SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
(Estd U/S(3) of UGC Act, 1956) Accredited with Grade "A" by NAAC

Unit II SCS1613- DISTRIBUTED DATABASE

UNIT 2: QUERIES AND OPTIMAZATION

1. Global Queries to Fragment Queries
2. Equivalence Transformations for Queries
3. Distributed Grouping and Aggregate Function Evaluation
4. Parametric Queries
5. Optimization of Access Strategies
6. Framework for Query Optimization
7. Join Queries
8. General Queries
9. Introduction to Distributed Transactions.

1. Distributed Query Processing

A distributed database query is processed in stages as follows:

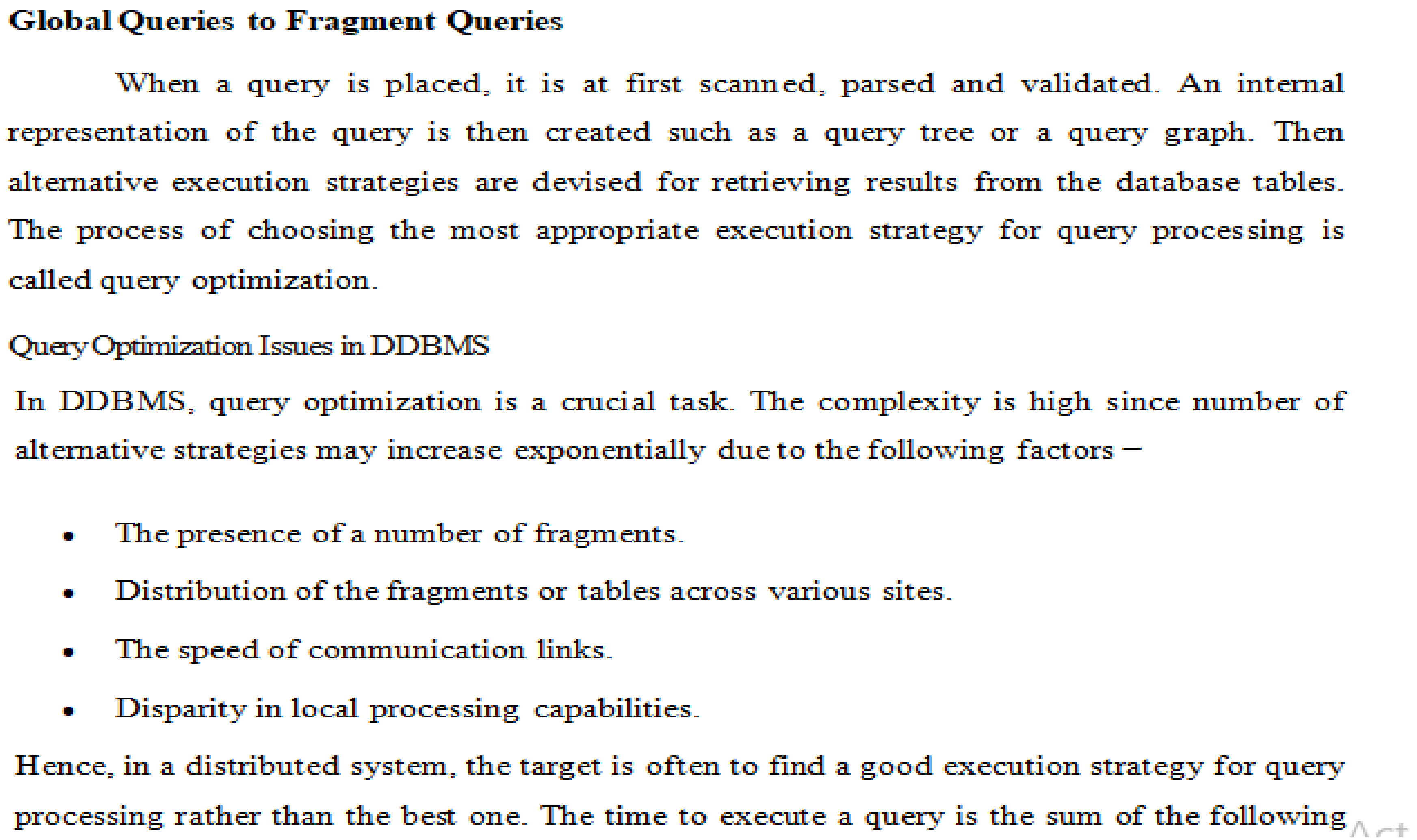
Query Mapping. The input query on distributed data is specified formally using a query language. It is then translated into an algebraic query on global relations. This translation is done by referring to the global conceptual schema and does not take into account the actual distribution and replica-tion of data. Hence, this translation is largely identical to the one performed in a centralized DBMS. It is first normalized, analyzed for semantic errors, simplified, and finally restructured into an algebraic query.

Localization. In a distributed database, fragmentation results in relations being stored in separate sites, with some fragments possibly being replicated. This stage maps the distributed query on the global schema to separate queries on individual fragments using data distribution and replication information.

Global Query Optimization. Optimization consists of selecting a strategy from a list of candidates that is closest to optimal. A list of candidate queries can be obtained by permuting the ordering of operations within a fragment query generated by the previous stage. Time is the preferred unit for measuring cost. The total cost is a weighted combination of costs such as CPU cost, I/O costs, and communication costs. Since DDBs are connected by a net-work, often the communication costs over the network are the most significant. This is especially true when the sites are connected through a wide area network (WAN).

Local Query Optimization. This stage is common to all sites in the DDB. The techniques are similar to those used in centralized systems.

