ADVANCED TOPICS IN DATABASES

QUERIES OPTIMIZATION

Master in Informatics Engineering
Data Engineering

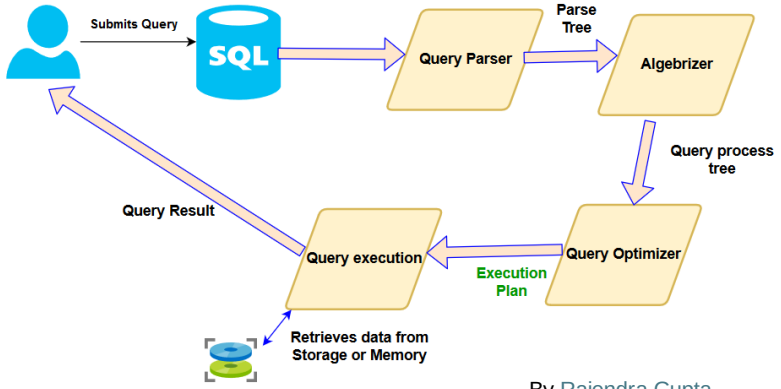
Informatics Engineering Department

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

SQL Query Optimization

- SQL Query optimization is defined as the iterative process of enhancing the performance of a query in terms of execution time, the number of disk accesses, and many more cost measuring criteria.

- Query processing is defined as the group of phases associated with the extraction of data from the database.
  - It includes conversion of queries written in a high-level language such as
    - SQL into a form that can be **understood by** the **physical level implementation** of the database
    - SQL query **optimization techniques**, and the **original evaluation of the query.**



By Rajendra Gupta

SQL Query Optimization

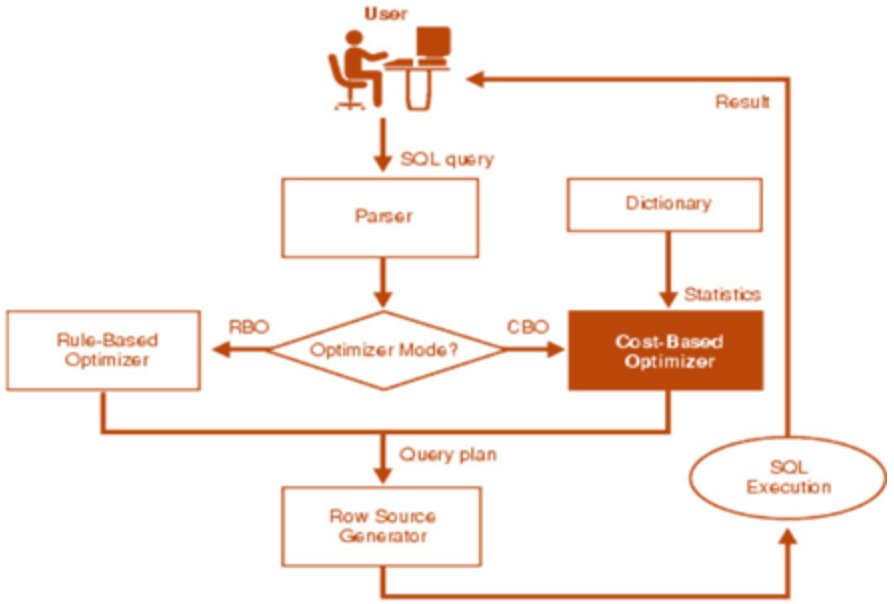
Purpose of SQL Query Optimization

The major purposes of SQL Query optimization are:

1. **Reduce Response Time:** The major goal is to enhance performance by reducing the response time.
    - The time difference between users requesting data and getting responses should be minimized for a better user experience.
2. **Reduced CPU execution time:** The CPU execution time of a query must be reduced so that faster results can be obtained.
3. **Improved Throughput:** The number of resources to be accessed to fetch all necessary data should be minimized.
    - The number of rows to be fetched in a particular query should be in the most efficient manner such that the least number of resources are used.

