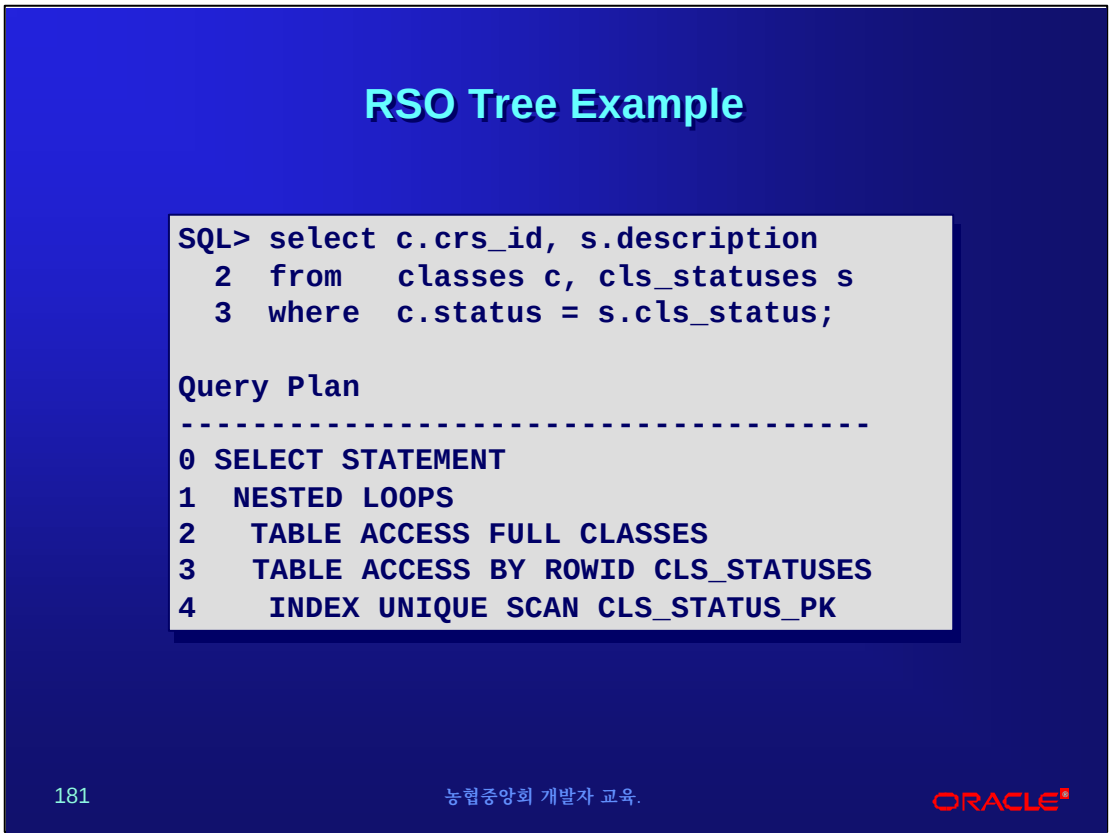
Row Source Operators

A query execution plan (QEP) can be viewed as a tree of operations which are executed in a certain order. Each operation produces a set of rows for consumption by the next operation. This set of rows is defined as a *row source* and the operations are row source operators (RSOs).



Data Flow Operators

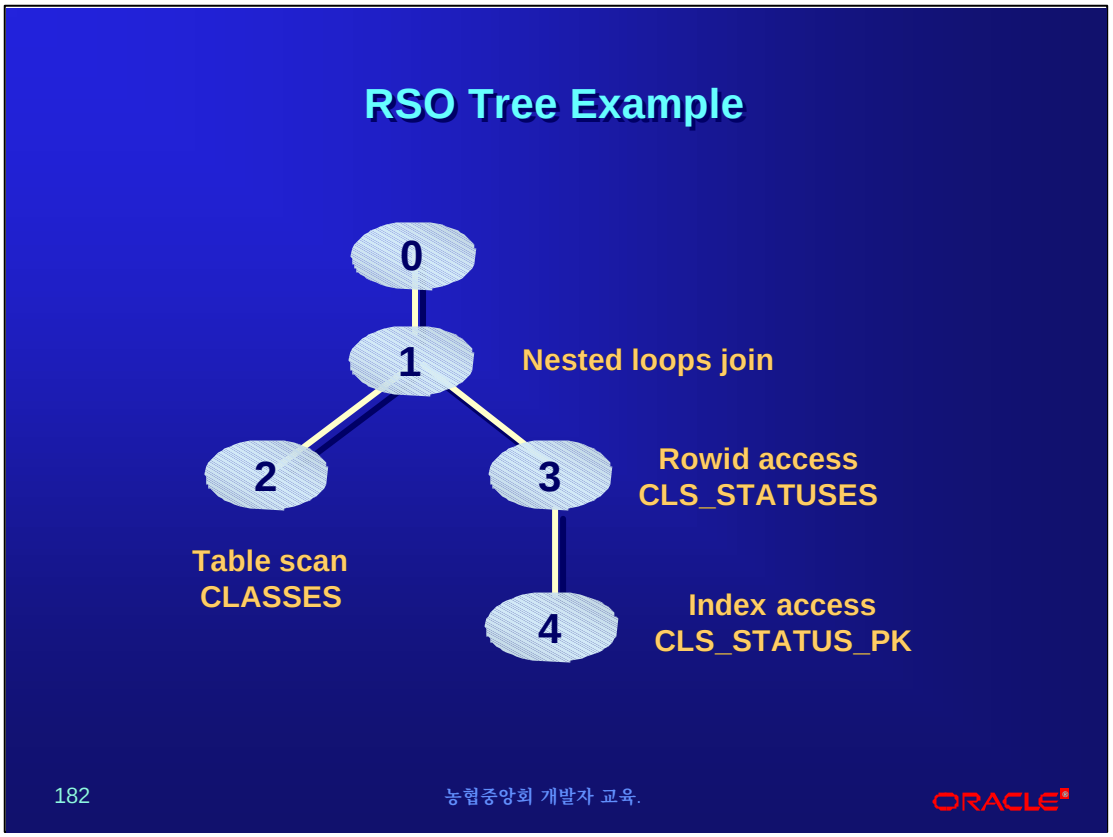
The optimizer builds a RSO tree and a DFO tree (if parallel is being used). Which of these is used depends on the run-time environment. For example, are enough slaves available?



RSO Tree Example

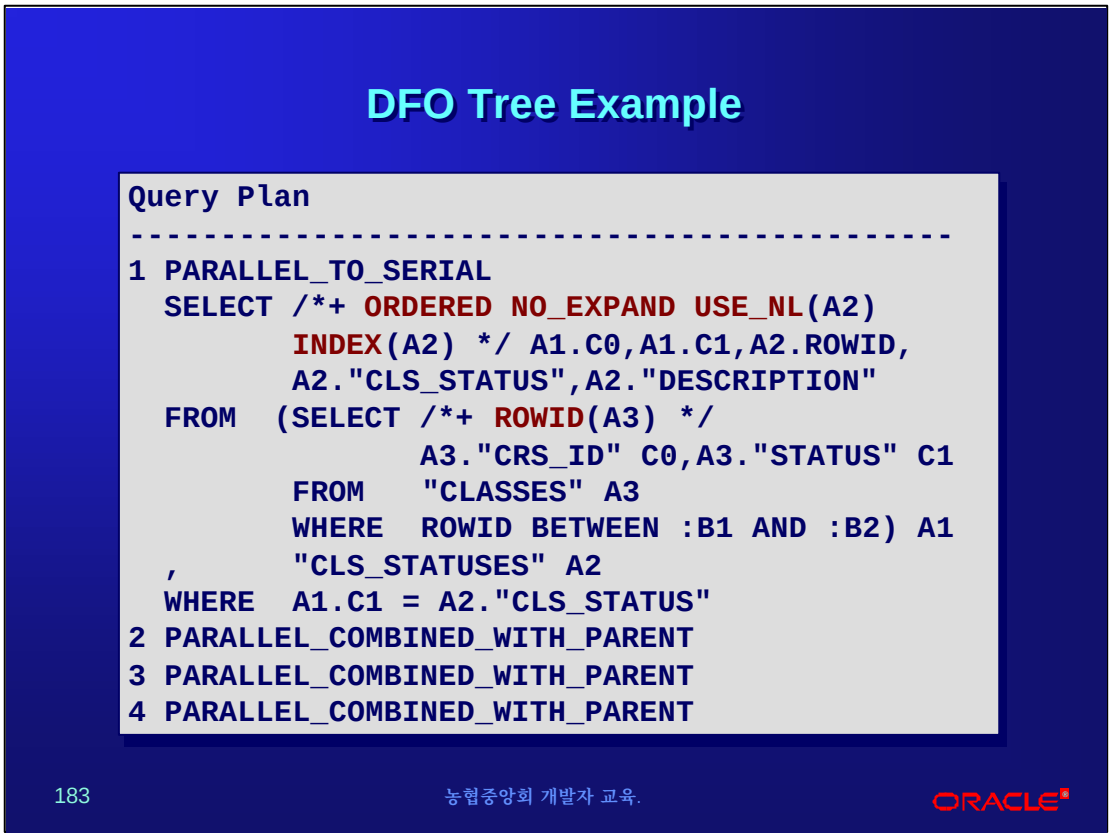
Above is the RSO plan for the query which would be followed were the query to be executed in serial. The corresponding graphical representation is on the next slide.

The corresponding DFO tree for the same statement will follow.



RSO Tree Example (continued)

Each of the row source operators in the RSO tree above corresponds with a record from the query plan on the previous slide.

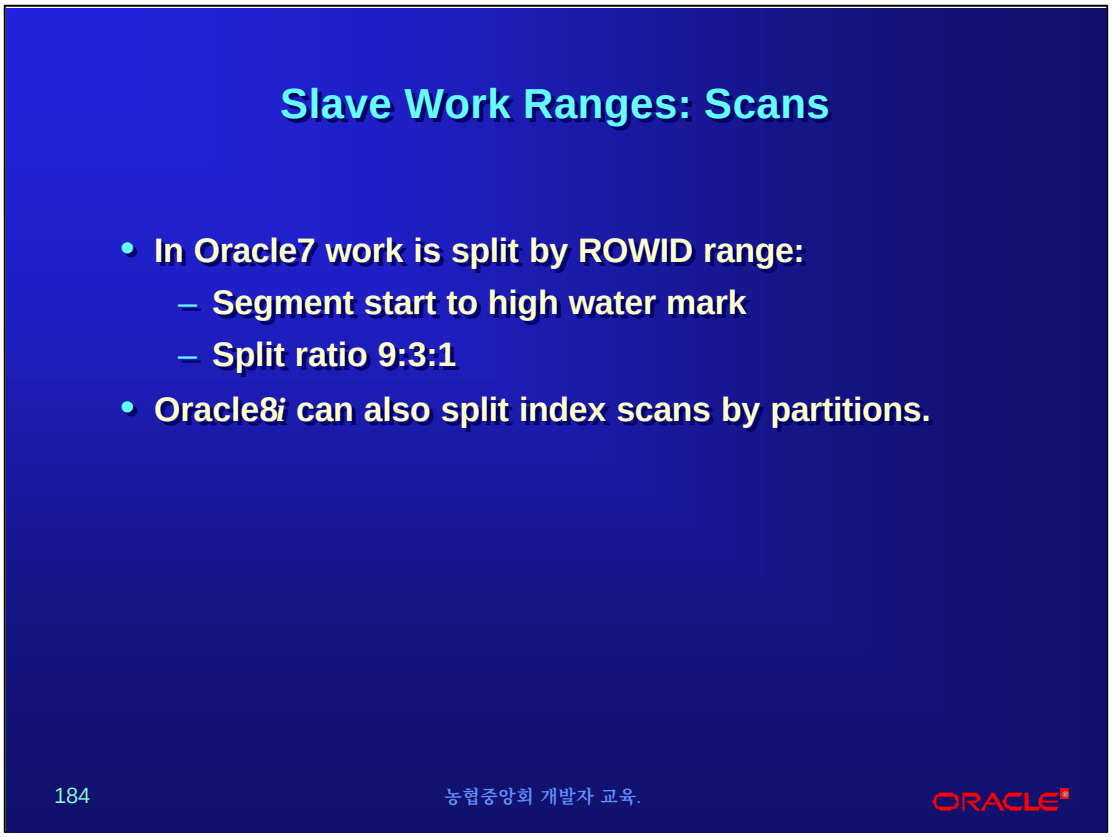


DFO Tree Example

As you can see from the DFO tree above, all steps from the RSO tree on the previous page are contained in a DFO tree containing a single DFO node.