1. Global Queries to Fragment Queries



Global Queries to Fragment Queries

When a query is placed, it is at first scanned, parsed and validated. An internal representation of the query is then created such as a query tree or a query graph. Then alternative execution strategies are devised for retrieving results from the database tables. The process of choosing the most appropriate execution strategy for query processing is called query optimization.

Query Optimization Issues in DDBMS

In DDBMS, query optimization is a crucial task. The complexity is high since number of alternative strategies may increase exponentially due to the following factors −

- The presence of a number of fragments.
- Distribution of the fragments or tables across various sites.
- The speed of communication links.
- Disparity in local processing capabilities.

Hence, in a distributed system, the target is often to find a good execution strategy for query processing rather than the best one. The time to execute a query is the sum of the following

1. Global Queries to Fragment Queries

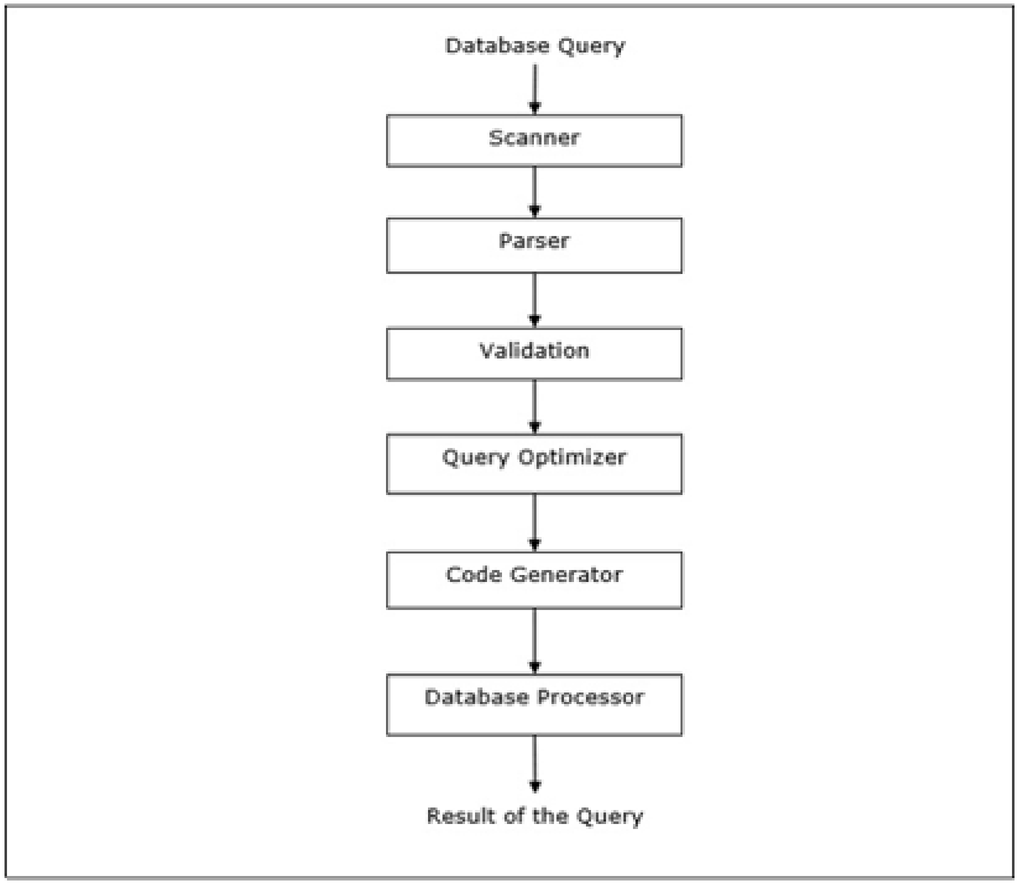
- The presence of a number of fragments.
- Distribution of the fragments or tables across various sites.
- The speed of communication links.
- Disparity in local processing capabilities.

Hence, in a distributed system, the target is often to find a good execution strategy for query processing rather than the best one. The time to execute a query is the sum of the following –

- Time to communicate queries to databases.
- Time to execute local query fragments.
- Time to assemble data from different sites.
- Time to display results to the application.

Query Processing

Query processing is a set of all activities starting from query placement to displaying the results of the query. The steps are as shown in the following diagram −



- Query processing in a distributed context is to transform a high-level query on a distributed database, which is seen as a single database by the users, into an efficient execution strategy expressed in a low-level language on local databases.

- The main function of a relational query processor is to transform a high-level query (typically, in relational calculus) into an equivalent lower-level query (typically, in some variation of relational algebra).

Query Processing Problem

- The main difficulty is to select the execution strategy that minimizes resource consumption.
- The low-level query actually implements the execution strategy for the query. The transformation must achieve both correctness and efficiency.
- It is correct if the low-level query has the same semantics as the original query, that is, if both queries produce the same result.
- The well-defined mapping from **relational calculus** to **relational algebra** makes the correctness issue easy.