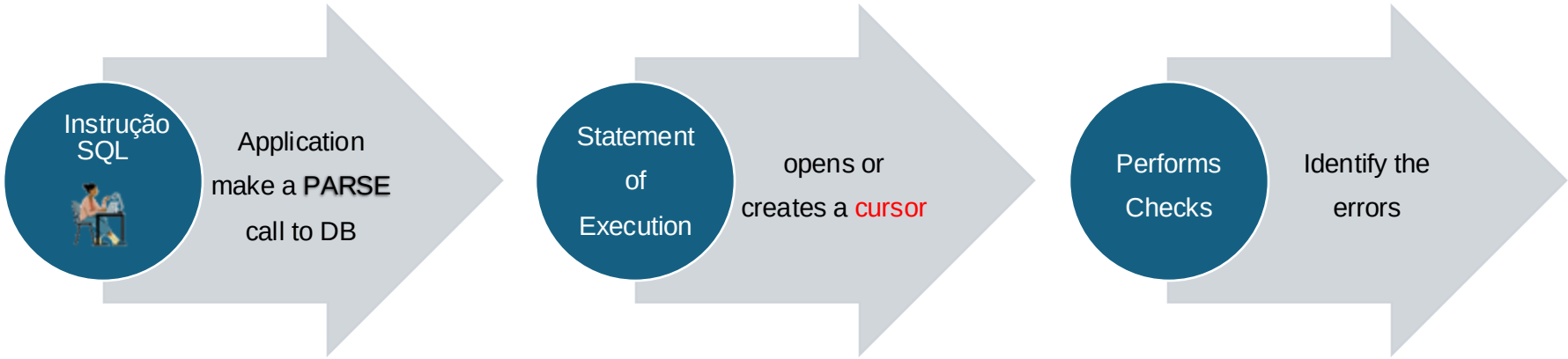
SQL Query Optimization

- **Query optimization divide into two types:**
  - **Heuristic** (sometimes called Rule based)
    - A query can be represented as a tree data structure.
    - Operations are at the interior nodes and data items (tables, columns) are at the leaves.
    - The query is evaluated in a depth-first pattern.
  - **Systematic** (Cost based)
    - is based on the cost of the query that to be optimize
    - cost calculation is based on the operations in use and estimated row number. In the end the cost value serves as the benchmark for picking the “best” execution plan



Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

SQL Query Optimization

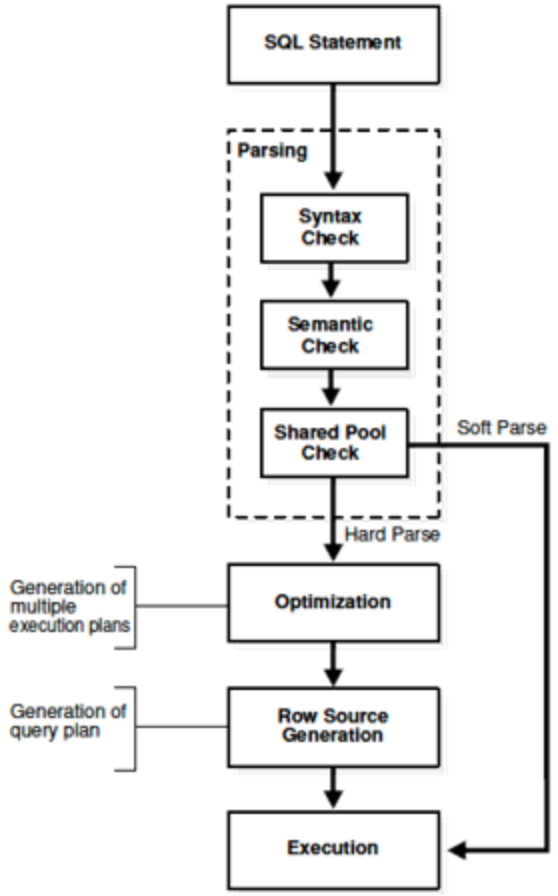


is a handle for the session-specific private SQL area that holds a parsed SQL statement and other processing information

Some errors cannot be detected by analysis

SQL Query Optimization

- **Stages:**
  1. **Parsing** - involves separating the pieces of a SQL statement into a data structure that other routines can process.
  2. **SQL Optimization** – Database must perform a hard parse at least once for every unique DML statement and performs the optimization during this parse.
  3. **Row Source Generation-** is software that receives the optimal execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database



SQL Parsing Stage

- **Steps:**

  1- Syntax Check - the database performs checks that identify the errors that can be found before statement execution. Some errors cannot be caught by parsing.

  2- Semantic Check - determines whether a statement is meaningful, for example, whether the objects and columns in the statement exist.

  3- **Shared Pool Check** - to determine whether it can skip resource-intensive steps of statement processing.

  To this end, the database uses a hashing algorithm to generate a hash value (SQL ID) for every SQL statement.

SQL Parsing Stage

Instrução SQL

Application make a **PARSE** call to DB

Statement of Execution

opens or creates a cursor

is a handle for the session-specific private SQL area that holds a parsed SQL statement and other processing information

Performs Checks

Identify the errors

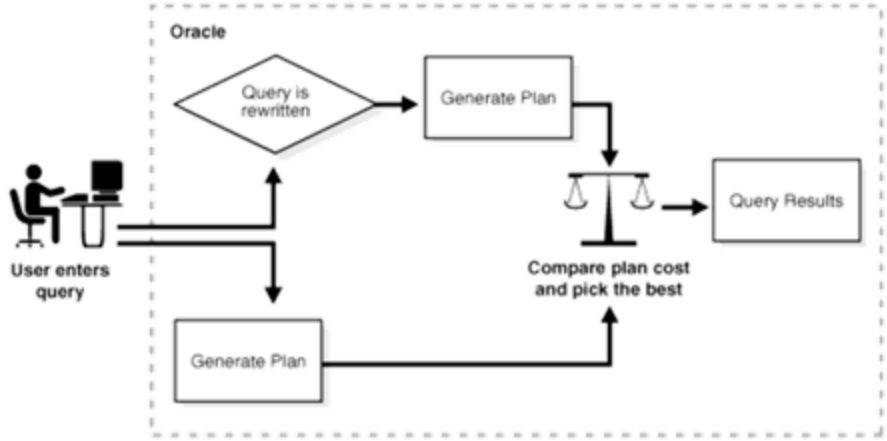
Some errors cannot be detected by analysis