This DFO node has a RSO tree generated if and when it is executed by a parallel slave. In this case the DFO node has the following execution plan:

```
Query Plan
----------------------------------------
0 SELECT STATEMENT
1  FILTER
2   NESTED LOOPS
3    TABLE ACCESS BY ROWID RANGE CLASSES
4    TABLE ACCESS BY ROWID CLS_STATUSES
5     INDEX UNIQUE SCAN CLS_STATUS_PK
```

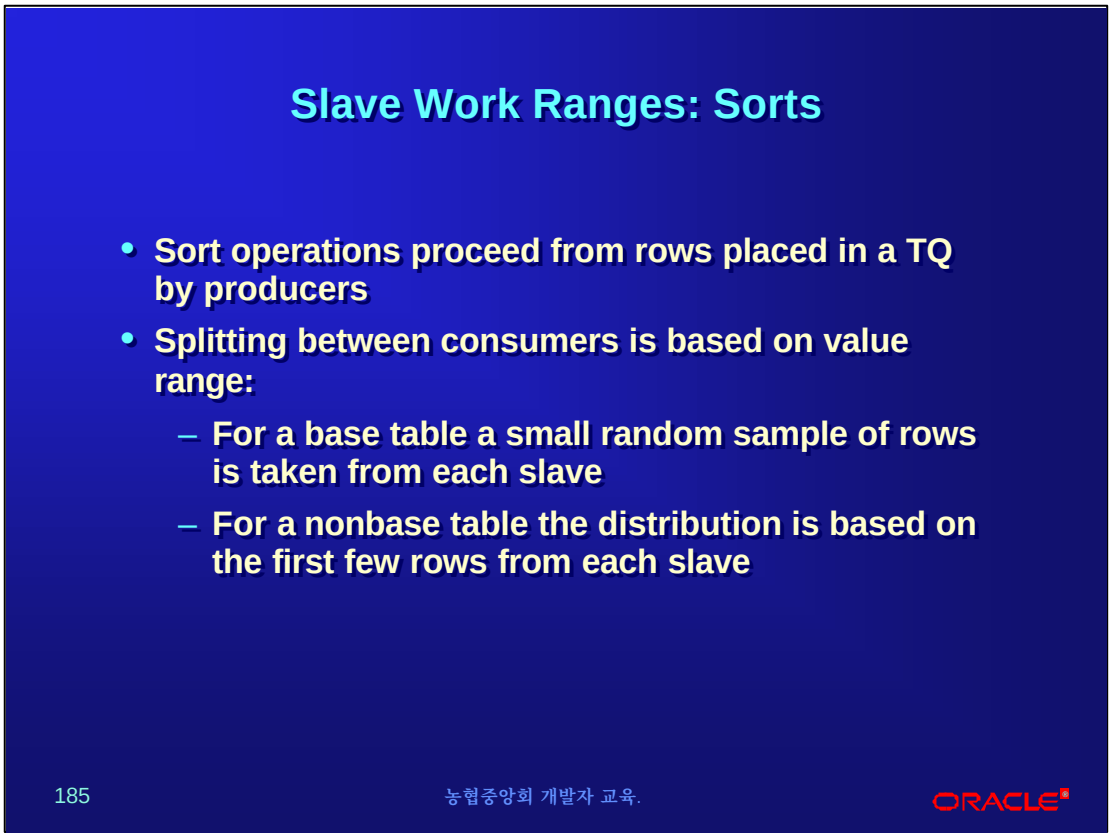


Slave Work Ranges: Scans

Split by ROWID
The number of blocks used in the segment is determined (segment start to highwater mark). This is then divided by the number of slaves in the scan set to give ROWID groups. Each group of ROWIDs is then split by the ratio 9:3:1 giving 3 sets of ROWIDs per slave. Each slave starts with its own 9 portion. When it finishes with that it is given the next biggest portion not yet scanned.

In Oracle8*i* where partitioned objects are being scanned the ROWID ranges do not cross partition boundaries.

Split by Partition
Splitting by partition is done for partition index scans. When splitting by partition in Oracle8*i* there will be at most one slave per index partition scanned.

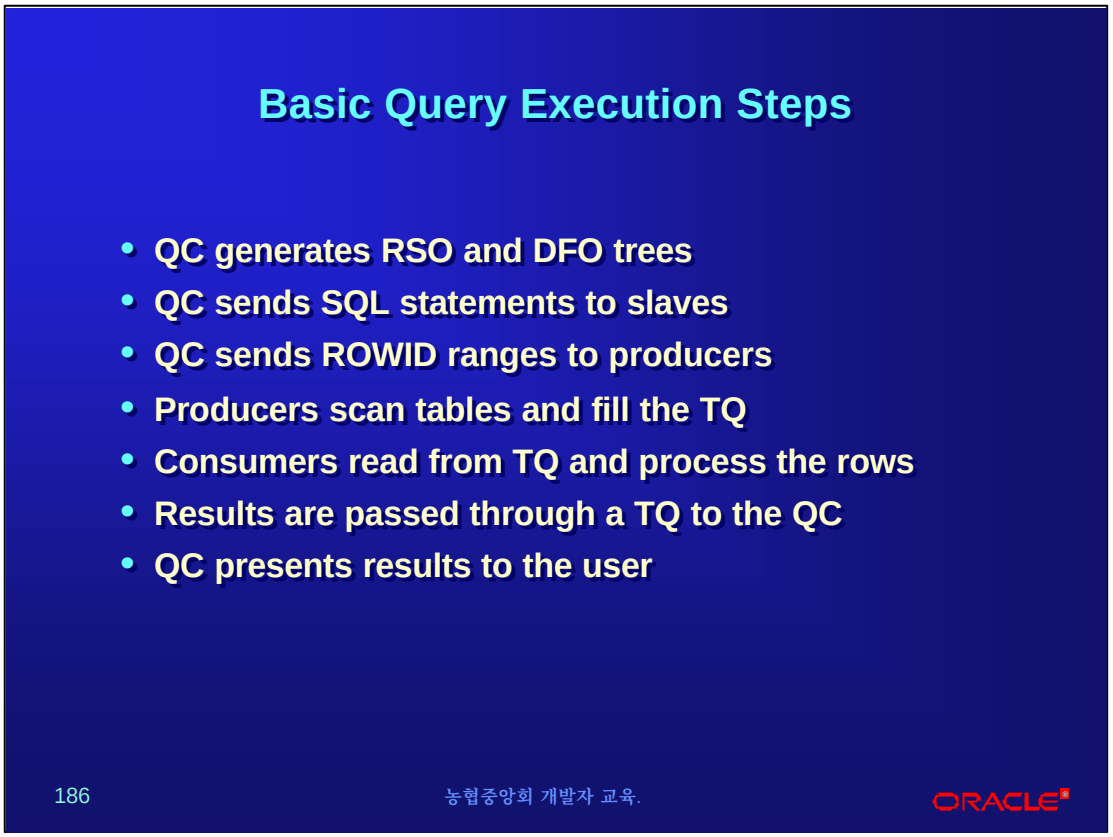


Slave Work Ranges: Sorts

Base Table
A database table being scanned by parallel slave processes

Non-base Table
A step higher up the DFO tree; that is, a table queue

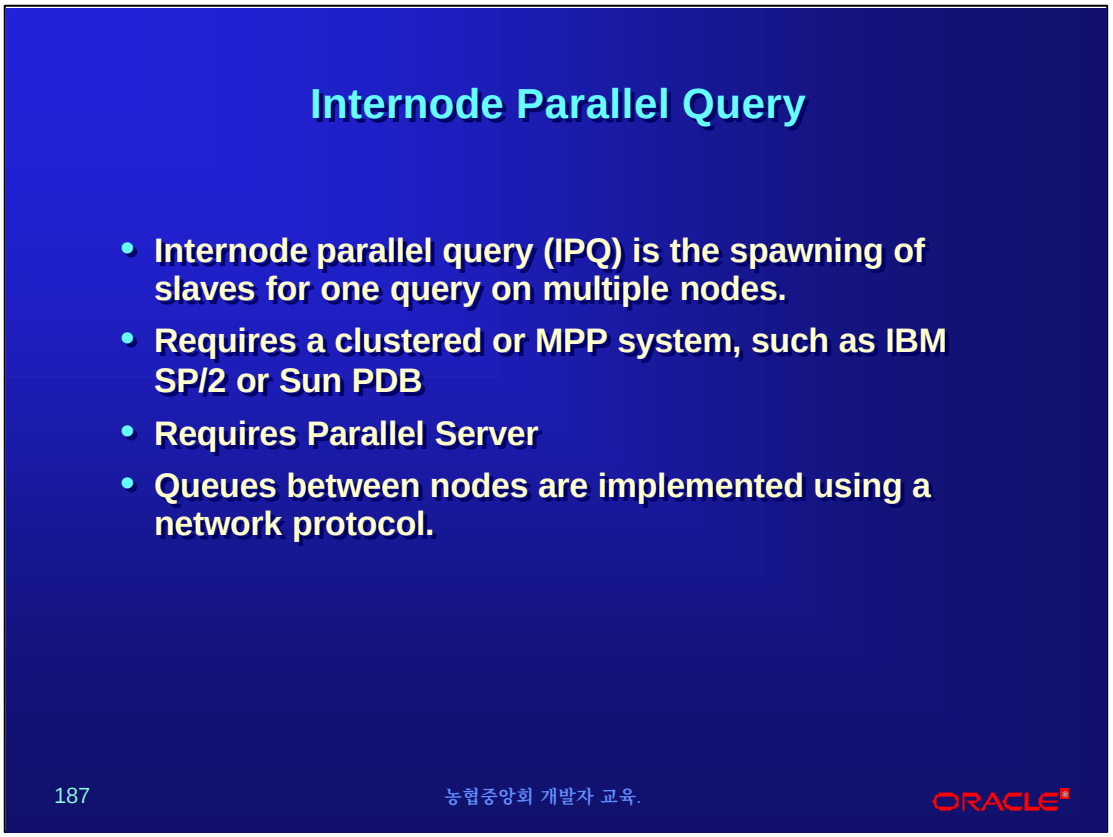
Because the sample may not be representative of the entire table it is possible for the distribution of sort work to be biased. This results in some slaves doing more work than others.



