Query Optimization

Query optimization is the overall process of choosing the most efficient means of executing a SQL statement.

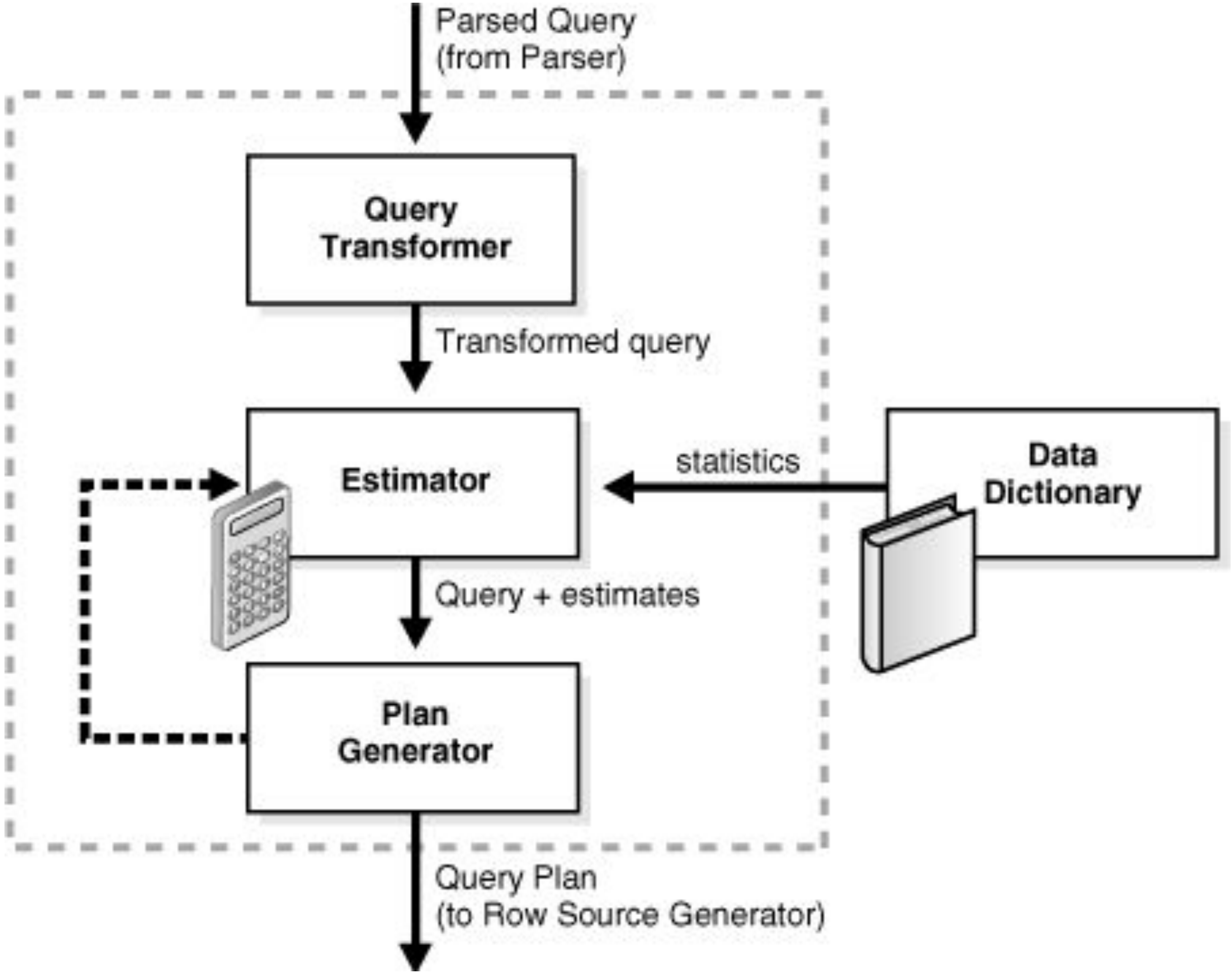
SQL is a nonprocedural language, so the optimizer is free to merge, reorganize, and process in any order.

The database optimizes each SQL statement based on statistics collected about the accessed data.