Query Processing Example

- **Example:** Transformation of an SQL-query into an RA-query.
  Relations: EMP(ENO, ENAME, TITLE), ASG(ENO,PNO,RESP,DUR)
  Query: *Find the names of employees who are managing a project?*

– **High level query**

```
SELECT ENAME
FROM EMP,ASG
WHERE EMP.ENO = ASG.ENO AND DUR > 37
```

– **Two possible transformations of the query are:**

- Expression 1: ENAME(DUR>37∧EMP.ENO=ASG.ENO(EMP × ASG))
- Expression 2: ENAME(EMP ⋊⋉ENO (DUR>37(ASG)))

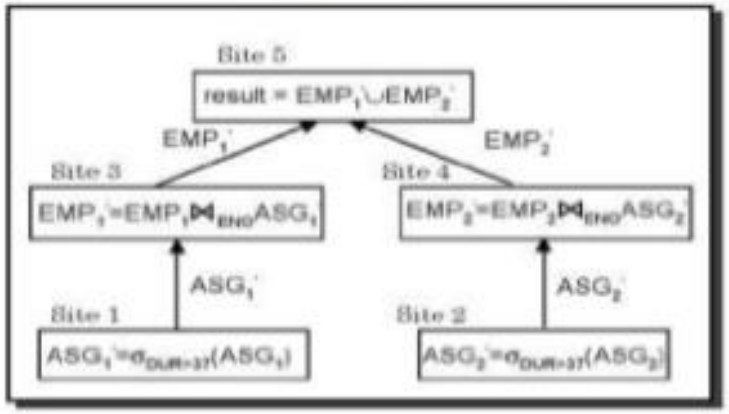
– Expression 2 avoids the expensive and large intermediate Cartesian product, and therefore typically is better.

Query Processing Example

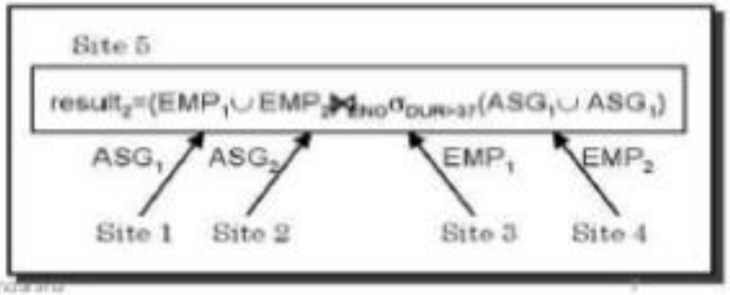
- We make the following assumptions about the data fragmentation
  - **Data is (horizontally) fragmented:**
- Site1: ASG1 = ENO≤"E3"(ASG)
- Site2: ASG2 = ENO>"E3"(ASG)
- Site3: EMP1 = ENO≤"E3"(EMP)
- Site4: EMP2 = ENO>"E3"(EMP)
- Site5: Result
- Relations ASG and EMP are fragmented in the same way
- Relations ASG and EMP are locally clustered on attributes RESP and ENO,
- respectively

Query Processing Example . . .

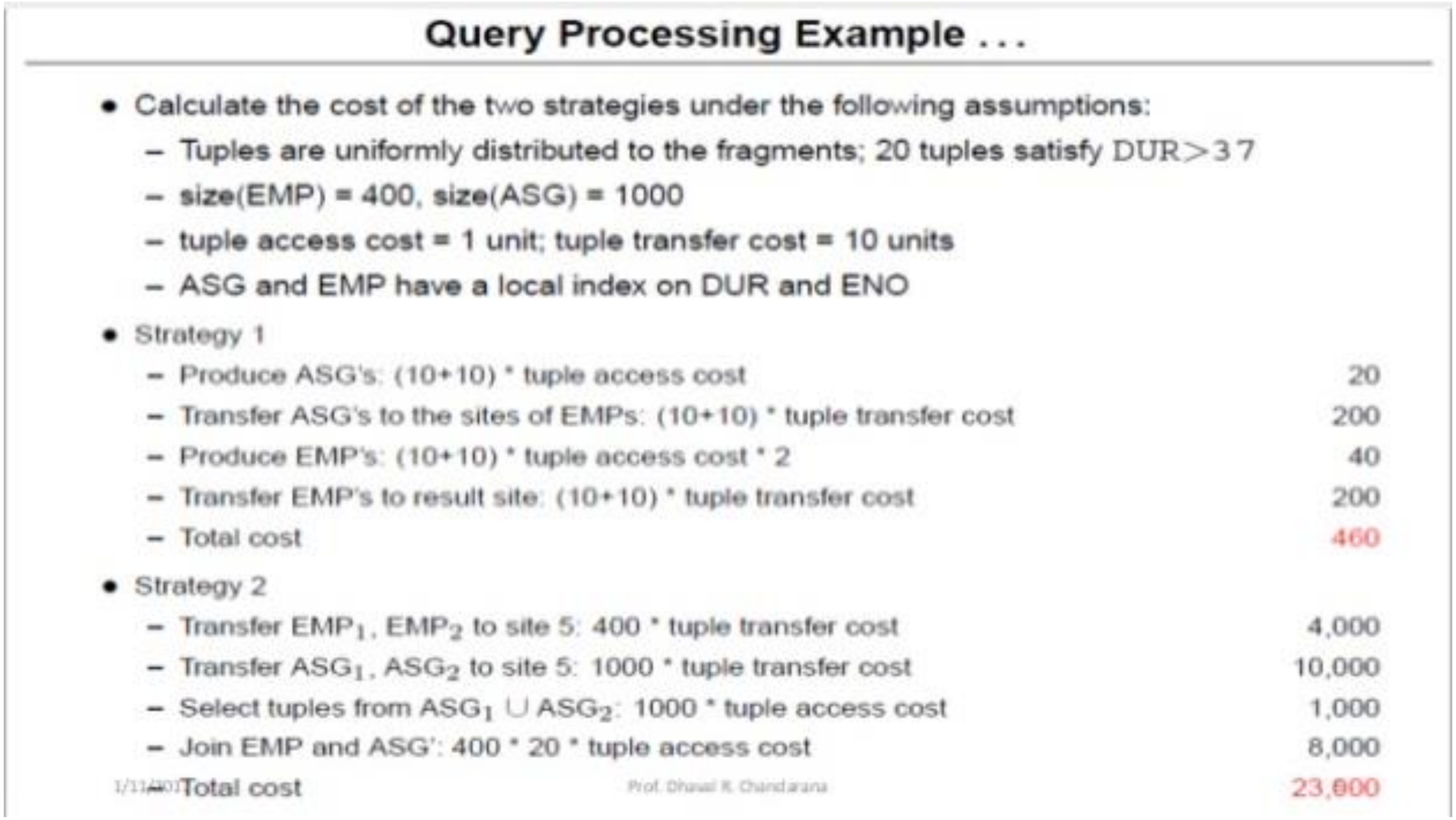
- Now consider the expression $\Pi_{ENAME}(EMP \ltimes_{ENO} (\sigma_{DUR>37}(ASG)))$
- Strategy 1 (partially parallel execution):
  - Produce ASG'_1 and move to Site 3
  - Produce ASG'_2 and move to Site 4
  - Join ASG'_1 with EMP_1 at Site 3 and move the result to Site 5
  - Join ASG'_2 with EMP_2 at Site 4 and move the result to Site 5
  - Union the result in Site 5