Basic Query Execution Steps

1. The QC parses SQL generating RSO and DFO trees.
2. The QC sends SQL statements to all query slaves who build relevant structures.
3. The QC sends ROWID ranges to the producers.
4. The producers scan the tables and put the resulting rows in the TQ.
5. Consumers read from the TQ and process the rows based on the columns specified in the SQL.
6. The results are passed through a TQ to the QC, which presents them to the user.

Note: Remember that there may not always be consumers.

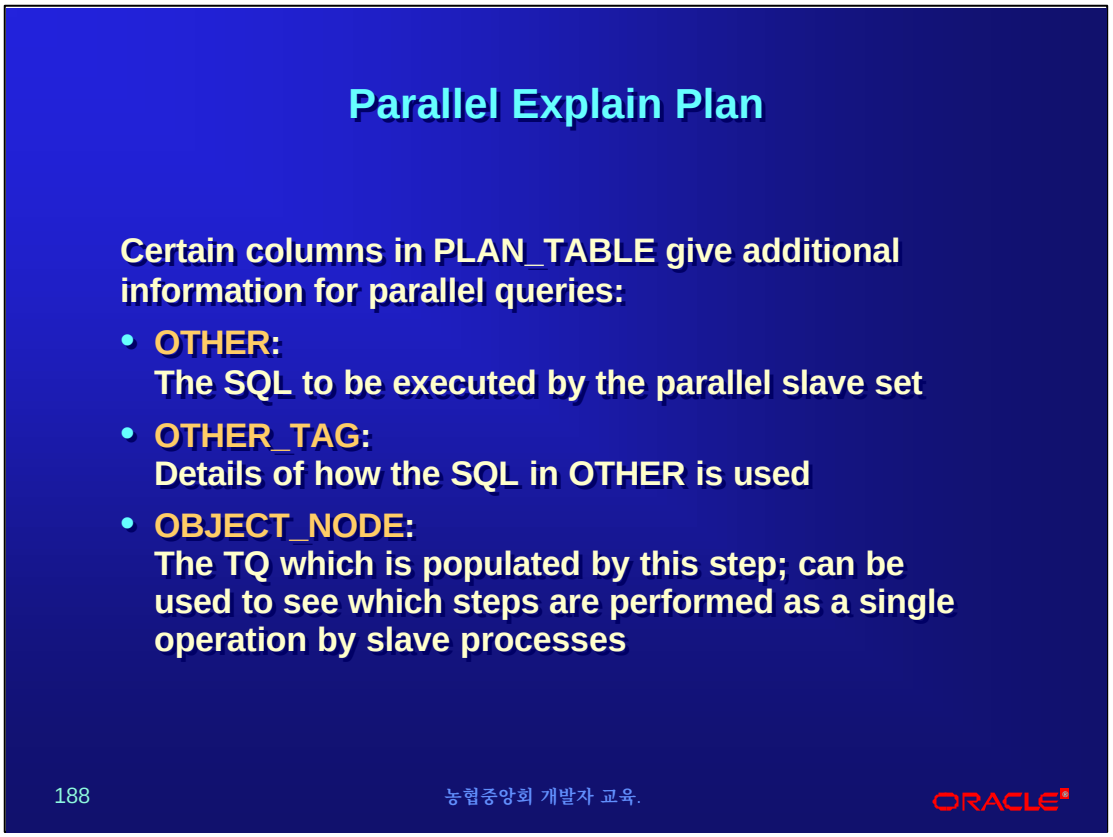


Internode Parallel Query (IPQ)

IPQ needs a low latency/high bandwidth communication channel between the nodes.

The steps in query execution are the same as for single instance PQ.

IPQ can be more difficult to monitor; this is made easier in Oracle8*i* by the global GV$ views implemented for Parallel Server.



Parallel Explain Plans

The following values for OTHER_TAG are used in parallel plans:

PARALLEL_FROM_SERIAL
Serial execution: Input is read from parallel slaves

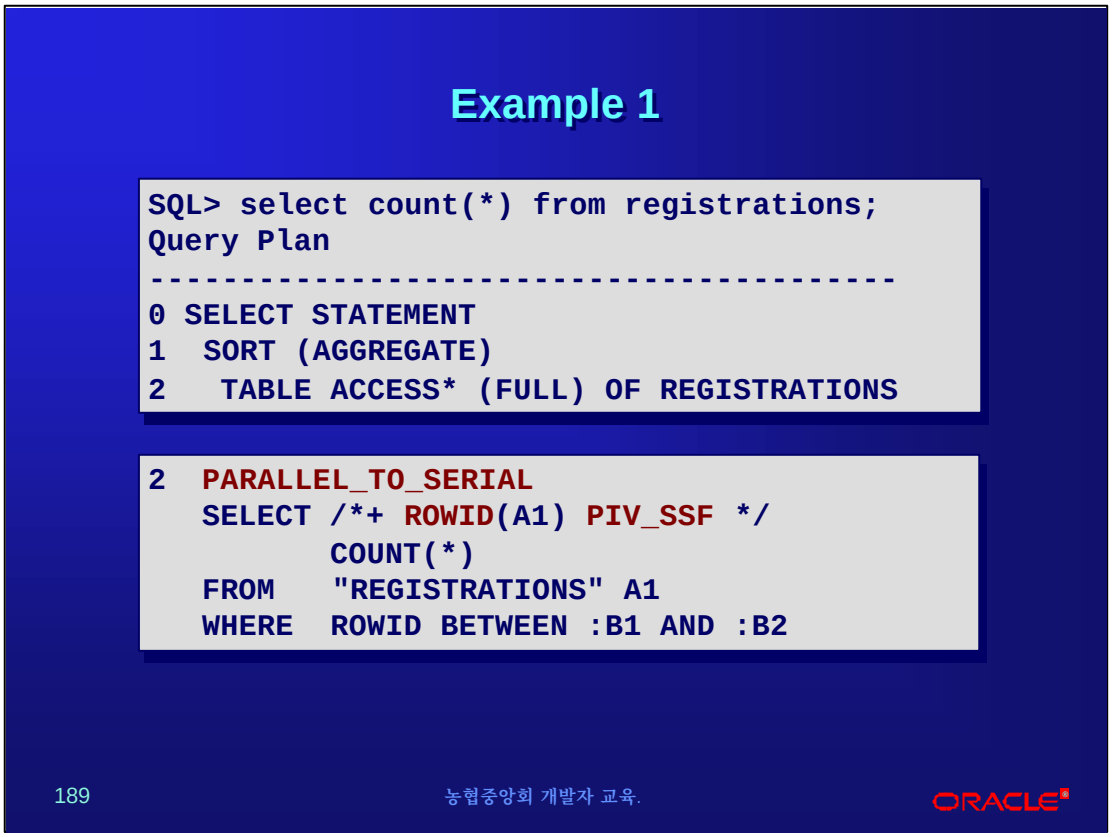
PARALLEL_TO_PARALLEL
Parallel execution: Output is sent to the next set of slaves

PARALLEL_TO_SERIAL
Parallel execution: Output is returned to a serial process

PARALLEL_COMBINED_WITH_PARENT
Parallel execution: Output goes to the next step in the same parallel process (no interprocess communication)

PARALLEL_COMBINED_WITH_CHILD
Parallel execution: Input comes from the prior step in the same parallel process (no interprocess communication)

The serial execution is done by the QC process.



Example 1

Note: The SORT referred to above is not a real sort but the way AGGREGATE is denoted in explain plan.

The DFO information in the second box above can be retrieved from the object_node, other_tag, and other columns of the plan_table; it is also produced by SQL*Plus AUTOTRACE.

If executed in parallel, this query gets a single slave set which acts as producer; the consumer is the QC process. The QC reads the results from each table queue, sums the count(*) results, and then returns them to the user.

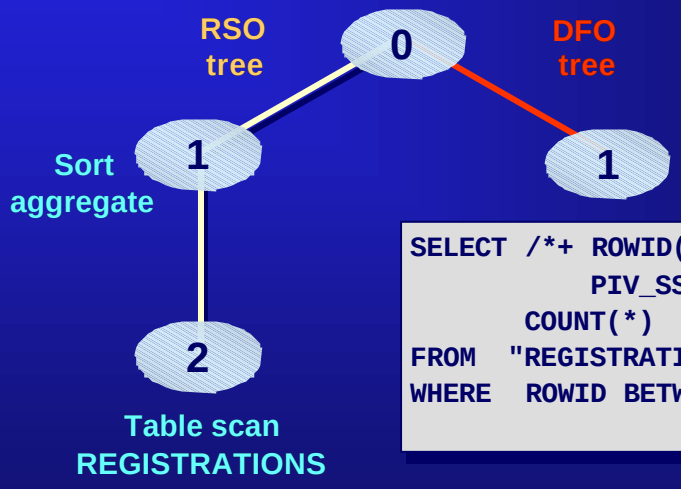
There is no second slave set because there are no sort operations or subsequent steps in the execution plan.

Slave SQL Hints

As the above SQL shows, there are special hints used in the slave SQL statements. These hints are:

- **PIV_SSF** Producer intermediate values—single set function
- **CIV_SSF** Consumer intermediate values—single set function
- **PIV_GB** Producer intermediate values—group by
- **CIV_GB** Consumer intermediate values—group by

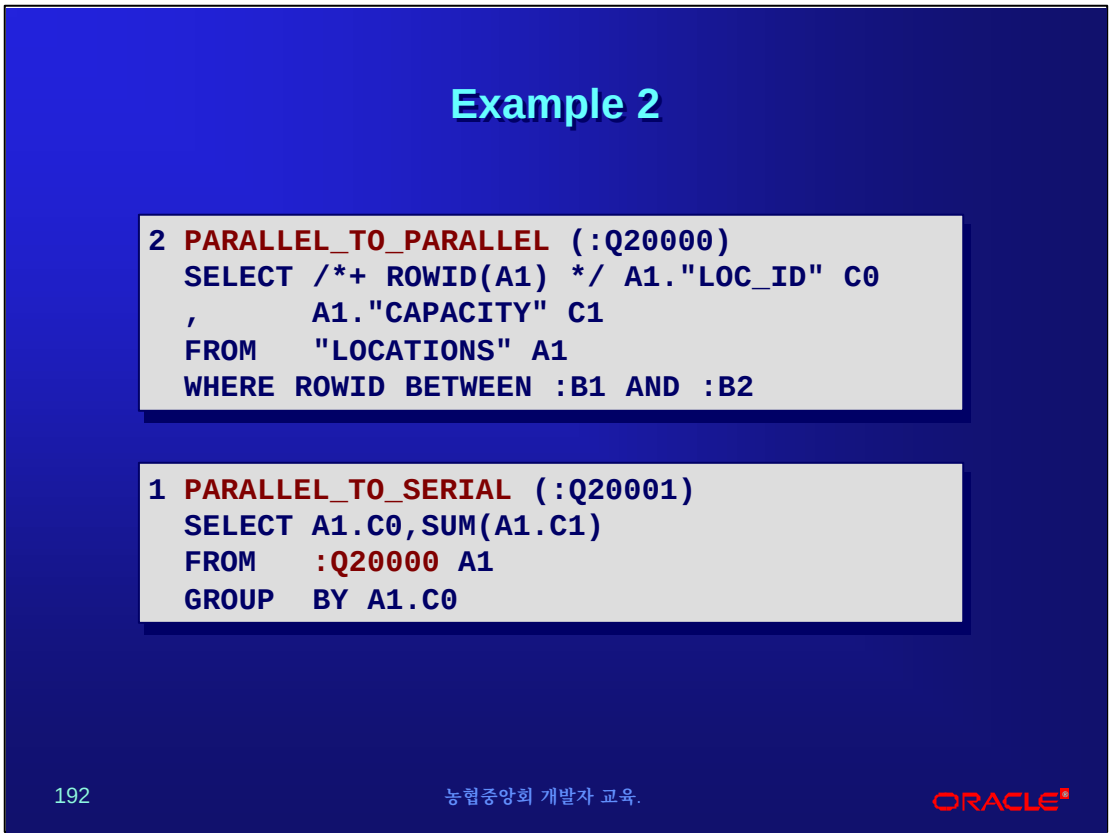
Example 1




```
SELECT /*+ ROWID(A1)
           PIV_SSF   */
       COUNT(*)
FROM  "REGISTRATIONS" A1
WHERE  ROWID BETWEEN :B1
                 AND :B2
```

