5

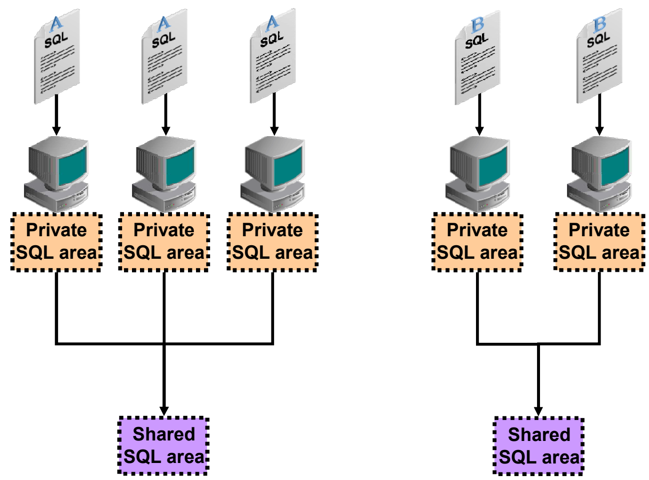
Optimizer Fundamentals

Objectives

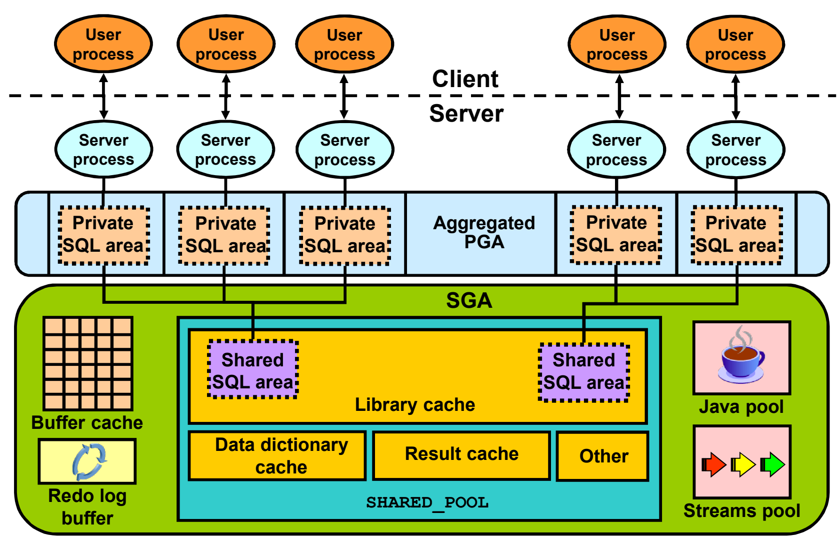
After completing this lesson, you should be able to:

- Describe each phase of SQL statement processing
- Discuss the need for an optimizer
- Explain the various phases of optimization
- Describe the quick-tuning strategy
- Control the behavior of the optimizer

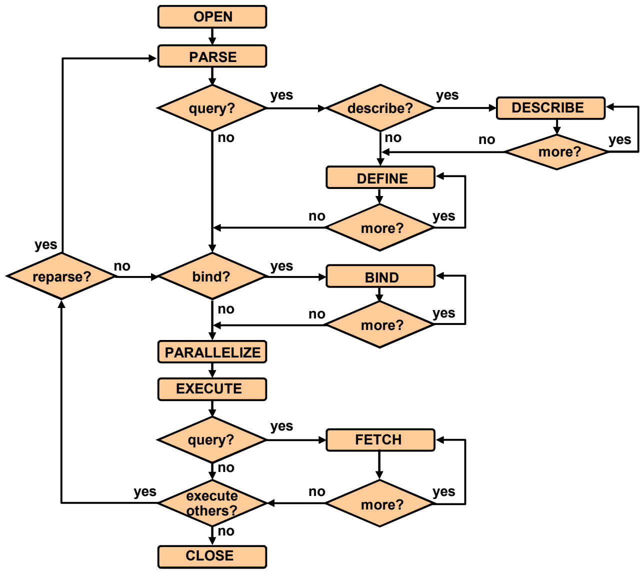
SQL Statement Representation



SQL Statement Representation



SQL Statement Processing: Overview



SQL Statement Processing: Steps

1. Create a cursor.
2. Parse the statement.
3. Describe query results.
4. Define query output.
5. Bind variables.
6. Parallelize the statement.
7. Execute the statement.
8. Fetch rows of a query.
9. Close the cursor.