- Strategy 2:
  - Move ASG_1 and ASG_2 to Site 5
  - Move EMP_1 and EMP_2 to Site 5
  - Select and join at Site 5



- For simplicity, the final projection is omitted.



Query Processing Example . . .

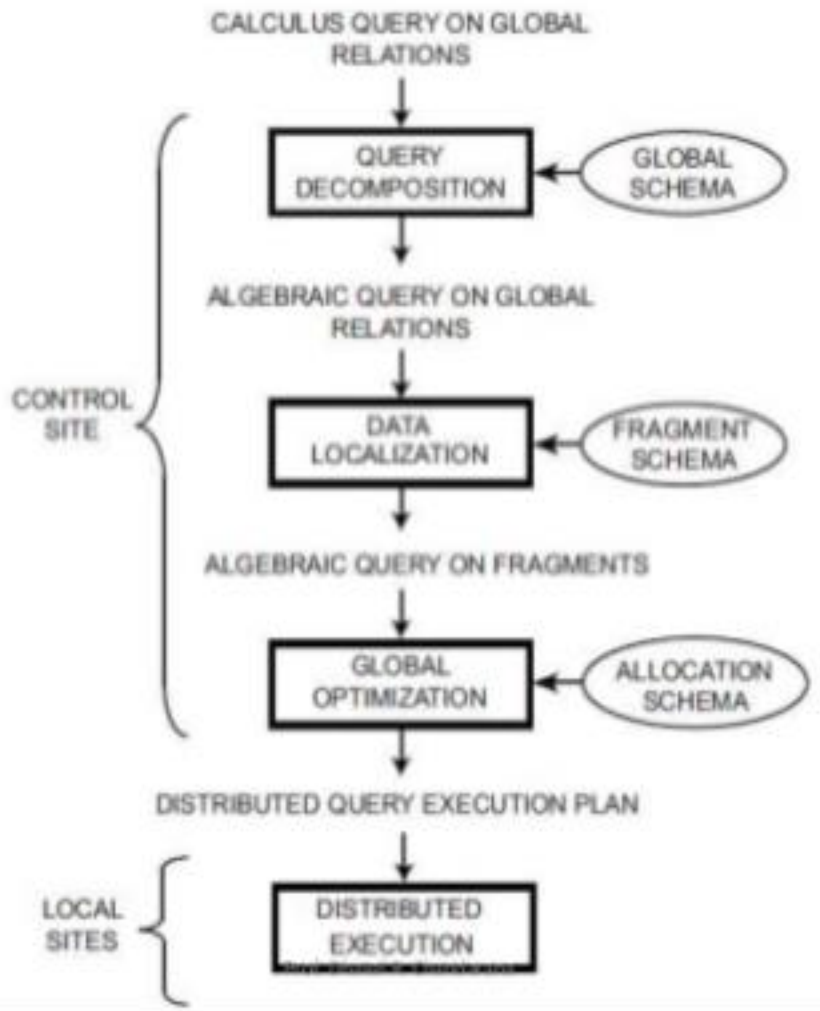
- Calculate the cost of the two strategies under the following assumptions:
  - Tuples are uniformly distributed to the fragments; 20 tuples satisfy DUR>37
  - size(EMP) = 400, size(ASG) = 1000
  - tuple access cost = 1 unit; tuple transfer cost = 10 units
  - ASG and EMP have a local index on DUR and ENO
- Strategy 1

– Produce ASG's: (10+10) * tuple access cost	20
– Transfer ASG's to the sites of EMPs: (10+10) * tuple transfer cost	200
– Produce EMP's: (10+10) * tuple access cost * 2	40
– Transfer EMP's to result site: (10+10) * tuple transfer cost	200
– Total cost	460

- Strategy 2

– Transfer EMP_1, EMP_2 to site 5: 400 * tuple transfer cost	4,000
– Transfer ASG_1, ASG_2 to site 5: 1000 * tuple transfer cost	10,000
– Select tuples from $ASG_1 \cup ASG_2$: 1000 * tuple access cost	1,000
– Join EMP and ASG': 400 * 20 * tuple access cost	8,000
– Total cost	23,000

Layer of Query Processing



Query Decomposition

- The first layer decomposes the calculus query into an algebraic query on global relations. The information needed for this transformation is found in the global conceptual schema describing the global relations.
- Query decomposition can be viewed as four successive steps.
- **First**, the calculus query is rewritten in a normalized form that is suitable for subsequent manipulation. Normalization of a query generally involves the manipulation of the query quantifiers and of the query qualification by applying logical operator priority.
- **Second**, the normalized query is analyzed semantically so that incorrect queries are detected and rejected as early as possible. Techniques to detect incorrect queries exist only for a subset of relational calculus. Typically, they use some sort of graph that captures the semantics of the query.