Example 2

```
SQL> select loc_id, sum(capacity)
  2  from   locations
  3  group  by loc_id;

Query Plan
----------------------------------------------
0 SELECT STATEMENT
1  SORT* (GROUP BY)                    :Q20001
2   TABLE ACCESS* (FULL) OF LOCATIONS  :Q20000
```



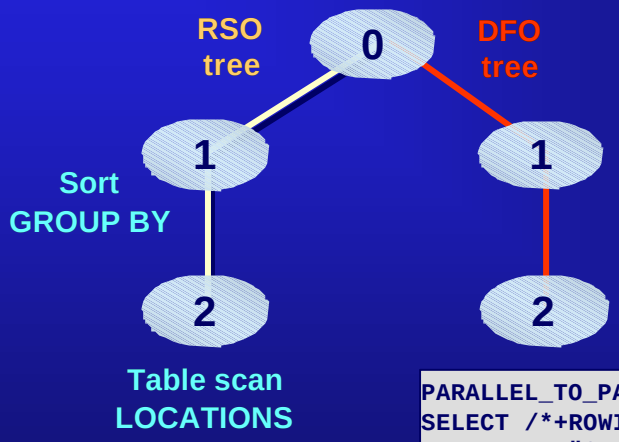
Example 2 (continued)

First the *producer* slave set reads from LOCATIONS and populates the table queue Q20000.

Then the *consumer* slave set reads the table queue, sums the data, and puts the results onto another table queue (Q20001).

Finally the query coordinator reads the results from this second table queue.

Example 2

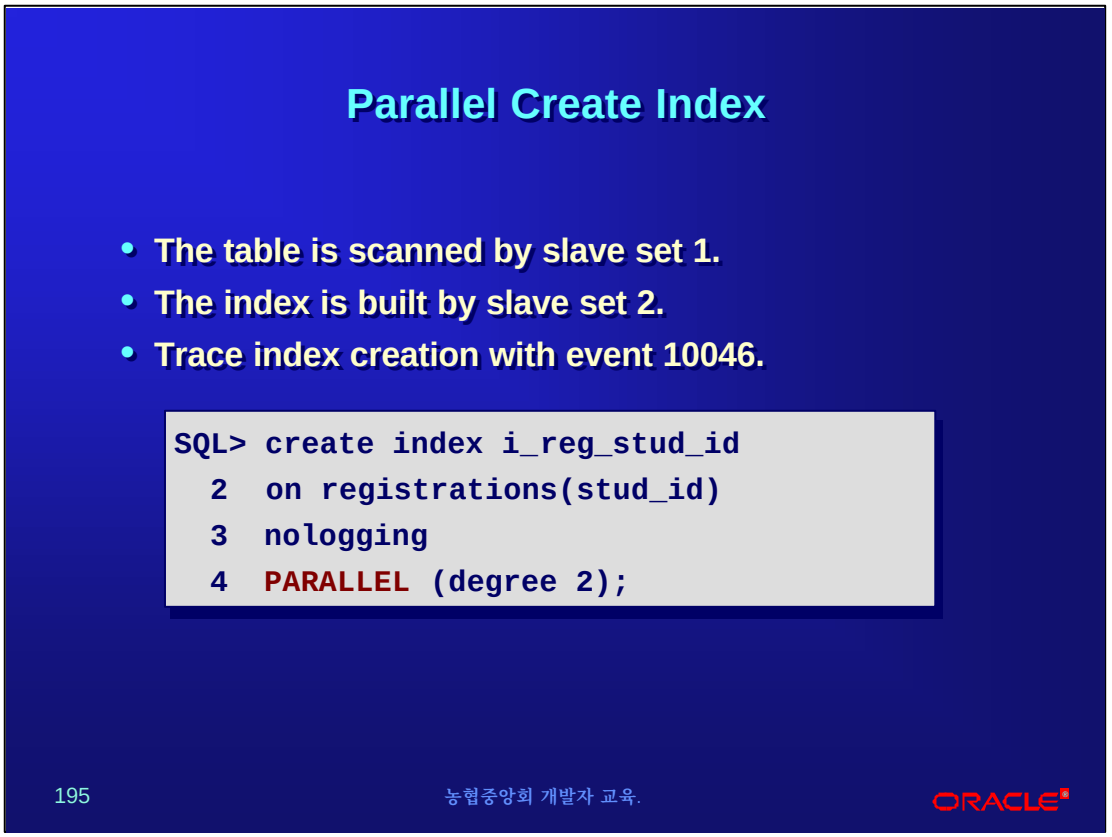



```
PARALLEL_TO_SERIAL
SELECT A1.C0,SUM(A1.C1)
FROM   :Q20000 A1
GROUP  BY A1.C0
```

```
PARALLEL_TO_PARALLEL (:Q20000)
SELECT /*+ROWID(A1)*/ A1."LOC_ID" C0
,      A1."CAPACITY" C1
FROM   "LOCATIONS" A1
WHERE ROWID BETWEEN :B1 AND :B2
```

Parallel DML

- Parallel DML (PDML) is a parallel implementation of UPDATE, INSERT, and DELETE.
- PDML can be used on both partitioned and nonpartitioned tables.
- PDML is available since Oracle8.0



Parallel Create Index

For the above example slave set 1 executes the following slave SQL:

```
SELECT A1."STUD_ID" C0,A1.ROWID C1
FROM   :I."REGISTRATIONS"."REG_PK" A1
WHERE  ROWID BETWEEN :B1 AND :B2
```

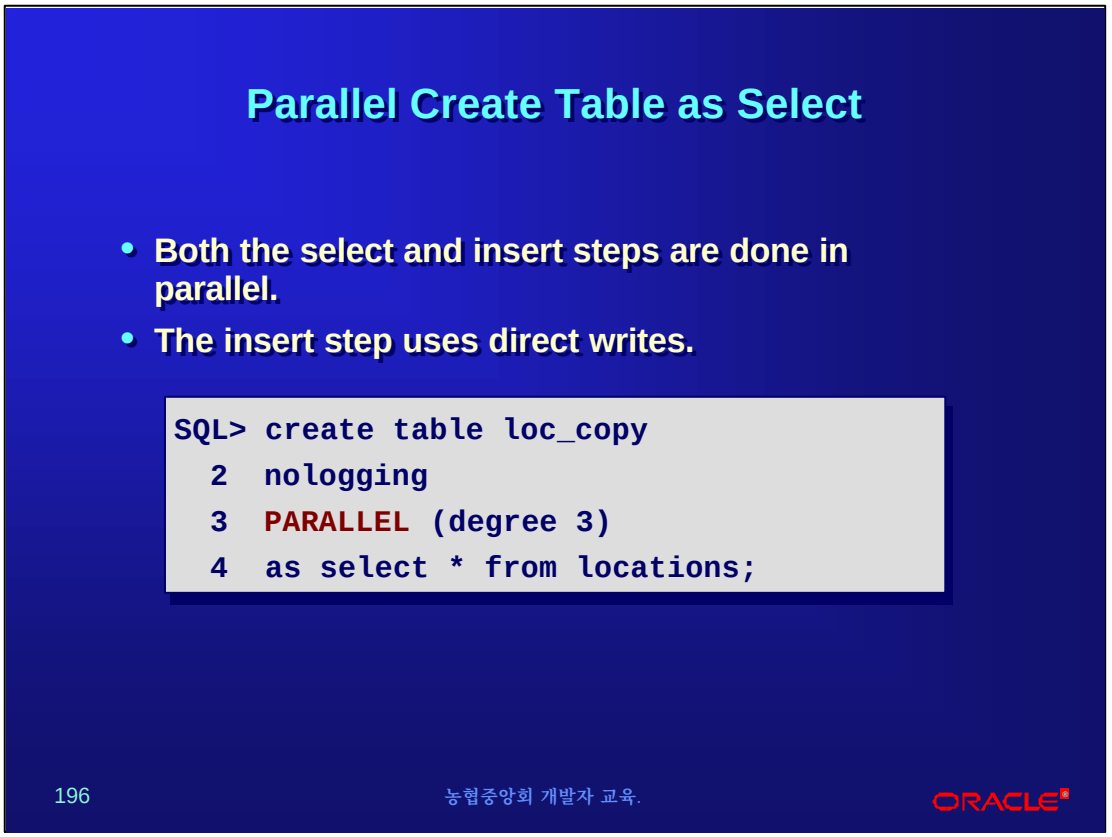
Note that the STUD_ID is part of the primary key, and that the optimizer does not access the table but rather uses the existing index. The data is divided between slaves in the second set based on the values in the column being indexed. Note that the trace file also contains the following statement:

```
SELECT /*+ FORCE_SAMPLE_BLOCK NO_EXPAND*/
A1."STUD_ID" C0, A1.ROWID C1 FROM "REGISTRATIONS"
SAMPLE BLOCK(0.100000) A1
```

Then slave set 2 builds the index as follows:

```
CREATE INDEX "I_REG_STUD_ID" ON :Q147000 (C0,C1)
TABLESPACE_NO 1 PCTFREE 10 INITRANS 2 MAXTRANS 255
STORAGE (OBJNO 13332 DATAOBJNO 13332) NOLOGGING
NOPARALLEL
```

Unfortunately EXPLAIN PLAN cannot be used with CREATE INDEX statements. These statements are extracted from a trace obtained with event 10046.