SQL Parsing Stage

Parse operations fall into the following categories:

- ***Hard parse*** *– If Database cannot reuse existing code, then it must build a new executable version of the application code.*

- ***Soft parse -*** *If the submitted statement is the same as a reusable SQL statement in the shared pool, then Database reuses the existing code. This reuse of code is also called a library cache hit.*



Transparent Query Rewrite

SQL Optimization Stage

- **Application of rules** to the internal **data structures** of the **query to transform** these structures into **equivalent, but more efficient representations**.
  - The rules can be based:
    - upon mathematical models of the **relational algebra expression and tree (heuristics),**
    - upon **cost estim**ates of **different algorithms applied to operations o**r upon **the semantics within the query and the relations it involves.**
  - **Selecting the proper rules to apply**, **when** to apply them and **how** they are applied is the **function of the query optimization engine.**

Row Source Generation Stage

- The iterative execution **plan produced *is a binary program*** *that, when executed by the SQL engine,* ***produces the result set.***
  - *Each step returns a row set. The next step either uses the rows in this set, or the last step returns the rows to the application issuing the SQL statement.*
- *A row source is a row set returned by a step in the execution plan along with a control structure that can iteratively process the rows.*
  - *The row source can be a table, view, or result of a join or grouping operation.*

Row Source Generation Stage

- *The row source generator produces a **row source tree**, which is a collection of row sources.*

- *The row source tree shows the following information:*
  - ***An ordering of the t**ables referenced by the statement*
  - *An **access method** for each table mentioned in the statement*
  - *A **join method for tables** affected by join operations in the statement*
  - *Data operations such as filter, sort, or aggregation*

Row Source Generation Stage

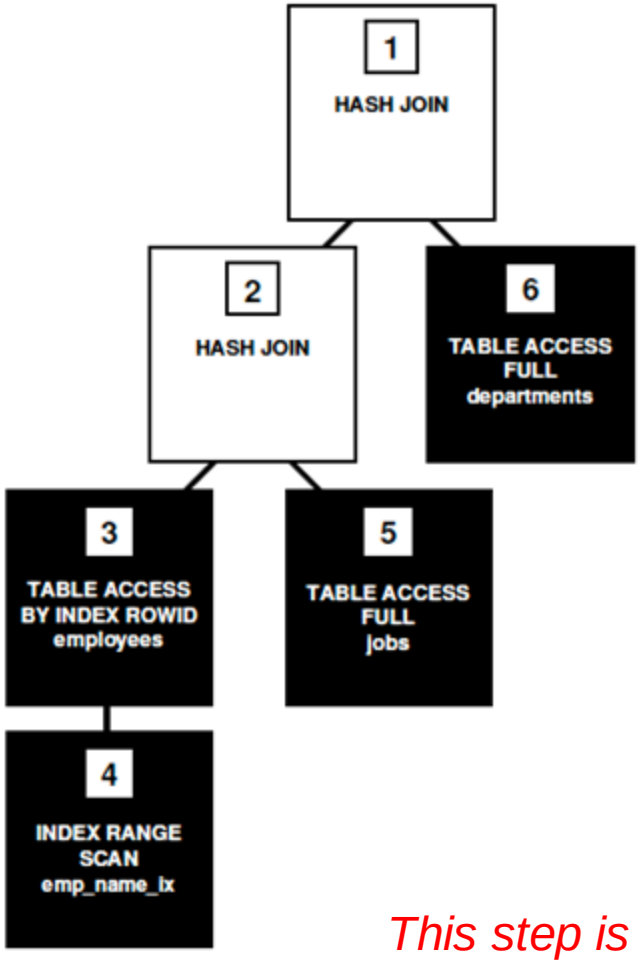
Example of execution Plan :

```
SELECT e.last_name, j.job_title, d.department_name
FROM hr.employees e, hr.departments d, hr.jobs j
WHERE e.department_id = d.department_id
AND e.job_id = j.job_id
AND e.last_name LIKE 'A%';
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 975837011

-------------------------------------------------------------------------------
----
| Id| Operation                    | Name      |Rows|Bytes|Cost(%CPU)|
Time|
-------------------------------------------------------------------------------
----
| 0| SELECT STATEMENT             |           | 3 |189 |7(15)|
00:00:01 |
|*1|  HASH JOIN                   |           | 3 |189 |7(15)|
00:00:01 |
|*2|   HASH JOIN                  |           | 3 |141 |5(20)|
00:00:01 |
| 3|    TABLE ACCESS BY INDEX ROWID| EMPLOYEES   | 3 | 60 |2 (0)|
00:00:01 |
|*4|     INDEX RANGE SCAN         | EMP_NAME_IX | 3 |    |1 (0)|
00:00:01 |
| 5|    TABLE ACCESS FULL         | JOBS        |19 |513 |2 (0)|
00:00:01 |
| 6|   TABLE ACCESS FULL          | DEPARTMENTS |27 |432 |2 (0)|
00:00:01 |
-------------------------------------------------------------------------------
----

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

   1 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
   2 - access("E"."JOB_ID"="J"."JOB_ID")
   4 - access("E"."LAST_NAME" LIKE 'A%')
       filter("E"."LAST_NAME" LIKE 'A%')
```