Query Decomposition

- **Third**, the correct query (still expressed in relational calculus) is simplified. One way to simplify a query is to eliminate redundant predicates. Note that redundant queries are likely to arise when a query is the result of system transformations applied to the user query. such transformations are used for performing semantic data control (views, protection, and semantic integrity control).
- **Fourth**, the calculus query is restructured as an algebraic query. The traditional way to do this transformation toward a "better" algebraic specification is to start with an initial algebraic query and transform it in order to find a "go
- The algebraic query generated by this layer is good in the sense that the worse executions are typically avoided.

Data Localization

- The input to the second layer is an algebraic query on global relations. The main role of the second layer is to localize the query's data using data distribution information in the fragment schema.
- This layer determines which fragments are involved in the query and transforms the distributed query into a query on fragments.
- A global relation can be reconstructed by applying the fragmentation rules, and then deriving a program, called a localization program, of relational algebra operators, which then act on fragments.
- Generating a query on fragments is done in two steps
  - **First**, the query is mapped into a fragment query by substituting each relation by its reconstruction program (also called materialization program).
  - **Second**, the fragment query is simplified and restructured to produce another "good" query.

Global Query Optimization

- The input to the third layer is an algebraic query on fragments. The goal of query optimization is to find an execution strategy for the query which is close to optimal.
- The previous layers have already optimized the query, for example, by eliminating redundant expressions. However, this optimization is independent of fragment characteristics such as fragment allocation and cardinalities.
- **Query optimization consists of finding the "best" ordering of operators in the query, including communication operators that minimize a cost function.**
- The output of the query optimization layer is a optimized algebraic query with communication operators included on fragments. It is typically represented and saved (for future executions) as a distributed query execution plan .

Distributed Query Execution

- The last layer is performed by all the sites having fragments involved in the query.
- Each sub query executing at one site, called a local query, is then optimized using the local schema of the site and executed.

Characterization of Query Processors

- The input language to the query processor can be based on relational calculus or relational algebra.
- Query optimization is to select a best point of solution space that leads to the minimum cost.
- Optimization can be done statically before executing the query or dynamically as the query is executed.
- Dynamic query optimization requires statistics in order to choose the operation that has to be done first.
- Static query optimization requires statistics to estimate the size of intermediate relations.
- Distributed query processor exploits the network topology.

Query Processing in Centralized Systems

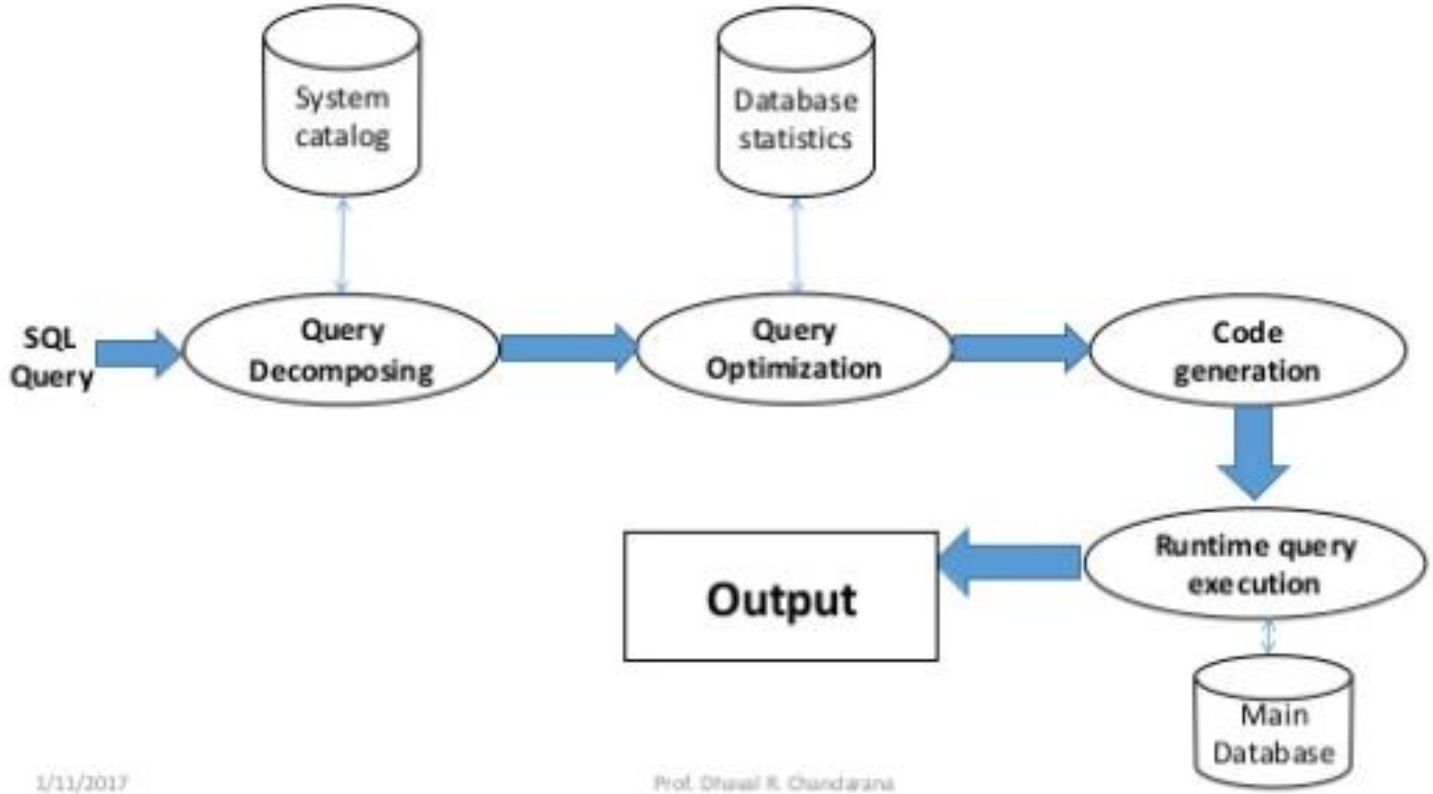
- **Goal of query processor in centralized system is:**