The optimizer determines the optimal plan for a SQL statement by examining multiple access methods, such as **full table scan** or **index scans**, **different join methods** such as **nested loops** and **hash joins**, **different join orders**, and **possible transformations.**

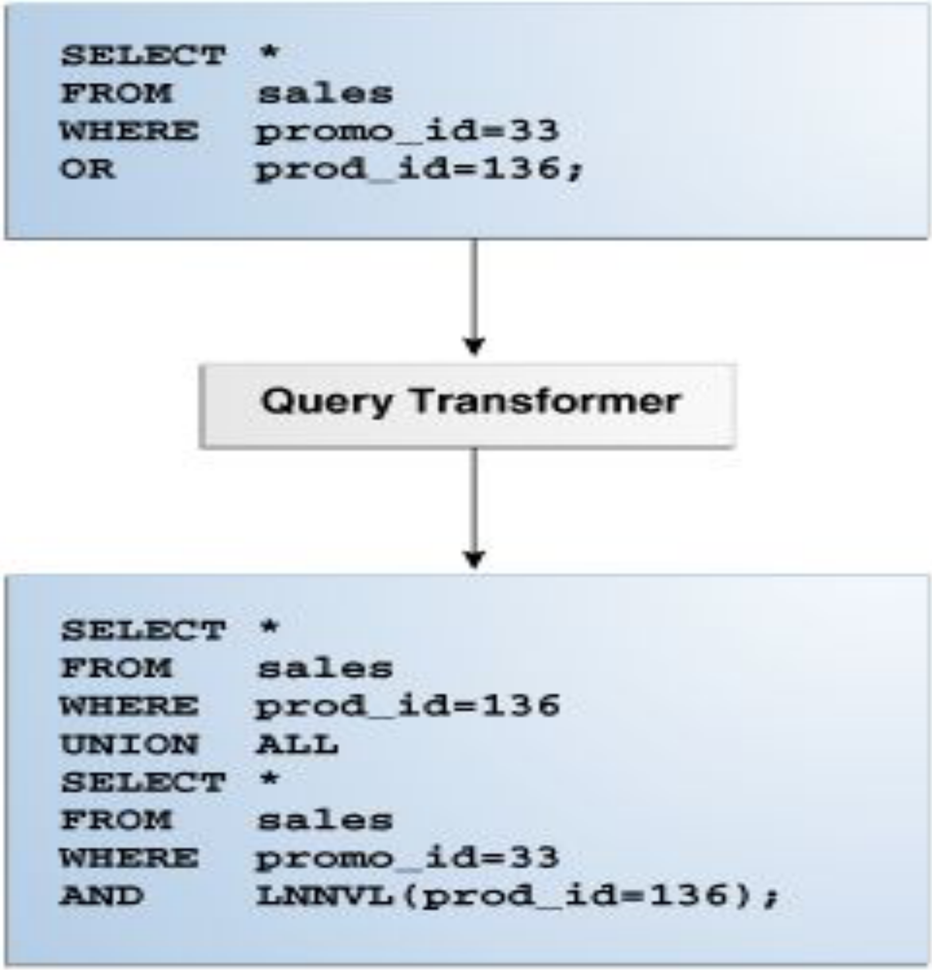
- For a given query and environment, the optimizer assigns a relative numerical cost to each step of a possible plan, and then factors these values together to generate an overall cost estimate for the plan. After calculating the costs of alternative plans, the optimizer chooses the plan with the lowest cost estimate. For this reason, the optimizer is sometimes called the **cost-based optimizer (CBO)**

The optimizer contains three components: the transformer, estimator, and plan generator.



Phase	Operation	Description
1	Query Transformer	The optimizer determines whether it is helpful to change the form of the query so that the optimizer can generate a better execution plan.
2	Estimator	The optimizer estimates the cost of each plan based on statistics in the data dictionary.
3	Plan Generator	The optimizer 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