SQL Execution

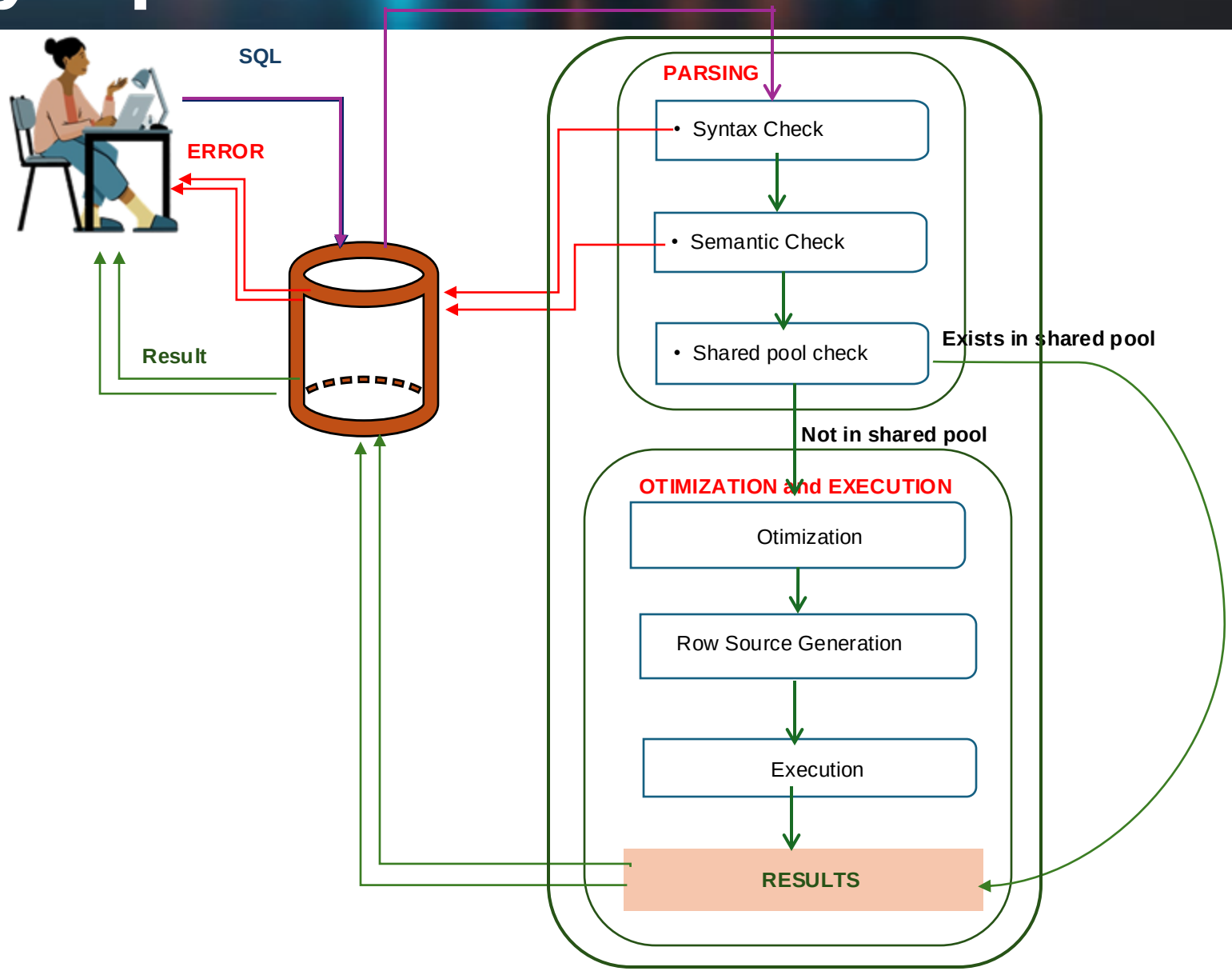
SQL engine executes each row source in the ***tree produced by the row source gener****ator.*



- *the* ***black boxes*** *physically retrieve data from an object in the database. These steps are the access paths, or techniques for retrieving data from the database.*
- *the* ***clear boxes*** *operate on row sources.*

This step is the only mandatory step in DML processing.