1. Minimize the query response time.
2. Maximize the parallelism in the system
3. Maximize the system throughput.

- In centralized DBMS query processing consists of four steps:

1. Query decomposition
2. Query optimization
3. Code generation
4. Query execution

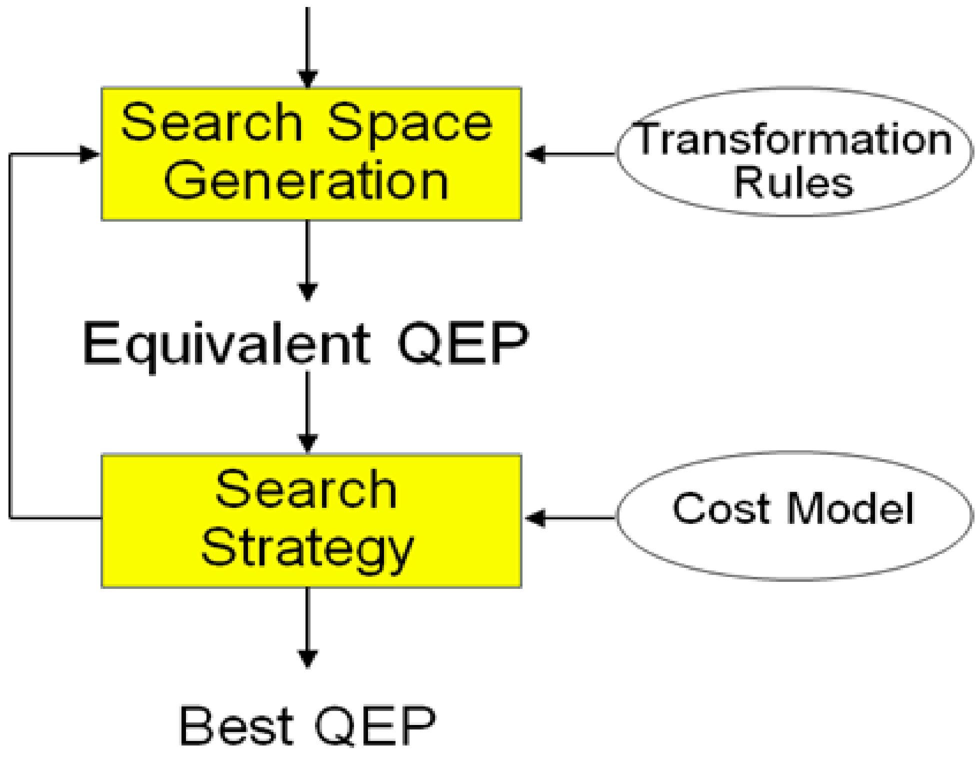
Query Processing in Centralized Systems



Cost-Based Optimization

- Solution space
  - ➡ The set of equivalent algebra expressions (query trees).
- Cost function (in terms of time)
  - ➡ I/O cost + CPU cost + communication cost
  - ➡ These might have different weights in different distributed environments (LAN vs WAN).
  - ➡ Can also maximize throughput
- Search algorithm
  - ➡ How do we move inside the solution space?
  - ➡ Exhaustive search, heuristic algorithms (iterative improvement, simulated annealing, genetic,…)

Query Optimization Process



Search Space

- Search space characterized by alternative execution
- Focus on join trees
- For *N* relations, there are O(*N*!) equivalent join trees that can be obtained by applying commutativity and associativity rules

SELECT ENAME,RESP

FROM EMP, ASG,PROJ

WHERE EMP.ENO=ASG.ENO

AND ASG.PNO=PROJ.PNO

Cost Functions

- Total Time (or Total Cost)
  - ➡ Reduce each cost (in terms of time) component individually
  - ➡ Do as little of each cost component as possible
  - ➡ Optimizes the utilization of the resources

Increases system throughput

- Response Time
  - ➡ Do as many things as possible in parallel
  - ➡ May increase total time because of increased total activity
- Summation of all cost factors
- Total cost = CPU cost + I/O cost + communication cost
- CPU cost = unit instruction cost * no.of instructions
- I/O cost = unit disk I/O cost * no. of disk I/Os
- communication cost = message initiation + transmission

2-Step – Problem Definition

- Given
  - ➡ A set of sites $S = \{s_1, s_2, \ldots, s_n\}$ with the load of each site
  - ➡ A query $Q = \{q_1, q_2, q_3, q_4\}$ such that each subquery q_i is the maximum processing unit that accesses one relation and communicates with its neighboring queries
  - ➡ For each q_i in Q, a feasible allocation set of sites $S_q = \{s_1, s_2, \ldots, s_k\}$ where each site stores a copy of the relation in q_i
- The objective is to find an optimal allocation of Q to S such that
  - ➡ the load unbalance of S is minimized
  - ➡ The total communication cost is minimized
- For each q in Q compute load (S_q)
- While Q not empty do
  - ➡ Select sub query a with least allocation flexibility

A
Go

- ➡ Select best site b for a (with least load and best benefit)
- ➡ Remove a from Q and recompute loads if needed

2-Step Algorithm Example

- Let $Q = \{q_1, q_2, q_3, q_4\}$ where q_1 is associated with R_1, q_2 is associated with R_2 joined with the result of q_1, etc.
- Iteration 1: select q_4, allocate to s_1, set load(s_1)=2
- Iteration 2: select q_2, allocate to s_2, set load(s_2)=3
- Iteration 3: select q_3, allocate to s_1, set load(s_1) =3
- Iteration 4: select q_1, allocate to s_3 or s_4

Relational Algebra :

The Relational Algebra is used to define the ways in which relations (tables) can be operated to manipulate their data.