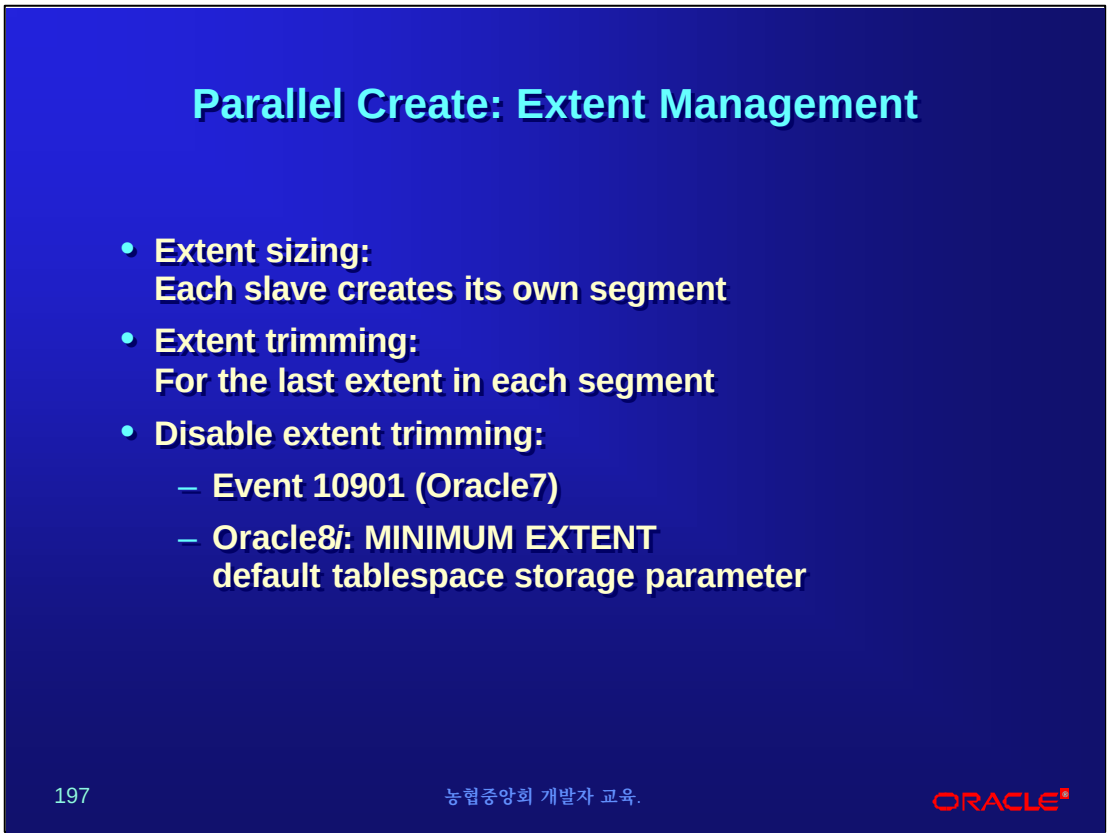
Example Execution Plan

The execution plan for the PCTAS command above looks like:

```
Query Plan
---------------------------------
0 CREATE TABLE STATEMENT
1   CREATE AS SELECT
2     TABLE ACCESS FULL LOCATIONS
```

This extract from the trace file shows the slave SQL:

```
CREATE TABLE "SST".:Q148000
("LOC_ID","NAME","CITY","STATE","CAPACITY")
NOLOGGING
AS
SELECT C0,C1,C2,C3,C4
FROM  (SELECT /*+ NO_EXPAND ROWID(A1) */ A1."LOC_ID"
C0,
              A1."NAME" C1,A1."CITY" C2, A1."STATE"
C3,
              A1."CAPACITY" C4
       FROM   "LOCATIONS" A1
       WHERE  ROWID BETWEEN :B1 AND :B2)
```



Extent Sizing

Each slave creates its own temporary segment; these segments are then merged into a single segment as the last step of the operation. This means that attention must be paid to the INITIAL and NEXT storage values.

In Oracle7 each slave will have an extent of size INITIAL and may have one or more of size NEXT.

In Oracle8*i* the value of INITIAL is ignored and each slave only has extents of size NEXT.

Extent Trimming

For the merge phase one temporary segment is chosen into which the others are merged. By default the last extents for these other segments are trimmed back to free up any unused space.

This behavior can be stopped in Oracle7 by setting event 10901; in Oracle8*i* you can stop it by using the MINIMUM EXTENT default storage parameter for a tablespace.

Parallel DML and DDL (Parallel Create(Extents Management)#1)

PARALLEL CREATE : EXAMPLE OF EXTENTS(DB_BLOCK_SIZE = 2048)

```
CREATE TABLE BIG_EMP_COPY
PARALLEL (DEGREE 3)
STORAGE (INITIAL 200K NEXT 4000K)
AS SELECT * FROM BIG_EMP
```

Extents Allocated - Oracle7

EXTENT	BLOCKS	KBYTES
0	100	200
1	2000	4000
2	100	200
3	858	1716
4	100	200
5	867	1734

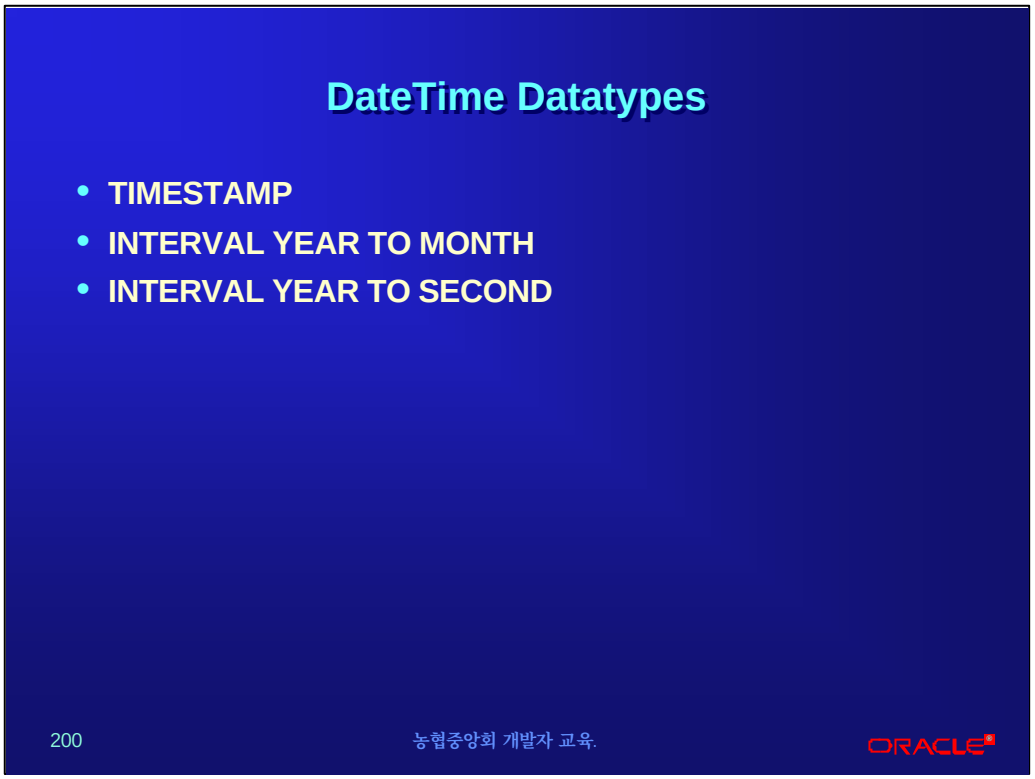
Extents Allocated >= Oracle8

EXTENT	BLOCKS	KBYTES	
0 NEXT	2000	4000	→ allocated an extent of size
1	211	422	→ NEXT alloc & trimmed down
2	272	544	→ NEXT alloc & trimmed down

So we can see each slave had an extent of 200K and allocated one extra extent - two of these (3 and 5) have been trimmed down from the 4000K expected.

7

Data Warehousing Enhancements

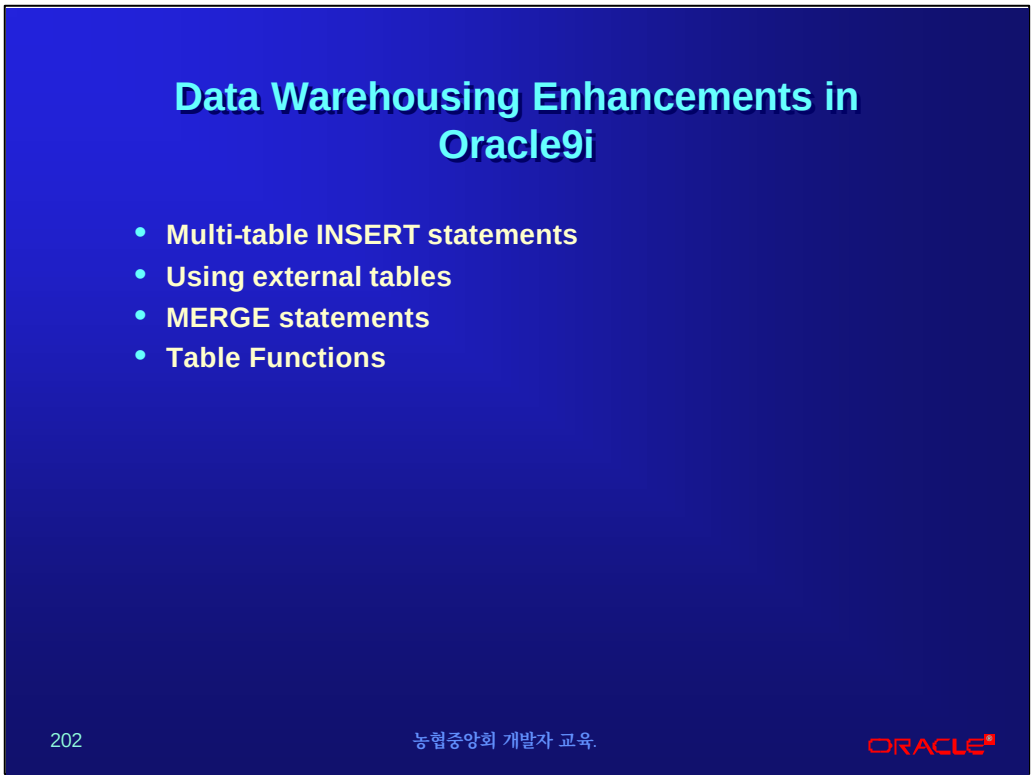


REFERENCE

Note:These Date/ Time features as well as others are covered in greater detail in the Globalization module