SQL Query Optimization

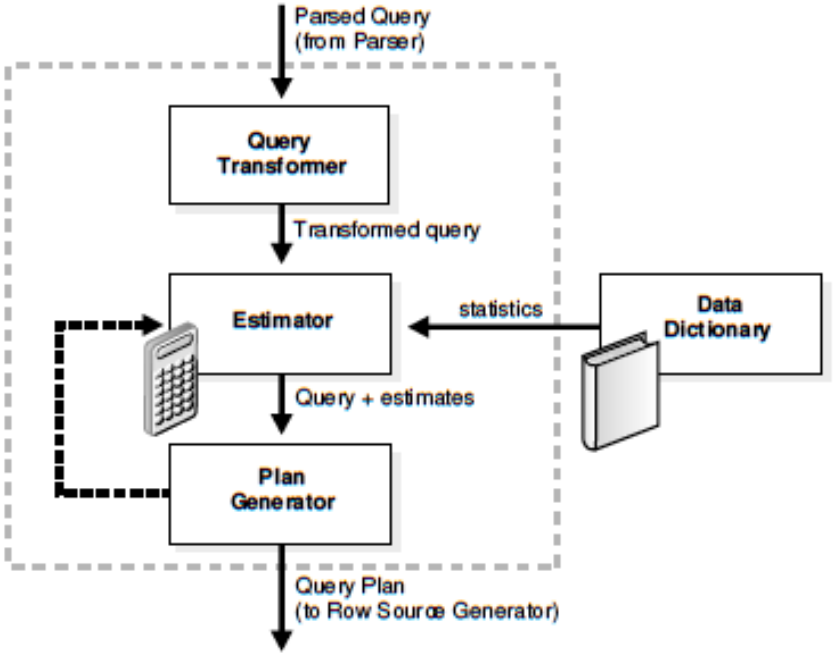


Optimizer Components

Query Transformer - determines whether it is helpful to change the form of the query so that the optimizer can generate a better execution plan.

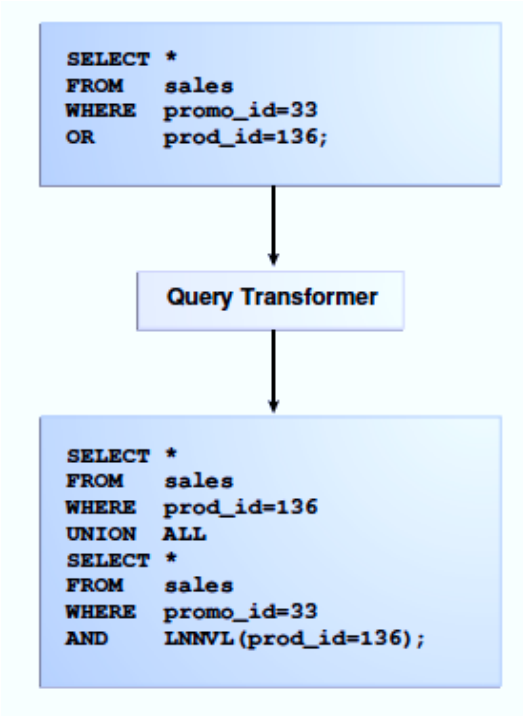
Estimator - estimates the cost of each plan based on statistics in the data dictionary.

Plan Generator - compares the costs of plans and chooses the lowest-cost plan, known as the execution plan, to pass to the **row source generator**



Query Transformer

- Determines whether it is advantageous to rewrite the original SQL statement into a semantically equivalent SQL statement with a lower cost.
  - When a viable alternative exists, the database calculates the cost of the alternatives separately and chooses the lowest-cost alternative.



Query Transformations

Or Expansion

- the optimizer transforms a query block containing top-level disjunctions into the form of a UNION ALL query that contains two or more branches.
- The optimizer can choose OR expansion for various reasons:
  - **avoid** Cartesian products.
  - only if the cost of the transformed statement **is lower than** the cost of the original statement.
  - performance improves because the database can **filter data** using the **indexes instead of full table scans**

View Merging

- **Each view referenced in a query** is **expanded** by the parser **into a separate query block**
- One option for **the optimizer is to analyze** the **view query block separately and generate a view subplan.**
  - This technique usually **leads** to a **suboptimal query plan**, because the **view is optimized separately from rest of the query.**
    - The **query transformer** then **removes the potentially suboptimal plan** by **merging** the **view query block into the query block** that contains the view.