□ This Algebra is composed of Unary operations (involving a single table) and Binary operations (involving multiple tables).

□ Join, Semi-join these are Binary operations in Relational Algebra.

Join

- Join is a binary operation in Relational Algebra.
- It combines records from two or more tables in a database.
- A join is a means for combining fields from two tables by using values common to each.

Semi-Join

•A Join where the result only contains the columns from one of the joined tables.

•Useful in distributed databases, so we don't have to send as much data over the network.

•Can dramatically speed up certain classes of queries.

What is "Semi-Join" ?

Semi-join strategies are technique for query processing in distributed database systems.

Used for reducing communication cost.

A semi-join between two tables returns rows from the first table where one or more matches are found in the second table.

The difference between a semi-join and a conventional join is that rows in the first table will be returned at most once. Even if the second table contains two matches for a row in the first table, only one copy of the row will be returned.

Semi-joins are written using EXISTS or IN.

A Simple Semi-Join Example "Give a list of departments with at least one employee." Query written with a conventional join:

```
SELECT D.deptno, D.dname FROM dept D, emp E WHERE E.deptno = D.deptno
ORDER BY D.deptno;
```

- A department with N employees will appear in the list N times.
- We could use a DISTINCT keyword to get each department to appear only once.

A Simple Semi-Join Example "Give a list of departments with at least one employee." Query written with a semi-join:

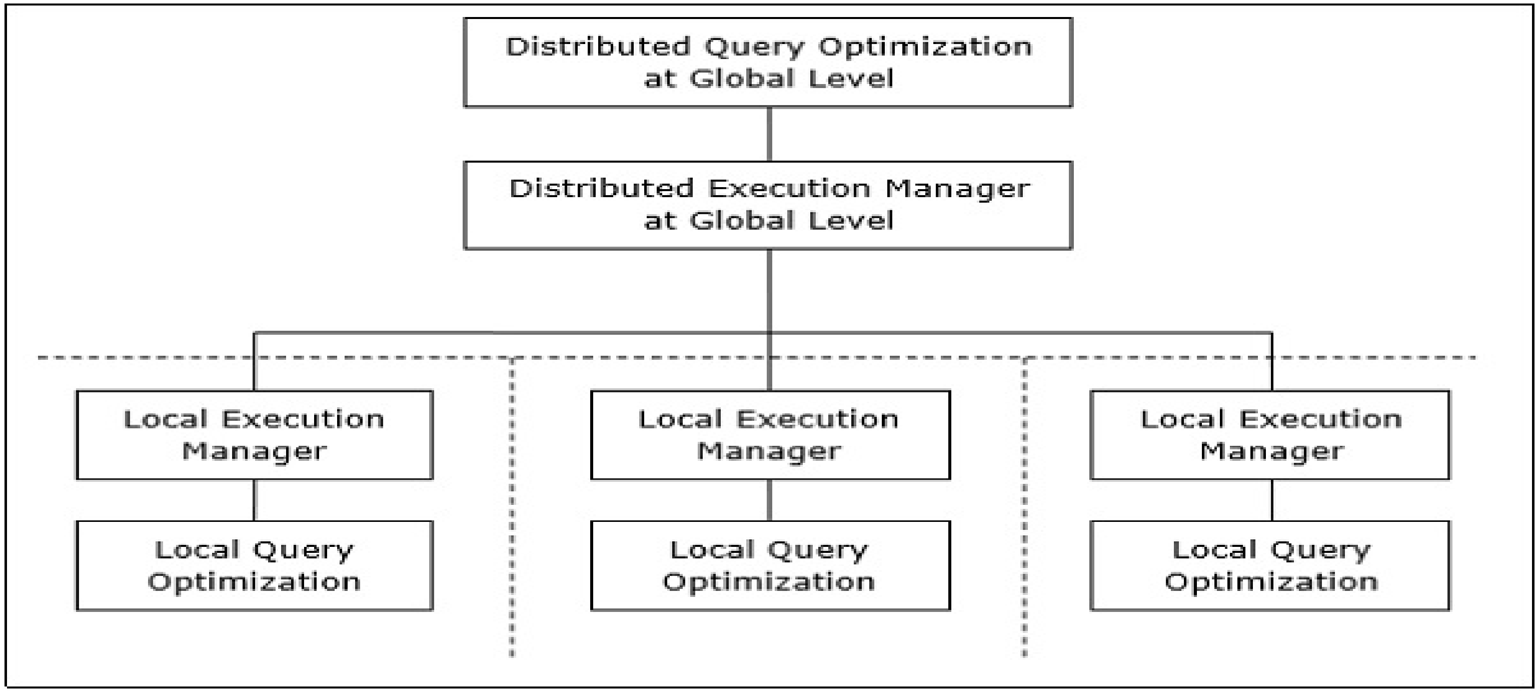
```
SELECT D.deptno, D.dname FROM dept D WHERE EXISTS (SELECT 1 FROM
emp E WHERE E.deptno = D.deptno) ORDER BY D.deptno;
```