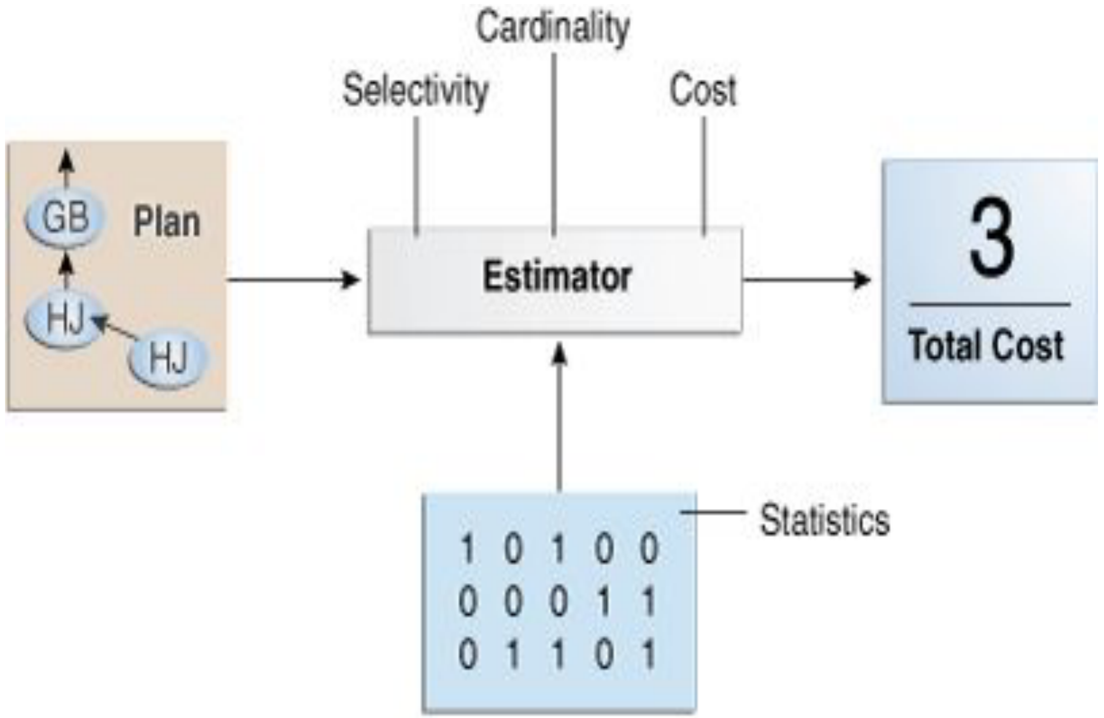
```
SELECT  *
FROM    sales
WHERE   promo_id=33
OR      prod_id=136;
```

```
SELECT  *
FROM    sales
WHERE   prod_id=136
UNION   ALL
SELECT  *
FROM    sales
WHERE   promo_id=33
AND     LNNVL(prod_id=136);
```

Estimator

The **estimator** is the component of the optimizer that determines the overall cost of a given execution plan.

The estimator uses three different measures to determine cost: Selectivity, Cardinality and Cost



Selectivity

The percentage of rows in the row set that the query selects, with 0 meaning no rows and 1 meaning all rows. Selectivity is tied to a query predicate, such as WHERE last_name LIKE 'A%', or a combination of predicates. A predicate becomes more selective as the selectivity value approaches 0 and less selective (or more unselective) as the value approaches 1.

Selectivity is an internal calculation that is not visible in the execution plans.

Cardinality

The cardinality is the number of rows returned by each operation in an execution plan. This input, which is crucial to obtaining an optimal plan, is common to all cost functions. The estimator can derive cardinality from the table statistics collected by DBMS_STATS, or derive it after accounting for effects from predicates (filter, join, and so on), DISTINCT or GROUP BY operations, and so on. The Rows column in an execution plan shows the estimated cardinality.