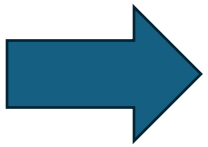
The **view merging optimization applies** to **simple views that contain only selections, projections, and joins.**

View Merging

```
CREATE or replace VIEW emp_30_vw AS
SELECT empno, ename, job, sal, deptno
FROM emp
WHERE deptno = 30;
```

Then execute a query on the view as below

```
SELECT empno
FROM emp_30_vw
WHERE empno > 150;
```



Oracle's statement transformer rewrites the query as below

```
SELECT empno, ename, job, sal, deptno
FROM emp
WHERE deptno = 30
AND empno >150;
```

Predicate Pushing

- The optimizer "pushes" the relevant predicates from the containing query block into the view query block.
  - This improve the subplan of the unmerged view because the database can use the pushed-in predicates to access indexes or to use as filters.

```
CREATE VIEW all_employees_vw AS
  ( SELECT employee_id, last_name, job_id, commission_pct,
department_id
    FROM   employees )
  UNION
  ( SELECT employee_id, last_name, job_id, commission_pct,
department_id
    FROM   contract_workers );
```

```
SELECT last_name
FROM   all_employees_vw
WHERE  department_id = 50;
```

```
SELECT last_name
FROM   ( SELECT employee_id, last_name, job_id, commission_pct,
department_id
         FROM   employees
         WHERE  department_id=50
         UNION
         SELECT employee_id, last_name, job_id, commission_pct,
department_id
         FROM   contract_workers
         WHERE  department_id=50 );
```

The transformed query can now consider index access in each of the query blocks.

Subquery Un-nesting

- In **subquery unnesting**, the optimizer transforms a nested query into an equivalent **JOIN statement**, and then optimizes the join.
- The optimizer can **perform** this transformation **only**
  - if the **resulting join** statement **is guaranteed to return the same rows** as the **original statement**, and
  - if subqueries do not contain aggregate functions such as AVG.

```
SELECT *
FROM   sales
WHERE  cust_id IN ( SELECT cust_id
                    FROM   customers );
```

```
SELECT sales.*
FROM   sales, customers
WHERE  sales.cust_id = customers.cust_id;
```

Subquery Un-nesting

- If the **optimizer cannot transform a complex statement** into **a join statement,**

1. **Selects execution plans for the parent statement** and **the subquery** as though they were **separate statements.**
2. Then **executes the subquery** and **uses the rows returned to execute the parent query.**
3. Orders the subplans efficiently - to improve execution speed of the overall execution plan

Queries involving a nested subquery connected by **IN** or **ANY** connector **can always be converted into a single block query**

Examples

```
SELECT DISTINCT snr
FROM Employee
WHERE deptname='ISEP'
```

"DISTINCT" may force a sort operation.

- If snr is unique

  Then **DISTINCT CAN BE OMITED**

```
SELECT snr
FROM Employee
WHERE deptid IN
        (SELECT deptid FROM Department)
```

Better

```
SELECT snr
FROM Employee E, Department D
WHERE e.depid=D.Depiod
```

Examples

```
SELECT E.sid
FROM Employee E
Where salary = (SELECT max(salary)
                FROM Employee E1
                Where E.depis=E1.depid)
```

- The subquery may be executed for each employe or at least each department

- Solution

```
Create view maximo_dep
SELECT max(salary)  m
FROM Employee
Group by  depid
```

```
SELECT E.sid
FROM Employee E , máximo_dep  MD
Where salary =  m AND E.depid = MD.depid
```

Examples

correlated sub-queries without aggregation

```
SELECT OrderNo FROM Order O
  WHERE ProdNo IN
   (SELECT ProdNo FROM Project P
    WHERE P.ProjNo = O.OrderNo
      AND P.Budget > 100,000)
```

```
SELECT OrderNo
  FROM Order O, Project P
  WHERE O.ProdNo = P.ProdNo
    AND P.ProjNo = O.OrderNo
    AND P.Budget > 100,000
```

correlated sub-queries with aggregation

```
SELECT OrderNo FROM Order O
  WHERE ProdNo IN
   (SELECT MAX(ProdNo)
     FROM Project P
     WHERE P.ProjNo = O.OrderNo
       AND P.Budget > 100,000)
```

```
SELECT OrderNo FROM Order O
  WHERE ProdNo IN
   (SELECT ProdNo FROM
     (SELECT ProjNo, MAX(ProdNo)
       FROM Project
       WHERE Budget > 100.000
       GROUP BY ProjNo) P
    WHERE P.ProjNo = O.OrderNo)
```

Query Rewrite with Materialized Views

- **A** materialized view is a query result **stored in a table**.
- When the optimizer finds a user **query compatible with the query associated with a materialized view**, the database can rewrite the query in terms of the materialized view.
- The optimizer does not rewrite the query when the plan generated, unless the query has a lower cost than the plan generated with the materialized views.
- **The optimizer uses two different methods** to determine when to rewrite a query in terms of a materialized view.