Step 1: Create a Cursor

- A cursor is a handle or name for a private SQL area.
- It contains information for statement processing.
- It is created by a program interface call in expectation of a SQL statement.
- The cursor structure is independent of the SQL statement that it contains.

Step 2: Parse the Statement

- During the parse call, Oracle Database always:
  - Checks the syntax
  - Checks semantics and privileges
- Statement passed from the user process to the Oracle instance
- Parsed representation of SQL created and moved into the shared SQL area if there is no identical SQL in the shared SQL area
- Can be reused if identical SQL exists

Steps 3 and 4: Describe and Define

- The Describe step provides information about the select list items; it is relevant when entering dynamic queries through an OCI application.
- The Define step defines location, size, and data type information that is required to store fetched values in variables.

Steps 5 and 6: Bind and Parallelize

- Bind any bind values:
  - Enables memory address to store data values
  - Allows shared SQL even though bind values may change
- Parallelize the statement:
  - `SELECT`
  - `INSERT`
  - `UPDATE`
  - `MERGE`
  - `DELETE`
  - `CREATE`
  - `ALTER`

Steps 7 Through 9: Execute, Fetch Rows, Close the Cursor

- Execute: Drives the SQL statement to produce the desired results
- Fetch rows:
  - Into defined output variables
  - Query results returned in table format
  - Array fetch mechanism
- Close the cursor.

SQL Statement Processing PL/SQL: Example

```
SQL> variable c1 number
SQL> execute :c1 := dbms_sql.open_cursor;
```

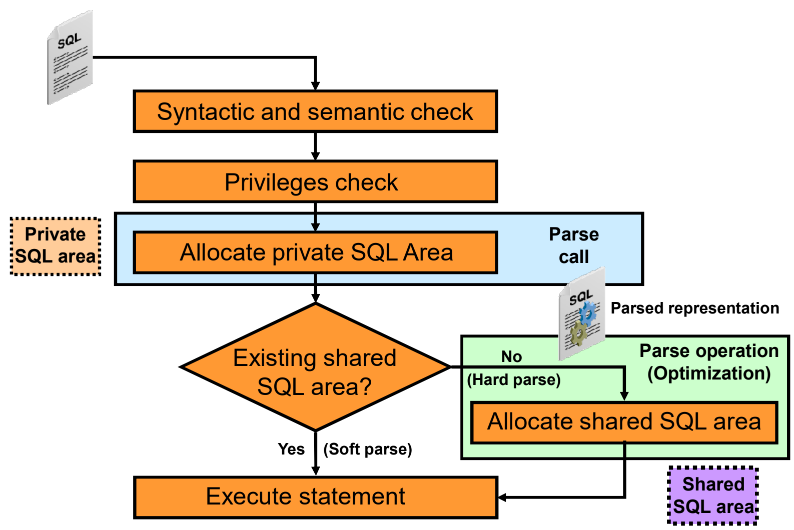
```
SQL> variable b1 varchar2
SQL> execute dbms_sql.parse
  2  (:c1
  3  ,'select null from dual where dummy = :b1'
  4  ,dbms_sql.native);
```