DateTime Functions

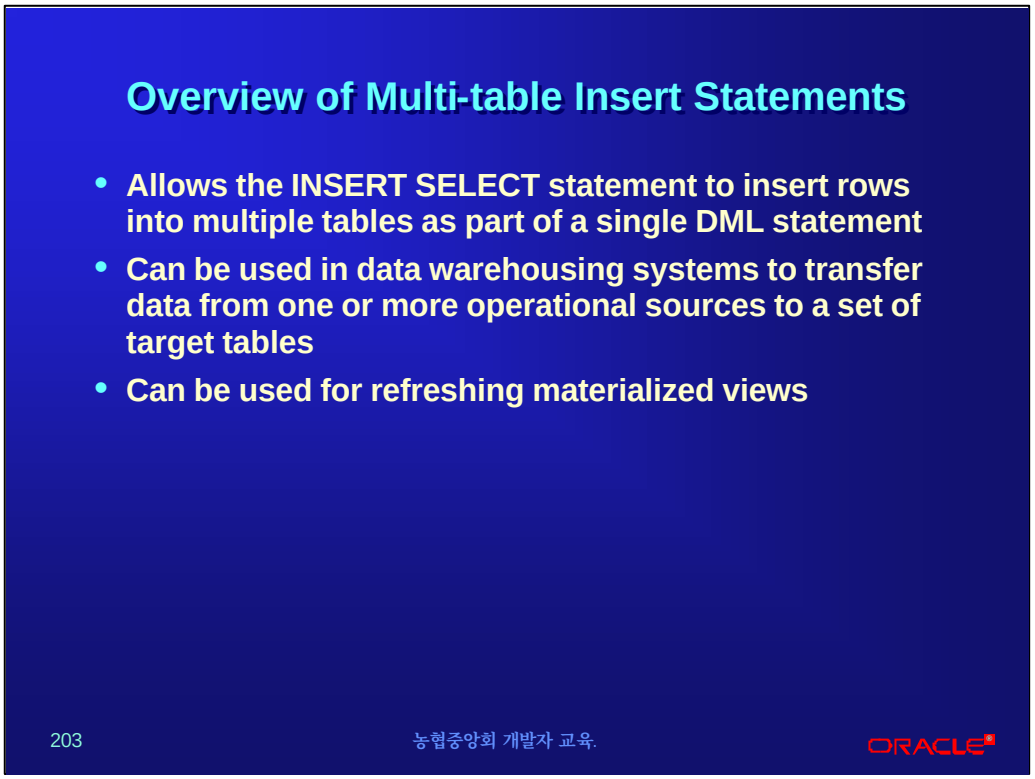
- CURRENT_DATE
- CURRENT_TIMESTAMP
- DBTIMEZONE
- EXTRACT
- FROM_TZ
- LOCALTIMESTAMP
- SESSIONTIMEZONE
- TO_TIMESTAMP
- TO_TIMESTAMP_TZ
- TO_YMINTERVAL
- TZ_OFFSET



REFERENCE

Note: These Data Warehousing features as well as others are covered in greater detail in the Business Intelligence module

Note: Table functions are discussed under the Extensibility lesson in the extensibility module and in the PL/SQL lesson in the SQL&PL/SQL module



Creating Multi-table Inserts

The INSERT AS SELECT statement with the new syntax can be parallelized and used with the direct-load mechanism. The multi table INSERT statement inserts computed rows derived from the rows returned from the evaluation of a subquery. There are two forms of the multi-table INSERT statement: unconditional and conditional.

For the unconditional form, an INTO clause list is executed once for each row returned by the subquery. For the conditional form INTO clause lists are guarded by WHEN clauses that determine whether the corresponding INTO clause list is executed.

•An INTO clause list consists of one or more INTO clauses . The execution of an INTO clause list causes the insertion of one row for each INTO clause in the list.

•An INTO clause specifies the target into which a computed row is inserted. The target specified may be any table expression that is legal for an INSERT SELECT statement. However aliases cannot be used. The same table may be specified as the target for more than one INTO clause.

•An INTO clause also provides the value of the row to be inserted using a VALUES clause . An expression used in the VALUES clause can be any legal expression , but may only refer to columns returned by the select list of the subquery. If the VALUES clause is omitted , the select list of the subquery provides the values to be inserted. If a column list is given, each column in the list is assigned a corresponding value from the VALUES clause or the subquery. If no column list is given , the computed row must provide values for all columns in the target table.

Advantages of Multi-table INSERTs

- Eliminates the need for multiple INSERT.. SELECT statements to populate multiple tables
- Eliminates the need for a procedure to do multiple INSERTs using IF.. THEN syntax
- Significant performance improvement over above two methods due to the elimination of the cost of materialization and repeated scans on the source data

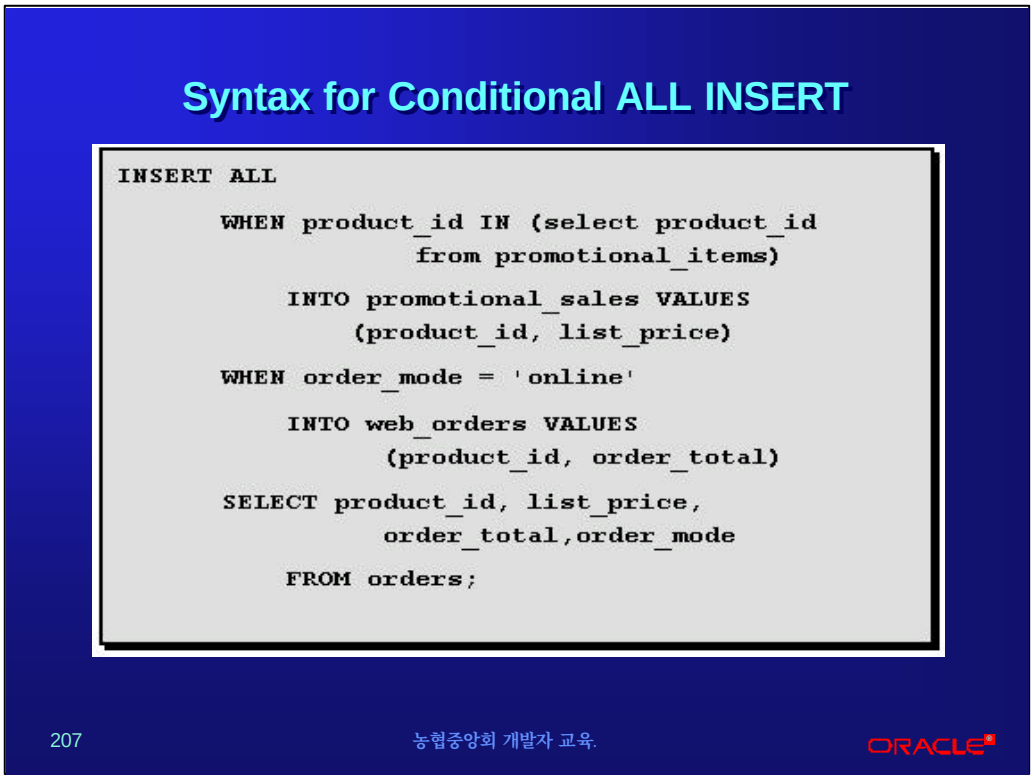
Types of Multi-table INSERT Statements

- Unconditional INSERT
- Pivoting INSERT
- Conditional ALL INSERT
- Conditional FIRST INSERT

Note: This feature is an Oracle extension to SQL and is not a SQL: 1999 standard.

Example of Unconditional INSERT

```
INSERT ALL
    INTO product_activity VALUES(today, product_id,
    quantity)
    INTO product_sales VALUES(today, product_id,
    total)
        SELECT trunc(order_date) today, product_id,
          SUM(unit_price) total, SUM(quantity) quantity
      FROM orders, order_items
         WHERE orders.order_id = order_items.order_id
           AND order_date = TRUNC(SYSDATE)
               GROUP BY product_id;
```



Syntax for Conditional ALL INSERT

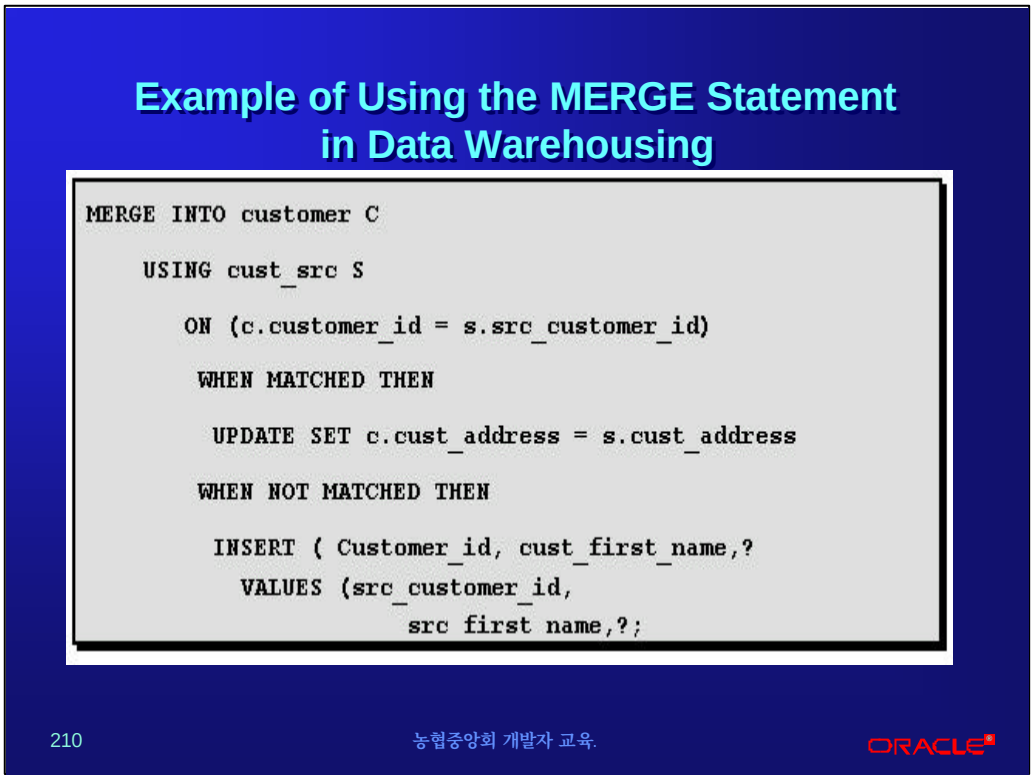
The above example inserts a row into the PROMOTIONAL_SALES table for products sold that are on the promotional list, and into the WEB_ORDERS table for products for which online orders were used. It is possible that two rows are inserted for some item lines, and none for others.