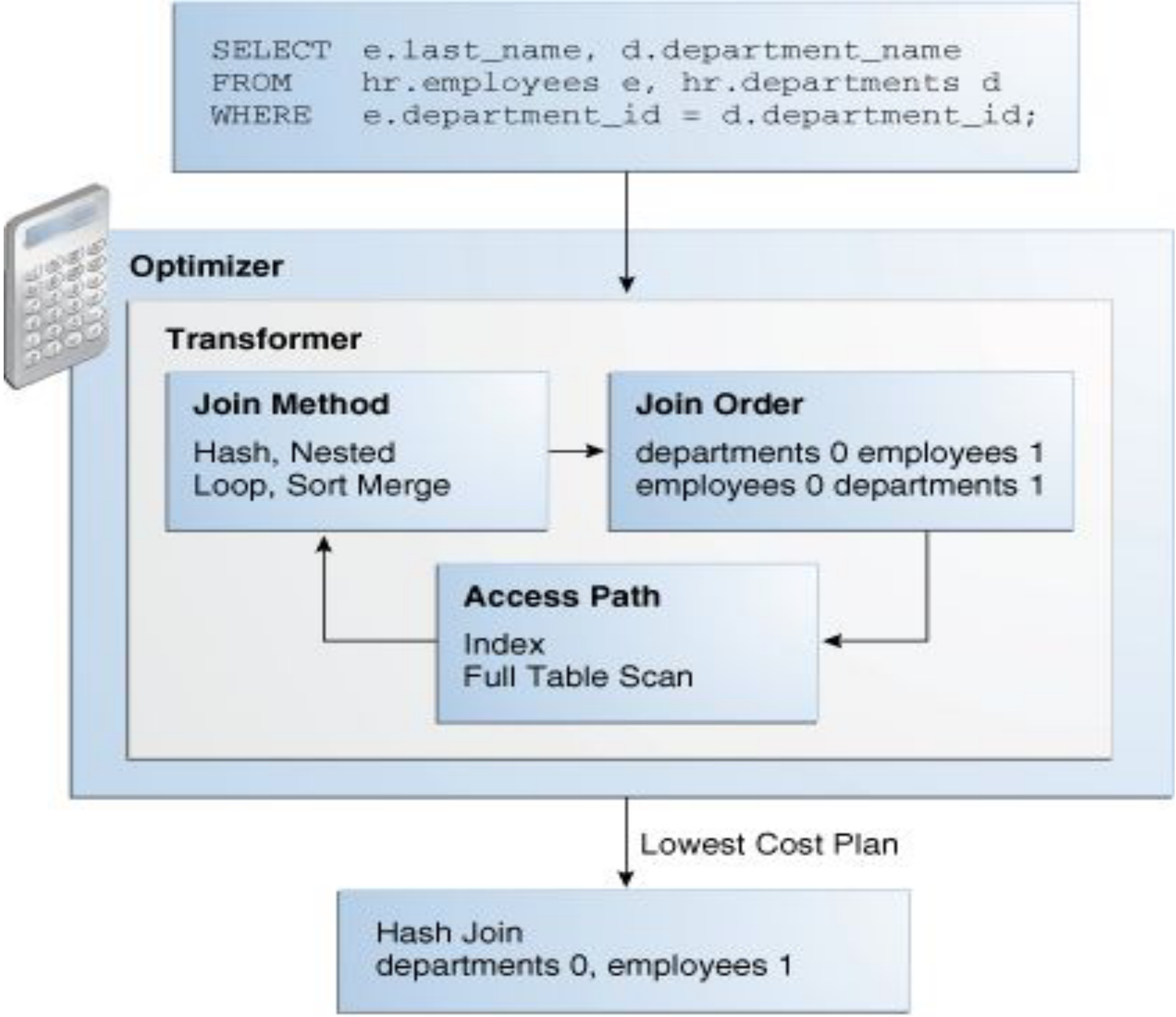
Cost

This measure represents units of work or resource used. The query optimizer uses disk I/O, CPU usage, and memory usage as units of work.

Plan Generator

The **plan generator** explores various plans for a query block by trying out different access paths, join methods, and join orders.

Many plans are possible because of the various combinations that the database can use to produce the same result. The optimizer picks the plan with the lowest cost.



```
SELECT  e.last_name, d.department_name
FROM    hr.employees e, hr.departments d
WHERE   e.department_id = d.department_id;
```

Optimizer

Transformer

Join Method

Hash, Nested Loop, Sort Merge

Join Order

departments 0 employees 1
employees 0 departments 1

Access Path

Index
Full Table Scan

Lowest Cost Plan

Hash Join
departments 0, employees 1

Performance Tuning

SQL tuning is the process of improving SQL queries to accelerate your servers performance. It's general purpose is to reduce the amount of time it takes a user to receive a result after issuing a query, and to reduce the amount of resources used to process a query.

Methods to Optimize Query

1. Use Indexes

A straightforward, yet effective, method for query optimization is indexing. **Indexes** can be thought of as duplicate tables containing records that may be needed often. This way, the whole database won’t need to be searched for these queries. For example, if a system needed to query first and last names and ID numbers of employees based on their age

```
CREATE INDEX age_queries

ON Employees (last_name, first_name, id, age);
```

2. Use Inner Join instead of Where

To link data from two or more tables, the Where clause is used. However, Where uses a kind of join that joins each record of each table and then filters it for the result. This means that if there are 100 records in the Employee table and 100 in the Salary table, it will generate a table of 10,000 records and then check for the correct records for this query.

```
Select Employees.first_name, Employees.last_name, Employees.id
From Employees, Salary
Where Employees.id = Salary.id;
```

In contrast, if the Inner Join is used, the required records will be produced without anything in the middle.

```
Select Employees.first_name, Employees.last_name, Employees.id
From Employees Inner Join Salary
On Employees.id = Salary.id;
```

3. Use Limit for testing

Before running a query on a large database with other potential users, it is advised to use the limit operator to ensure that it actually works. The limit keyword returns a limited number of results from a query, as shown below.

```
Select Employees.first_name, Employees.last_name, Employees.id
From Employees Inner Join Salary
On Employees.id = Salary.id
Where Salary.amount > 4500
Limit 10;
```

This will return only ten names and ID numbers of Employees with salaries greater than 4,500.

4. Use Explain

When querying a large database, it might be smart to use the Explain operator before running the query. Explain displays what is called a Query Plan. The query plan shows the order of execution and the approximate time each step might take. Using this information, more time-expensive steps may be better optimized.

```
Explain
Select Employees.first_name, Employees.last_name, Employees.id
From Employees Inner Join Salary
On Employees.id = Salary.id
Where Salary.amount > 4500;
```

5. Be Specific with Select queries

When working with Select queries, it is smart to avoid the * operator, which returns all columns for the queried records unless otherwise required. Instead, only the required information should be extracted. For example:

```
Select *
From Employees;
```

This is inefficient if only the names were required.

```
Select first_name, last_name
From Employees;
```