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

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

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

SQL Statement Parsing: Overview

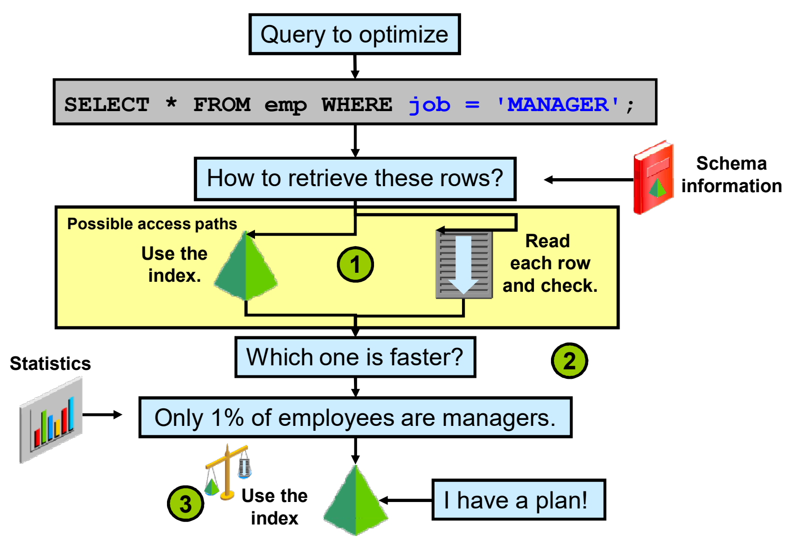


Quiz

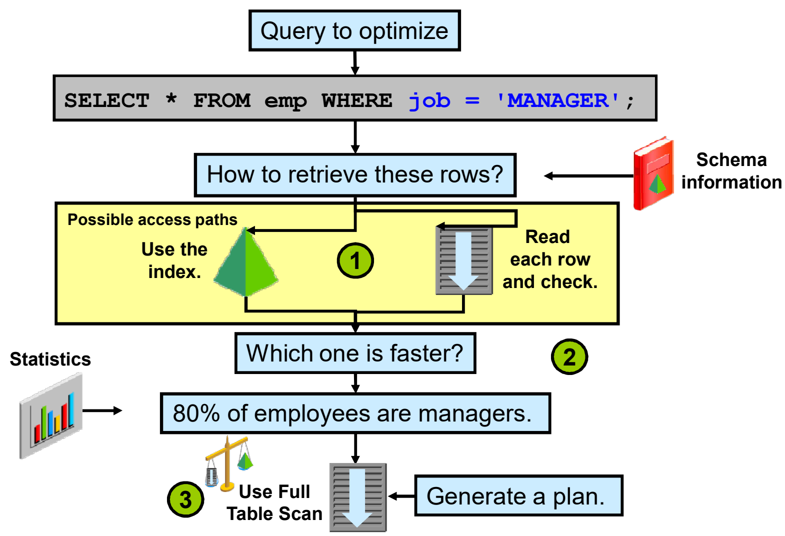
The __________ step provides information about the select list items and is relevant when entering dynamic queries through an OCI application.

a. Parse
b. Define
c. Describe
d. Parallelize

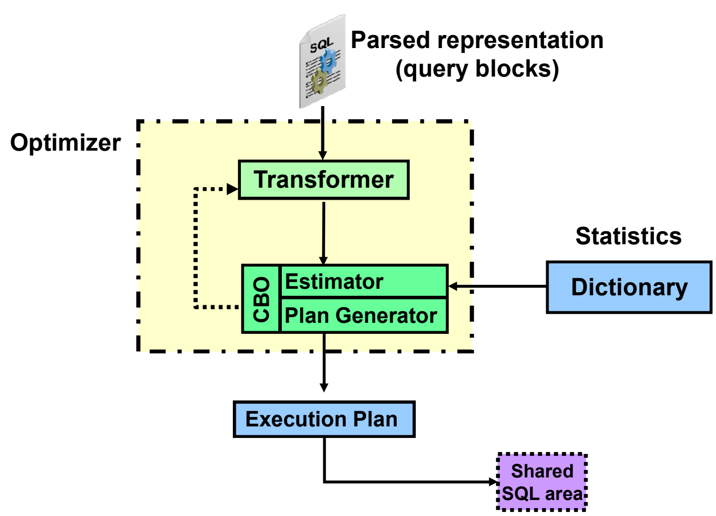
Why Do You Need an Optimizer?



Why Do You Need an Optimizer?



Components of the Optimizer



Query Transformer

- Determines whether it is advantageous to rewrite the original SQL statement into a semantically equivalent SQL statement that can be processed more efficiently
- Possible query transformation techniques:
  - `OR` expansion
  - Subquery Unnesting (SU)
  - Complex View Merging (CVM)
  - Join Predicate Push Down (JPPD)
  - Transitive Closure
  - IN into EXISTS (semijoins)
  - NOT IN into NOT EXISTS (antijoins)
  - Filter Push Down (FPD)
- See the list of New Query Transformations in 11*g.*

Transformer: `OR` Expansion Example

- Original query:

B*-tree Index

```
SELECT *
  FROM emp
  WHERE job = 'CLERK' OR deptno = 10;
```

- Equivalent transformed query:

```
SELECT *
  FROM emp
  WHERE job = 'CLERK'
UNION ALL
SELECT *
  FROM emp
  WHERE deptno = 10 AND job <> 'CLERK';
```