Overview of MERGE statements

- MERGE statements
  - Provide the ability to conditionally UPDATE/INSERT into the database
  - Do an UPDATE if the row exists and an INSERT if it is a new row
  - Avoid multiple updates
  - Can be used in Data Warehousing applications

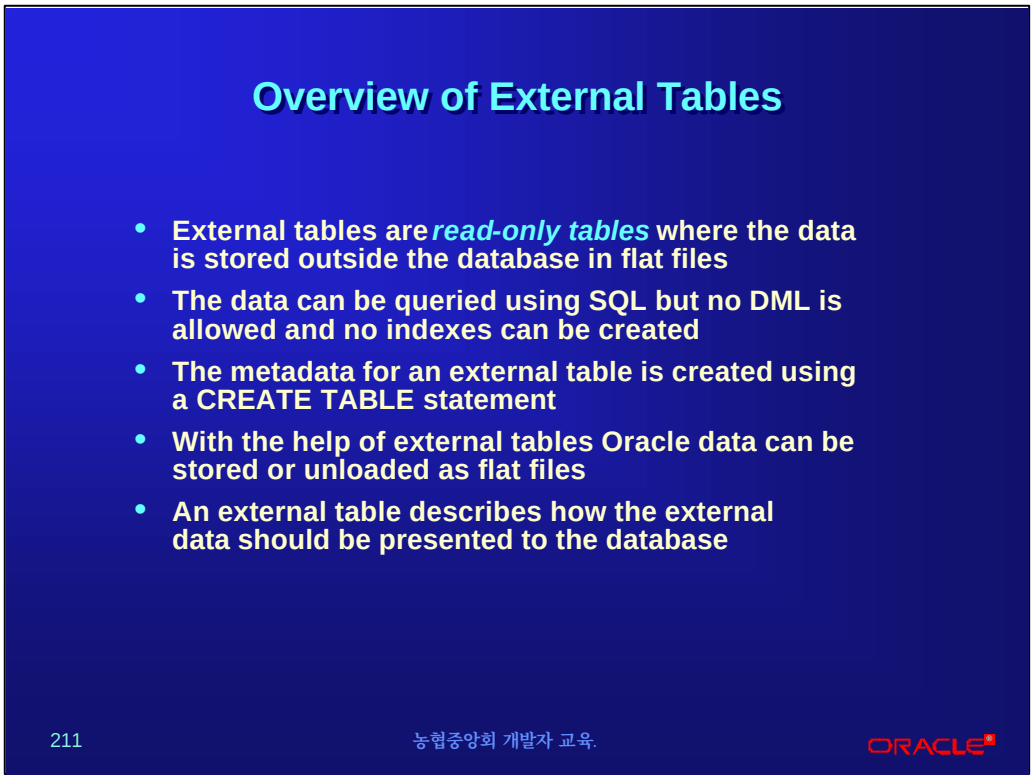
Applications of MERGE statements

- MERGE statements use a single SQL statement to complete and UPDATE or INSERT or both
- The statement can be parallelized transparently
- Bulk DML can be used
- Performance is improved because fewer statements require fewer scans of the source tables



Example of MERGE

This an example of using MERGEs in data warehousing. Customer(C) is a large fact table and cust_src is a smaller "delta" table with rows which need to be inserted into customer conditionally. This MERGE statement indicates that table customer has to be MERGEed with the rows returned from the evaluation of the ON clause of the MERGE. The ON clause in this case is the table cust_src (S), but it can be an arbitrary query. Each row from S is checked for a match to any row in C by satisfying the join condition specified by the On clause. If so, each row in C is updated using the UPDATE SET clause of the MERGE statement. If no such row exists in C then the rows are inserted into table Causing the ELSE INSERT clause.



Overview of External Tables

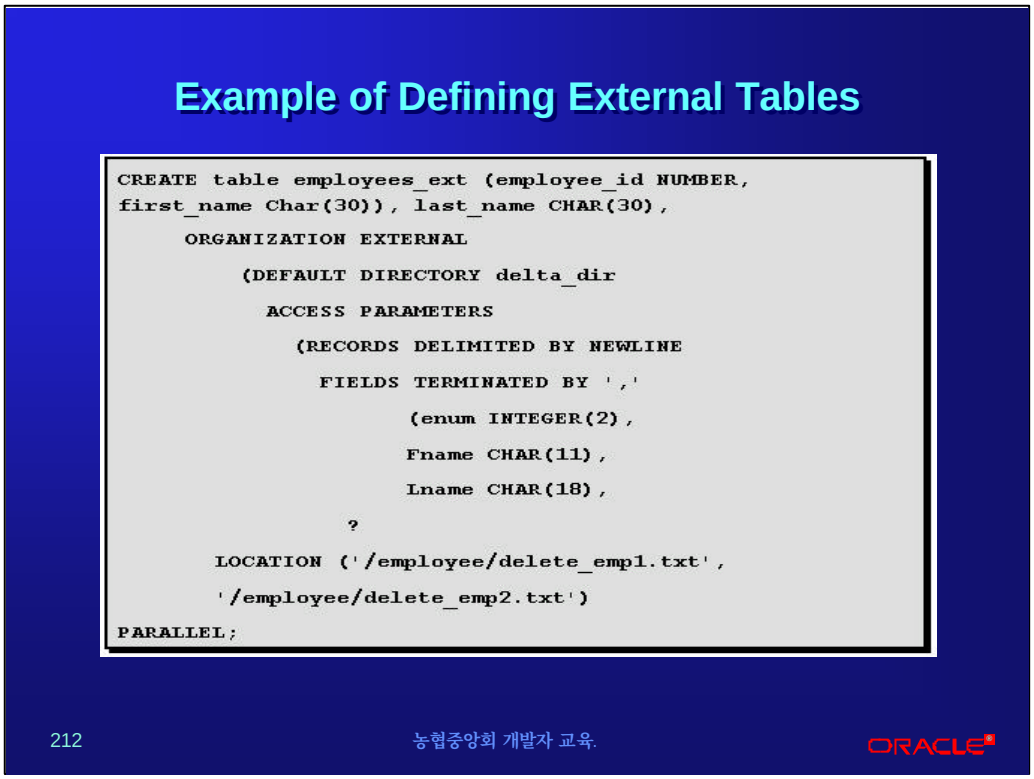
External tables are like regular SQL tables with the exception that the data is read only and does not reside in the database, thus the organization is external. The external table can be queried directly and in parallel using SQL. As a result , the external table acts as a view The metadata for the external table is created using the "CREATE TABLE ... ORGANIZATION EXTERNAL" statement.

No DML operations are possible and no indexes can be created on them. Although an INSERT statement cannot target an external organized table, a mechanism is provided to populate the external data via a "CREATE TABLE ... ORGANIZATION EXTERNAL AS SELECT" statement. Using this means Oracle data can be unloaded into flat files and published.

The CREATE TABLE . . ORGANIZATION EXTERNAL operation involves only the creation of metadata in the Oracle Dictionary since the external data already exists outside the database. Once the metadata is created, the external table feature enables the user to easily perform parallel extraction of data from the specified external sources.

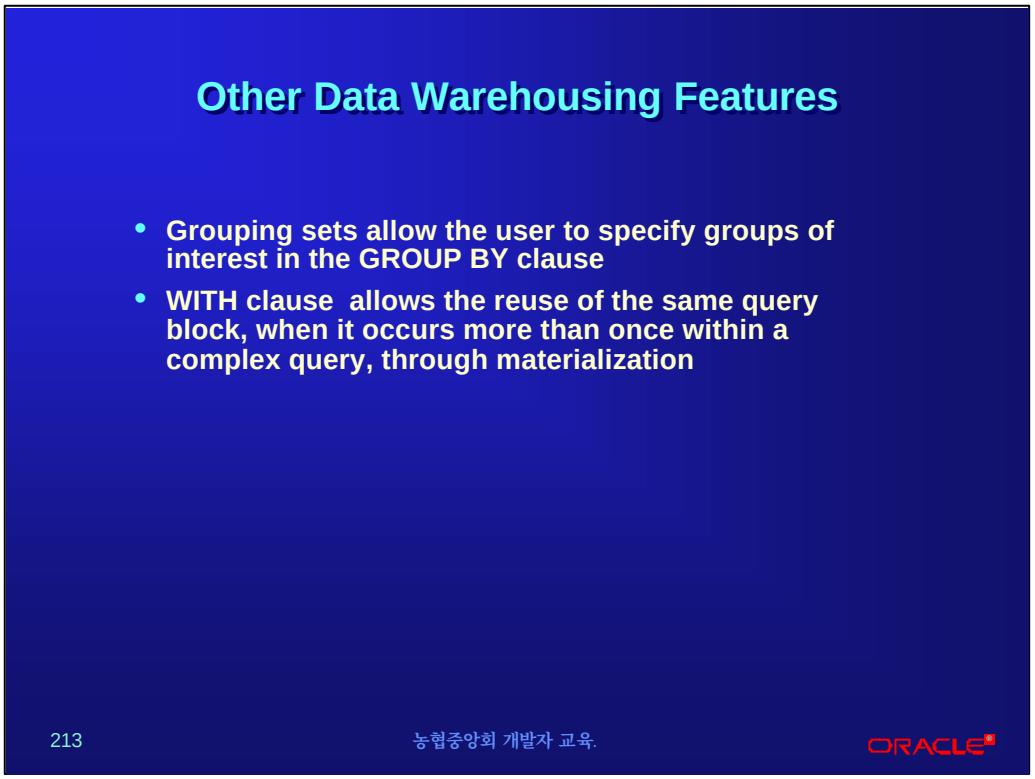
REFERENCE

Note: External tables and MERGES are covered in greater detail in the Business Intelligence module



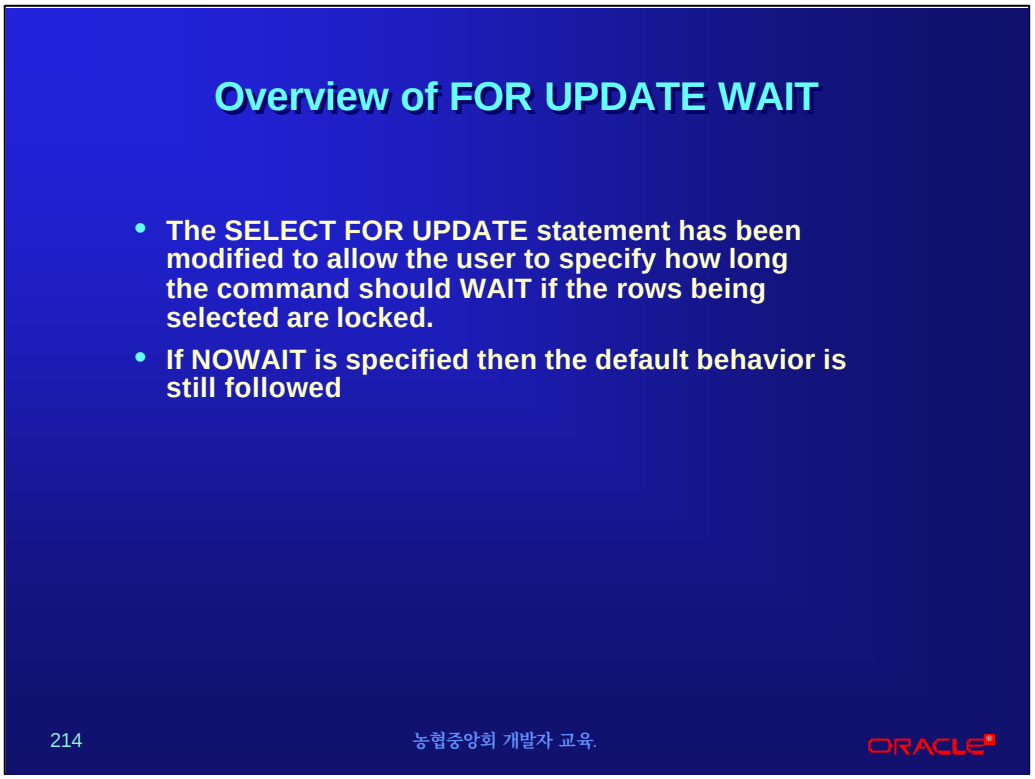
Example of Defining External Tables

In the example above an external table named employees_ext is defined. Delta_dir is a directory where the external flat files are residing. The access parameters control the extraction of data from the flat file using record and file formatting information. The directory object was introduced in Oracle8i.