The first method

matches the SQL text of the query to the SQL text of the materialized view definition

The second is the more general method

which it **compares** joins, selections, data columns, grouping columns, and aggregate functions **between the query and materialized views.**

Query Rewrite with Materialized Views

- Query rewrite operates on queries and subqueries in the following types of SQL statements:
  - SELECT
  - CREATE TABLE … AS SELECT
  - INSERT INTO … SELECT
- It also operates on subqueries in the set operators UNION, UNION ALL, INTERSECT, MINUS and subqueries in DML statements such as INSERT, DELETE, and UPDATE.
- Dimensions, constraints, and rewrite integrity levels affect whether a query is rewritten to use materialized views.

Example of Query Rewrite

- Consider the materialized view **cal_month_sales_mv**, which provides an aggregation of the dollar amount sold in every month.

```
CREATE MATERIALIZED VIEW cal_month_sales_mv
ENABLE QUERY REWRITE AS
SELECT t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

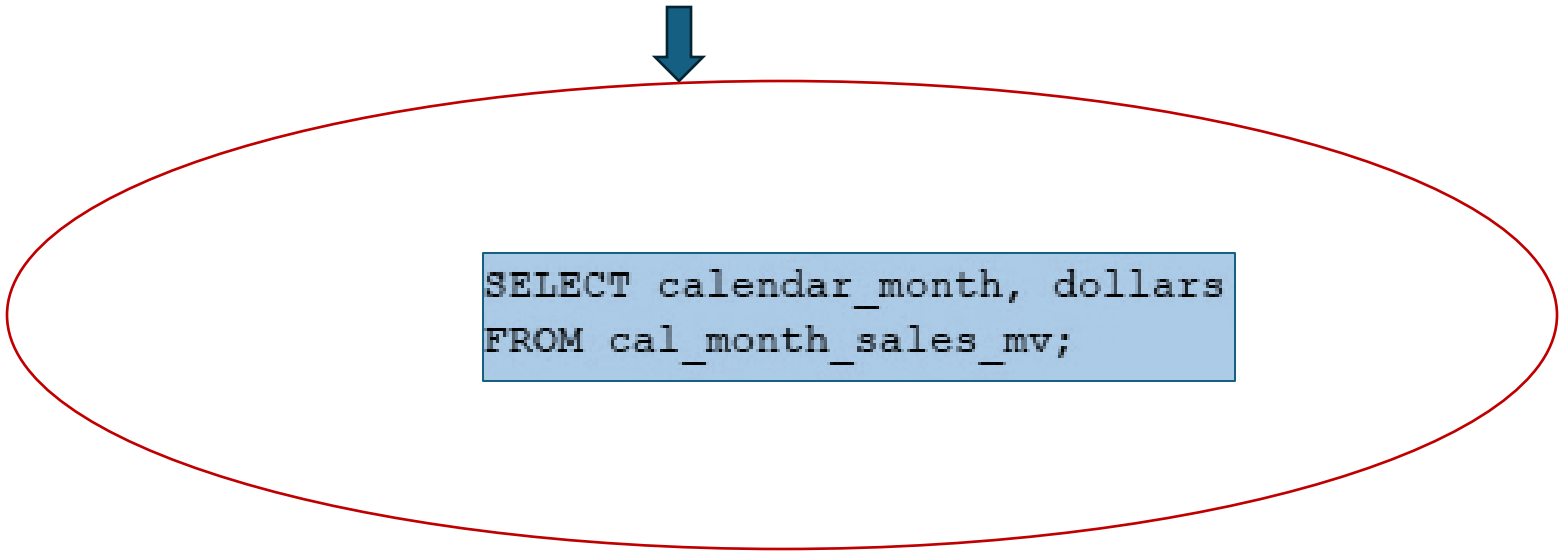
- **Let us assume that**, in a typical month, the number of sales in the store is around one million.
- Consider the following query, which asks for the sum of the amount sold at the store for each calendar month:

```
SELECT t.calendar_month_desc, SUM(s.amount_sold)
FROM sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

Example of Query Rewrite

- In the presence of the materialized view cal_month_sales_mv, query rewrite will transparently rewrite the previous query into the following query:

```
SELECT t.calendar_month_desc, SUM(s.amount_sold)
FROM sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```



```
SELECT calendar_month, dollars
FROM cal_month_sales_mv;
```

Materialized Views (Overview)

- A materialized view is a database object that stores the results of a query as a physical table.
  - It improves response times
  - Usually, they are associated to queries with aggregations
  - They may be used also for non aggregating queries
- Materialized views can be used as a **table** in any query
- Materialized views **can be automatically used by the DBMS without user intervention**
- Materialized views help answering queries very similar to the query which created them

Materialized Views (Overview)

- the purpose of the materialized view is to increase query execution performance
- materialized view consumes storage space and must be updated when the underlying detail tables are modified
    - Insert, update, or delete is not allowed in materialized views.
- key features of the materialized view
- physically exists in the database as a new table created from the original selection. Therefore, it represents a copy of the original data.
    - It is worth mentioning that if the data in the original table changes in any way, the content of the materialized view is not automatically updated.

Materialized Views Management

Syntax. - Oracle