Transformer: Subquery Unnesting Example

- Original query:

```
SELECT *
  FROM accounts
  WHERE custno IN
        (SELECT custno FROM customers);
```

- Equivalent transformed query:

```
SELECT accounts.*
  FROM accounts, customers
  WHERE accounts.custno = customers.custno;
```

Primary or unique key

Transformer: View Merging Example

- Original query:

Index

```
CREATE VIEW emp_10 AS
     SELECT empno, ename, job, sal, comm, deptno
     FROM emp
     WHERE deptno = 10;
```

```
SELECT empno FROM emp_10 WHERE empno > 7800;
```

- Equivalent transformed query:

```
SELECT empno
  FROM emp
  WHERE deptno = 10 AND empno > 7800;
```

Transformer: Predicate Pushing Example

- Original query:

Index

```
CREATE VIEW two_emp_tables AS
SELECT empno, ename, job, sal, comm, deptno FROM emp1
 UNION
SELECT empno, ename, job, sal, comm, deptno FROM emp2;
```

```
SELECT ename FROM two_emp_tables WHERE deptno = 20;
```

- Equivalent transformed query:

```
SELECT ename
  FROM ( SELECT empno, ename, job,sal, comm, deptno
           FROM emp1 WHERE deptno = 20
          UNION
         SELECT empno, ename, job,sal, comm, deptno
           FROM emp2 WHERE deptno = 20 );
```

Transformer: Transitivity Example

- Original query:

Index

```
SELECT *
 FROM emp, dept
 WHERE emp.deptno = 20 AND emp.deptno = dept.deptno;
```

- Equivalent transformed query:

```
SELECT *
 FROM emp, dept
 WHERE emp.deptno = 20 AND emp.deptno = dept.deptno
   AND dept.deptno = 20;
```

Hints for Query Transformation

The following hints instruct the optimizer to use a specific SQL query transformation:

- NO_QUERY_TRANSFORMATION
- USE_CONCAT
- NO_EXPAND
- REWRITE and NO_REWRITE
- MERGE and NO_MERGE
- STAR_TRANSFORMATION and NO_STAR_TRANSFORMATION
- FACT and NO_FACT
- UNNEST and NO_UNNEST

Full Notes Page

Full Notes Page

Quiz

A SQL statement used to take about 1 minute in 9*i*, but after upgrading to 11*g,* it takes 5 seconds. How was this accomplished in 11*g*?

a. Used a query transformer
b. Estimated and compared the cost of each plan
c. Selected the plan with the lowest cost
d. All of the above

Quiz

Review the following query and execution plan. Which statements are true in 11*g*?

```
SELECT p.prod_id, v1.cnt
FROM products p,(SELECT s.prod_id, count(*) cnt
                 FROM sales s
                 WHERE s.quantity_sold BETWEEN 1 AND 47
                 GROUP BY s.prod_id) v1
WHERE p.supplier_id = 12
AND p.prod_id = v1.prod_id(+);
```

```
--------------------------------------------------------------------------------------------
| Operation                              | Name           | Rows   | Bytes  | Cost  (%CPU)|
--------------------------------------------------------------------------------------------
| SELECT STATEMENT                       |                |      1 |     20 |   420    (0)|
|  NESTED LOOPS OUTER                    |                |      1 |     20 |   420    (0)|
|   TABLE ACCESS FULL                    | PRODUCTS       |      1 |      7 |     3    (0)|
|   VIEW PUSHED PREDICATE                |                |      1 |     13 |   417    (0)|
|    FILTER                              |                |        |        |             |
|     SORT AGGREGATE                     |                |      1 |      7 |             |
|      PARTITION RANGE ALL               |                |  12762 |  89334 |   417    (0)|
|       TABLE ACCESS BY LOCAL INDEX ROWID| SALES          |  12762 |  89334 |   417    (0)|
|        BITMAP CONVERSION TO ROWIDS     |                |        |        |             |
|         BITMAP INDEX SINGLE VALUE      | SALES_PROD_BIX |        |        |             |
--------------------------------------------------------------------------------------------
```