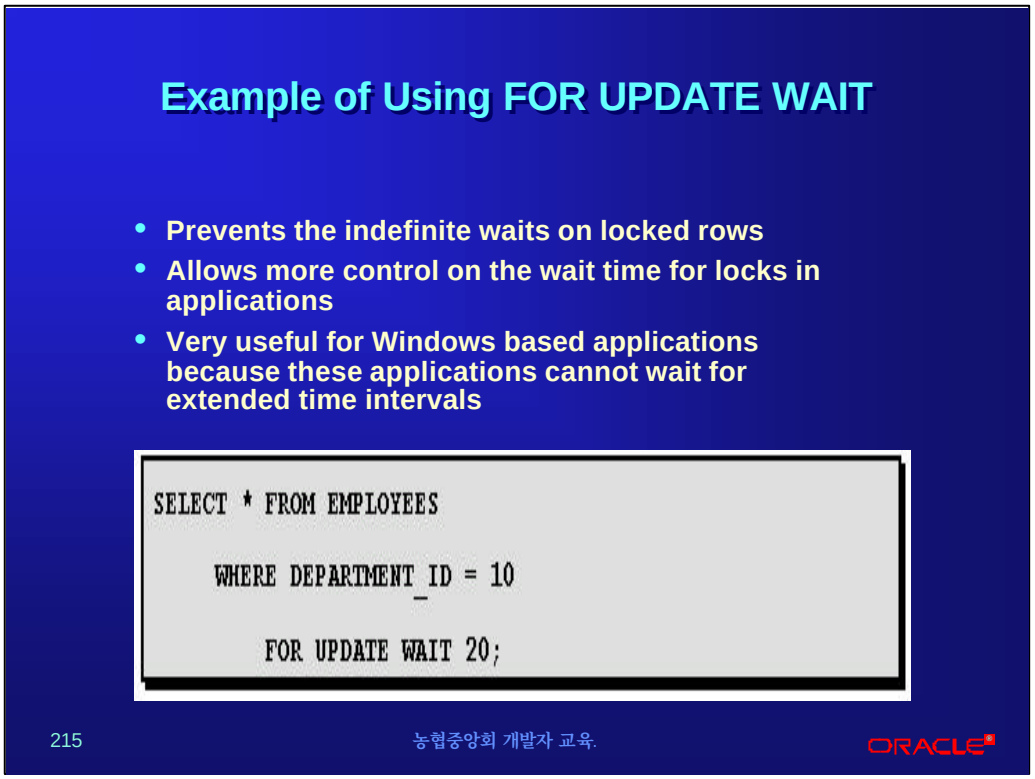
REFERENCE

Note: Grouping Sets and the WITH clause are covered in greater detail in the Business Intelligence module



Overview of FOR UPDATE WAIT

In prior releases the 'SELECT ?FOR UPDATE' statement has only two alternatives when the rows being selected are already locked: wait for the lock to be released or return immediately with an error message. Another alternative has been added in Oracle9i to allow the user to specify the time interval to wait before returning with the error.



Example of Using FOR UPDATE WAIT

An integer can be specified after the key word WAIT to indicate the number of seconds to wait for a lock. In the above example the query waits 20 seconds to obtain a lock and if it is unable to obtain the lock in that time interval it returns an error.

Free List Managed Versus Automatic Segment Space Managed Segments

- Automatic Segment Space Managed segments have the following advantages over Free List Managed segments:
  - Ease of use
  - Improved space utilization
  - Improved performance of concurrent space operations
  - Improved performance in a multi-instance environment

Free List Managed Versus Automatic Segment Space Managed Segments

The new space management implementation will provide the following benefits:

- Ease of use achieved by requiring fewer space-related parameters (no more FREELIST, FREELIST GROUPS, PCTUSED) that are often needed (and occasionally misused) today.
- Better space utilization, especially for the objects with highly varying size rows.
- Better run-time adjustment to variations in concurrent access.
- Better multi-instance behavior in terms of performance and space utilization.

8

Application & SQL Tuning사례

Case 1: Literal 비공유 SQL에 의한 시스템 Resource의 급속한 고갈 현상.

Problem

- 보통의 경우는 정상적이나, 사용자가 조금만 증가하여도 CPU사용량 100%.
- Peak시기동안 업무 Service 중단.
- Application은 MS ASP (ODBC) + Oracle Database

Tuning Key Point

- 같은 업무를 사용자가 증가하여 자원고갈이 급속도로 증가된다면 SQL공유문제를 의심할 수 있다. (특히 ODBC,JDBC,ADO,JSP, VB 등)
- 집중적으로 실행되는 Application은 공유화 형태(Bind변수 사용)로 구성되어야 한다. Literal SQL들은 비공유 되므로 관련 SQL들을 Compile하는 overhead가 있다.
- 모든 DB접속을 위한 App Method들은 공유화 기법을 제공한다.

Solution

- Application의 비공유 유형의 프로그램을 공유화 형태로 변경한다.
- 차선책으로 변경이 불가능한 경우는 CHRSOR_SHARING=SIMILAR를 이용한다.

Literal SQL유형의 Tuning 전/후의 비교 예

- Literal 비공유 SQL의 공유화로 안정성 확보 및 H/W도입 계획 취소

Tuning 전/후 Parsing율

Tuning 전/후 CPU사용율

Case 2: 시스템 도입이후 Application성능 점차적 저하와 Peak시 H/W CPU 100%.

Problem

- Open시점은 Application의 Performance가 문제가 없었으며, 급속히 성능저하현상이 나타났으며 Peak시기는 CPU사용량 100%.
- Buffer Cache Hit율이 낮아서 Buffer Cache(SGA)를 충분히 늘려줬으나 같은 현상.
- 일부 Consulting사는 H/W의 용량의 한계로 판단 함.

Tuning Key Point

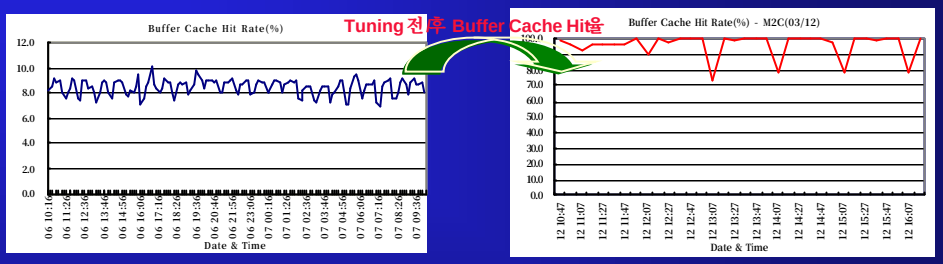
- Application이 시스템 open시에는 응답시간에 문제가 없었으나, Data량의 증가에 따른 성능저하가 발생되는 형태는 Index사용의 문제.
- Index가 있음에도 불구하고 App에서 Index를 사용하지 않는 경우는 통계정보 및 App의 구조적 문제를 의심한다.
- 통계정보의 운영 Pattern을 확인한다.

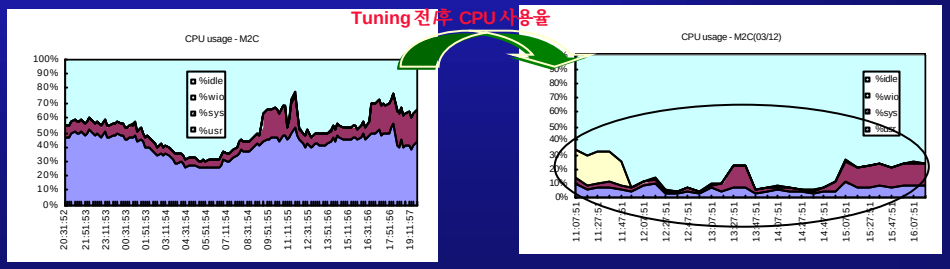
Solution

- CBO에서는 통계정보에 의해서 Plan들이 결정되므로 정확한 통계정보를 운영한다. (부정확한 통계정보는 없는 것 보다 못하다)

정상적인 통계정보 운영에 의한 Tuning 전/후의 비교 예

- Application 성능 획기적 개선, 시스템 안정성 유지(10~20% CPU) 및 H/W도입 계획 취소





Case 3: DML처리속도가 너무 느린 경우

Problem

- DML처리 속도가 느리고, CPU사용량이 높다.
- Lock현상도 자주 발생된다.

Tuning Key Point

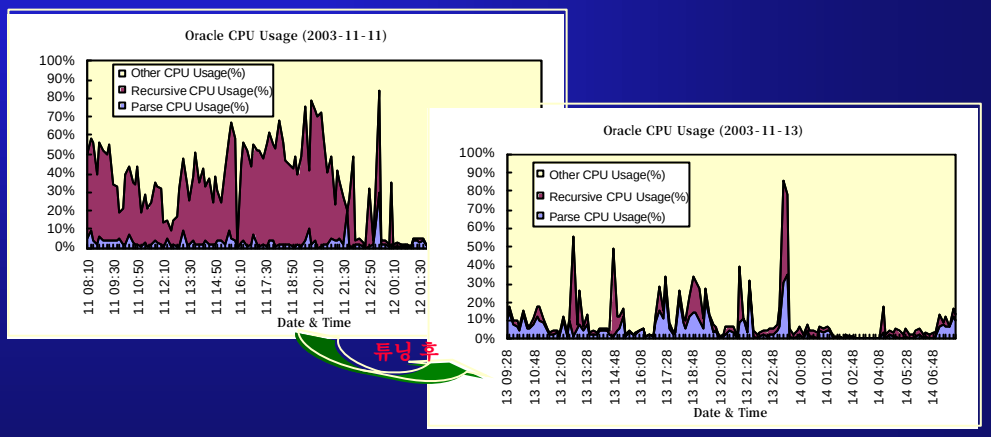
- 전체시스템에서 Execution이 높은 Top SQL을 검증한다. (V$SQL)
- Top I/O발생 SQL위주로 Tuning한다. (V$SQL)
- DML의 처리속도가 늦다면 Recursive SQL(호출된 Procedure,Function,Trigger등)을 확인한다.
- Index Split 또는 Lock,Wait정보를 확인한다.

Solution

- 시스템의 전체 Buffer I/O 82%를 차지하고 있는 Top I/O발생 Trigger 확인 및 Tuning

Critical(Top I/O) SQL Tuning 전/후의 비교 예

- Tuning 후 Recursive CPU사용량, I/O 감소. 시스템의 CPU사용량 Peak 60~70%가 ? 30~40%로 감소



Case 4: Temp Tablespace의 높은 I/O 발생

Problem

- 시스템에 전체 I/O가 높게 발생
- SORT_AREA_SIZE가 64KB로 너무 작게 설정.

Tuning Key Point

- TEMP Tablespace는 I/O를 거의 발생하지 않도록 할 수 있다.
- 각 SQL마다 필요한 만큼 최적의 Memory를 사용하기 위해서는 9i이상에서는 Automatic Workspace방법을 사용한다. (WORKAREA_SIZE_POLICY=AUTO)

Solution

- SORT_AREA_SIZE를 OLTP기준으로 Application특징을 확인 후 적절히 설정한다. (2MB or 4MB)
- 최선책은 Automatic Workspace방법을 제공하는 9i 를 사용하여 Application의 성능을 획기적으로 개선한다.
- 불필요한 Temp I/O를 거의 0% I/O상태로 바꿀 수 있다.

Sort를 위한 Memory 조정에 의한 Tuning 전/후의 비교 예

- Tuning 전 200MB/Min 의 I/O가 발생하였으나 Tuning 후의 경우는 특정 Batch Job의 경우를 제외하고 I/O의 규모가 극히 없음