```
-- Normal
CREATE MATERIALIZED VIEW view-name
BUILD [IMMEDIATE | DEFERRED]
REFRESH [FAST | COMPLETE | FORCE ]
ON [COMMIT | DEMAND ]
[[ENABLE | DISABLE] QUERY REWRITE]
AS
SELECT ...;
```

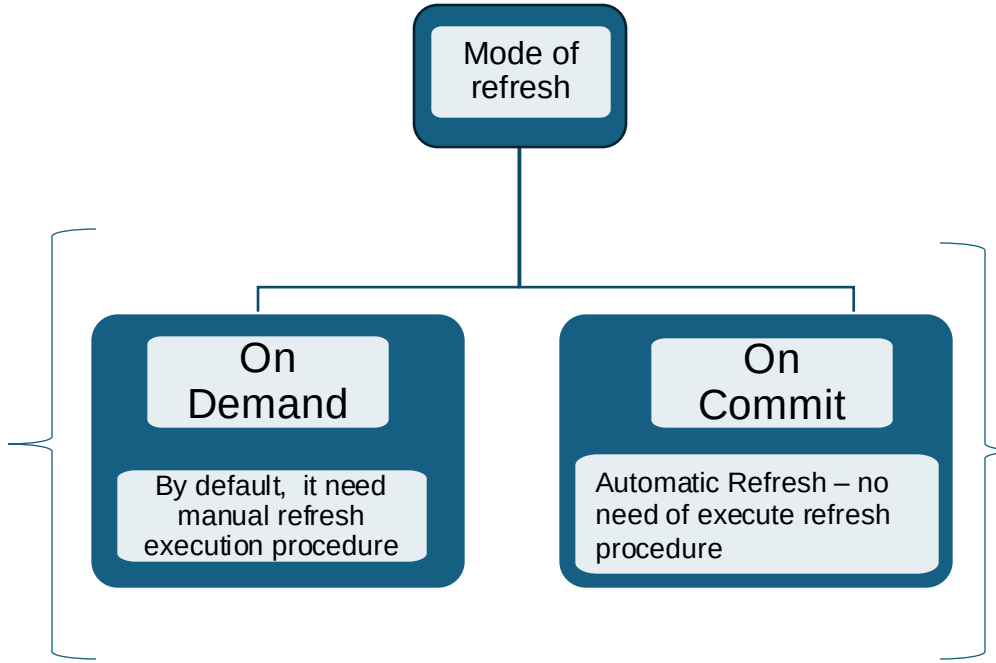
```
CREATE MATERIALIZED VIEW sales_mv
   BUILD IMMEDIATE
   REFRESH FAST ON COMMIT
   AS SELECT t.calendar_year, p.prod_id,
      SUM(s.amount_sold) AS sum_sales
      FROM times t, products p, sales s
      WHERE t.time_id = s.time_id AND p.prod_id = s.prod_id
      GROUP BY t.calendar_year, p.prod_id;
```

```
-- Normal
CREATE MATERIALIZED VIEW view-name
BUILD [IMMEDIATE | DEFERRED]
REFRESH [FAST | COMPLETE | FORCE ]
ON [COMMIT | DEMAND ]
[[ENABLE | DISABLE] QUERY REWRITE]
AS
SELECT ...;
```

- **BUILD** - when the materialized view is populated
  - IMMEDIATE: immediately
  - DEFERRED: First REFRESH
- **REFRESH -** How the view update is done
  - **COMPLETE**: completely updates the view by executing the SELECT command
  - **FAST:** only considers changes made (incremental update)
  - **FORCE** : does just that. It performs a FAST refresh, if possible, otherwise it performs a COMPLETE refresh.

Materialized Views

```
-- Normal
CREATE MATERIALIZED VIEW view-name
BUILD [IMMEDIATE | DEFERRED]
REFRESH [FAST | COMPLETE | FORCE ]
ON [COMMIT | DEMAND ]
[[ENABLE | DISABLE] QUERY REWRITE]
AS
SELECT ...;
```



create materialized view mv **REFRESH ON DEMAND** as select itemno,iname from item;

```
SQL> execute dbms_mview.refresh( 'MV' );
PL/SQL procedure successfully completed.
```

create materialized view **on Commit** as
select * from item;

It does not need “dbms_mview.refresh” to execute;

Synchronized with base table data just bi commit statement.

Materialized Views

- To delete a materialized view we use "drop materialized view":

  drop materialized view “view_name”;

- Update a materialized view we use:

```
BEGIN
DBMS_MVIEW.REFRESH(‘view_name’’);
END;
 /
```

Materialized Views

Exemplification in Oracle:

Scrip: **vista_materializada.sql**