Quiz

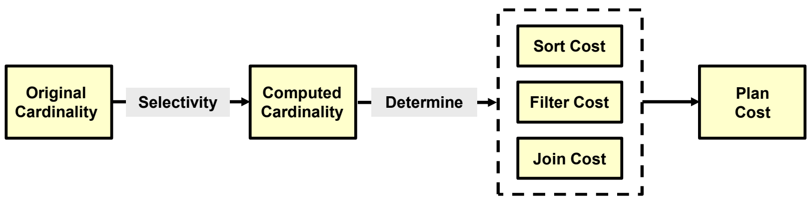
a. A query transformation has taken place.
b. The optimizer considered a nested loop join when table `p` and view `v1` are joined even with the `GROUP BY` clause in the view.
c. The optimizer considered only two possible join methods: a hash join and sort merge join to join table `p` and view `v1`.
d. A join predicate pushdown has become possible, and you are now taking advantage of the index on the `SALES` table to do a nested loop join instead of a hash join.

Estimator: Selectivity and Cardinality

- Selectivity is the estimated proportion of a row set retrieved by a particular predicate or combination of predicates.
  - Selectivity =
    Number of rows satisfying a condition / Total number of rows
- Expected number of rows retrieved by a particular operation in the execution plan
  - Cardinality = Total number of rows * Selectivity
- Selectivity is expressed as a value between 0.0 and 1.0:
  - High selectivity: Small proportion of rows
  - Low selectivity: Big proportion of rows
- Selectivity computation:
  - If no statistics: Use dynamic sampling.
  - If no histograms: Assume even distribution of rows.

Importance of Selectivity and Cardinality

- Selectivity affects the estimates of I/O cost.
- Selectivity affects the sort cost.
- Cardinality is important to determine join, filters, and sort costs.
- If incorrect selectivity and cardinality are used, the optimizer estimates the plan cost incorrectly.



Selectivity and Cardinality: Example

Facts:

- The number of rows in the CUSTOMERS table is 55500.
- The number of distinct values in:
  - CUST_CITY: 620
  - CUST_STATE_PROVINCE: 145
  - COUNTRY_ID: 19

```
SELECT * FROM customers
WHERE cust_city = 'EDISON'
AND cust_state_province = 'NJ'
AND country_id = 12345;
```

Questions:

- What is the selectivity of the WHERE predicate?
- What is the computed cardinality for the same predicate?

Selectivity and Cardinality: Example

Facts:

- The number of rows in the CUSTOMERS table is 55500.
- The number of distinct values in:
  - CUST_CITY: 620
  - CUST_STATE_PROVINCE: 145
  - COUNTRY_ID: 19


```
SELECT * FROM customers
WHERE cust_city = 'EDISON' AND
  cust_state_province = 'NJ' AND country_id = 12345;
```

Questions:

- Is the estimated selectivity the same as the actual selectivity? If not, describe why it is different.

Estimator: Cost

- Cost is the optimizer's best estimate of the number of standardized I/Os it takes to execute a particular statement.
- Cost unit is a standardized single block random read:
  - 1 cost unit = 1 SRd
- Example:
  - If a plan costs 1,000, the optimizer computes that it should take as long as 1,000 single-block reads.
  - Remember that it is an estimation.

Estimator: Cost Components

- The query optimizer uses disk I/O, CPU usage, and memory usage as units of work.
  - The operations can be scanning a table, accessing rows from a table by using an index, joining two tables together, or sorting a row set.
- The cost formula combines three different costs units into standard cost units.

$$\text{Cost} = \frac{\underbrace{\texttt{\#SRds*sreadtim}}_{\text{Single-block I/O cost}} + \underbrace{\texttt{\#MRds*mreadtim}}_{\text{Multiblock I/O cost}} + \underbrace{\texttt{\#CPUCycles/cpuspeed}}_{\text{CPU cost}}}{\texttt{sreadtim}}$$

`#SRds`: Number of single block reads
`#MRds`: Number of multiblock reads
`#CPUCycles`: Number of CPU Cycles

`Sreadtim`: Single block read time
`Mreadtim`: Multiblock read time
`Cpuspeed`: Millions instructions per second

Plan Generator

```
select e.last_name, d.department_name
from   employees e, departments d
where  e.department_id = d.department_id;
```

```
Join order[1]:  DEPARTMENTS[D]#0  EMPLOYEES[E]#1
 NL Join:  Cost: 41.13  Resp: 41.13  Degree: 1
 SM cost: 8.01
 HA cost: 6.51
Best:: JoinMethod: Hash
Cost: 6.51  Degree: 1  Resp: 6.51  Card: 106.00
Join order[2]:  EMPLOYEES[E]#1  DEPARTMENTS[D]#0
 NL Join:  Cost: 121.24  Resp: 121.24  Degree: 1
 SM cost: 8.01
 HA cost: 6.51
Join order aborted
Final cost for query block SEL$1 (#0)
All Rows Plan:
Best join order: 1
+-----------------------------------------------------------------+
| Id  | Operation            | Name        | Rows   | Bytes | Cost  |
+-----------------------------------------------------------------+
| 0   | SELECT STATEMENT     |             |        |       |     7 |
| 1   |  HASH JOIN           |             |    106 |  6042 |     7 |
| 2   |   TABLE ACCESS FULL  | DEPARTMENTS|     27 |   810 |     3 |
| 3   |   TABLE ACCESS FULL  | EMPLOYEES   |    107 |  2889 |     3 |
+-----------------------------------------------------------------+
```

Quiz

Background

- Suspose that a customer reported a problem query that takes 10 minutes to execute. The explain plan for that query is about the same as the row source plan from the `TKProf` showing 10 minutes. The cost of that explain plan is 2,000.

Questions

- Was the optimizer quite accurate when it computed the cost as 2,000?

Quick Solution Strategy

- Set a tuning goal to find a plan quickly by changing high-level settings of the optimizer.
- Find a designed plan by using basic techniques:
  - Try the SQL Tuning Advisor (STA) if possible.
  - Change the optimizer mode.
  - Dynamic Sampling
  - `OPTIMIZER_FEATURE_ENABLE`
  - Change other parameters.
  - Replace bind variables with literals.
- Implement the new good plan.
- Follow up.

Controlling the Behavior of the Optimizer

- **CURSOR_SHARING**: `SIMILAR`, `EXACT`, `FORCE`
- **DB_FILE_MULTIBLOCK_READ_COUNT**
- **PGA_AGGREGATE_TARGET**
- **STAR_TRANSFORMATION_ENABLED**
- **RESULT_CACHE_MODE**: `MANUAL`, `FORCE`
- **RESULT_CACHE_MAX_SIZE**
- **RESULT_CACHE_MAX_RESULT**
- **RESULT_CACHE_REMOTE_EXPIRATION**

Controlling the Behavior of the Optimizer

- **OPTIMIZER_INDEX_CACHING**
- **OPTIMIZER_INDEX_COST_ADJ**
- **OPTIMIZER_FEATURES_ENABLED**
- **OPTIMIZER_MODE**: ALL_ROWS, FIRST_ROWS, FIRST_ROWS_*n*
- **OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES**
- **OPTIMIZER_USE_SQL_PLAN_BASELINES**
- **OPTIMIZER_DYNAMIC_SAMPLING**
- **OPTIMIZER_USE_INVISIBLE_INDEXES**
- **OPTIMIZER_USE_PENDING_STATISTICS**

Optimizer Features and Oracle Database Releases

`OPTIMIZER_FEATURES_ENABLED`

Features	9.0.0 to 9.2.0	10.1.0 to 10.1.0.5	10.2.0 to 10.2.0.2	11.1.0.6
Index fast full scan	✓	✓	✓	✓
Consideration of bitmap access to paths for tables with only B-tree indexes	✓	✓	✓	✓
Complex view merging	✓	✓	✓	✓
Peeking into user-defined bind variables	✓	✓	✓	✓
Index joins	✓	✓	✓	✓
Dynamic sampling		✓	✓	✓
Query rewrite enables		✓	✓	✓
Skip unusable indexes		✓	✓	✓
Automatically compute index statistics as part of creation		✓	✓	✓
Cost-based query transformations		✓	✓	✓
Allow rewrites with multiple Materialized Views (MVs) and/or base tables			✓	✓
Adaptive cursor sharing				✓
Use extended statistics to estimate selectivity				✓
Use native implementation for full outer joins				✓
Partition pruning using join filtering				✓
Group by placement optimization				✓
Null aware antijoins				✓

Summary

In this lesson, you should have learned how to:

- Describe each phase of SQL statement processing
- Discuss the need for an optimizer
- Explain the various phases of optimization
- Describe the quick-tuning strategy
- Control the behavior of the optimizer

Practice 5: Overview

This practice covers exploring a trace file to understand the optimizer’s decisions.