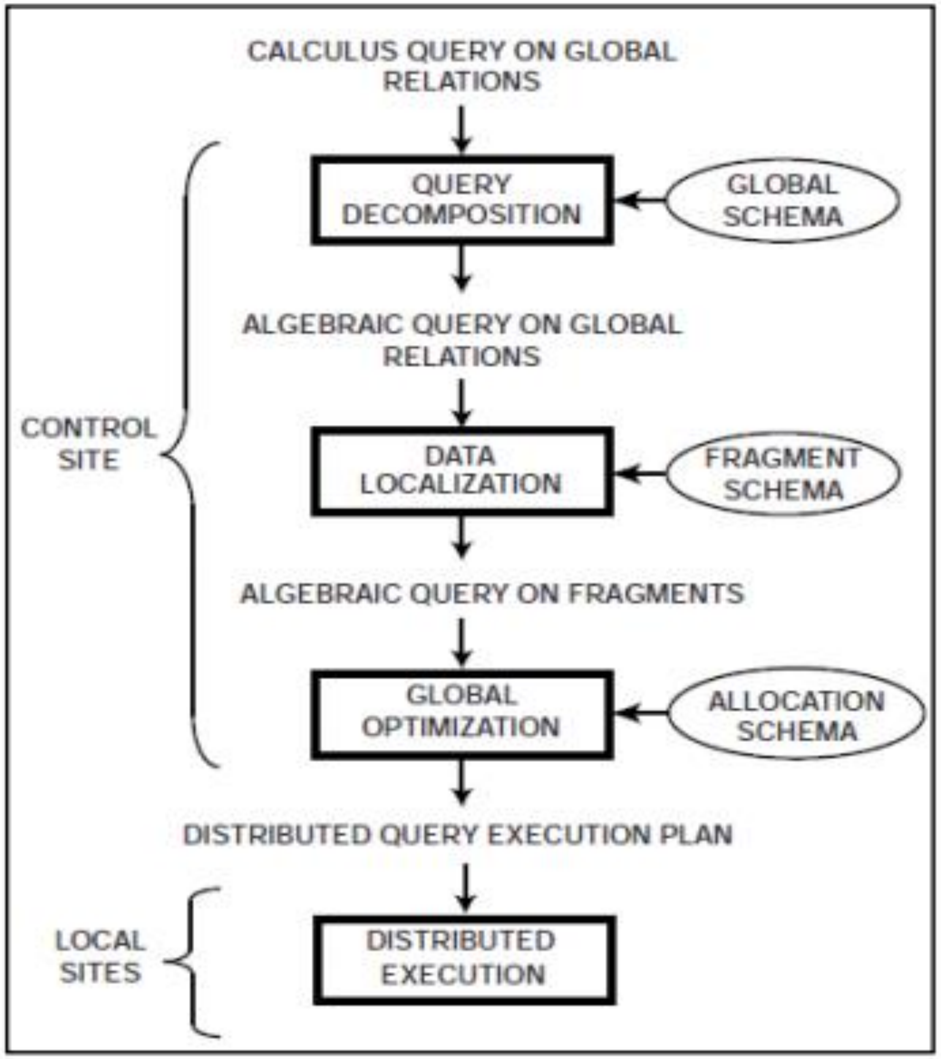
- No department appears more than once.
- Oracle stops processing each department as soon as the first employee in that department is found.

Distributed Query Processing Architecture

In a distributed database system, processing a query comprises of optimization at both the global and the local level. The query enters the database system at the client or controlling site. Here, the user is validated, the query is checked, translated, and optimized at a global level.

The architecture can be represented as –





Genetic layering scheme for distributed Query Processing

Mapping Global Queries into Local Queries

The process of mapping global queries to local ones can be realized as follows −

- The tables required in a global query have fragments distributed across multiple sites. The local databases have information only about local data. The controlling site uses the global data dictionary to gather information about the distribution and reconstructs the global view from the fragments.
- If there is no replication, the global optimizer runs local queries at the sites where the fragments are stored. If there is replication, the global optimizer selects the site based upon communication cost, workload, and server speed.
- The global optimizer generates a distributed execution plan so that least amount of data transfer occurs across the sites. The plan states the location of the fragments, order in which query steps needs to be executed and the processes involved in transferring intermediate results.
- The local queries are optimized by the local database servers. Finally, the local query results are merged together through union operation in case of horizontal fragments and join operation for vertical fragments.

For example, let us consider that the following Project schema is horizontally fragmented according to City, the cities being New Delhi, Kolkata and Hyderabad.

PROJECT

PId	City	Department	Status

Suppose there is a query to retrieve details of all projects whose status is "Ongoing".

The global query will be &inus;

$$\sigma_{status} = {\small "ongoing"}^{(PROJECT)}$$

Query in New Delhi's server will be −

$$\sigma_{status} = {\small "ongoing"}^{({NewD}_-{PROJECT})}$$

Query in Kolkata's server will be −

$$\sigma_{status} = {\small "ongoing"}^{({Kol}_-{PROJECT})}$$

Query in Hyderabad's server will be −

$$\sigma_{status} = {\small "ongoing"}^{({Hyd}_-{PROJECT})}$$

In order to get the overall result, we need to union the results of the three queries as follows −

$\sigma_{status} = {\small "ongoing"}^{({NewD}_-{PROJECT})} \cup \sigma_{status} = {\small "ongoing"}^{({kol}_-{PROJECT})} \cup \sigma_{status} = {\small "ongoing"}^{({Hyd}_-{PROJECT})}$

Distributed Query Optimization

Distributed query optimization requires evaluation of a large number of query trees each of which produce the required results of a query. This is primarily due to the presence of large amount of replicated and fragmented data. Hence, the target is to find an optimal solution instead of the best solution.

The main issues for distributed query optimization are −

- Optimal utilization of resources in the distributed system.
- Query trading.
- Reduction of solution space of the query.

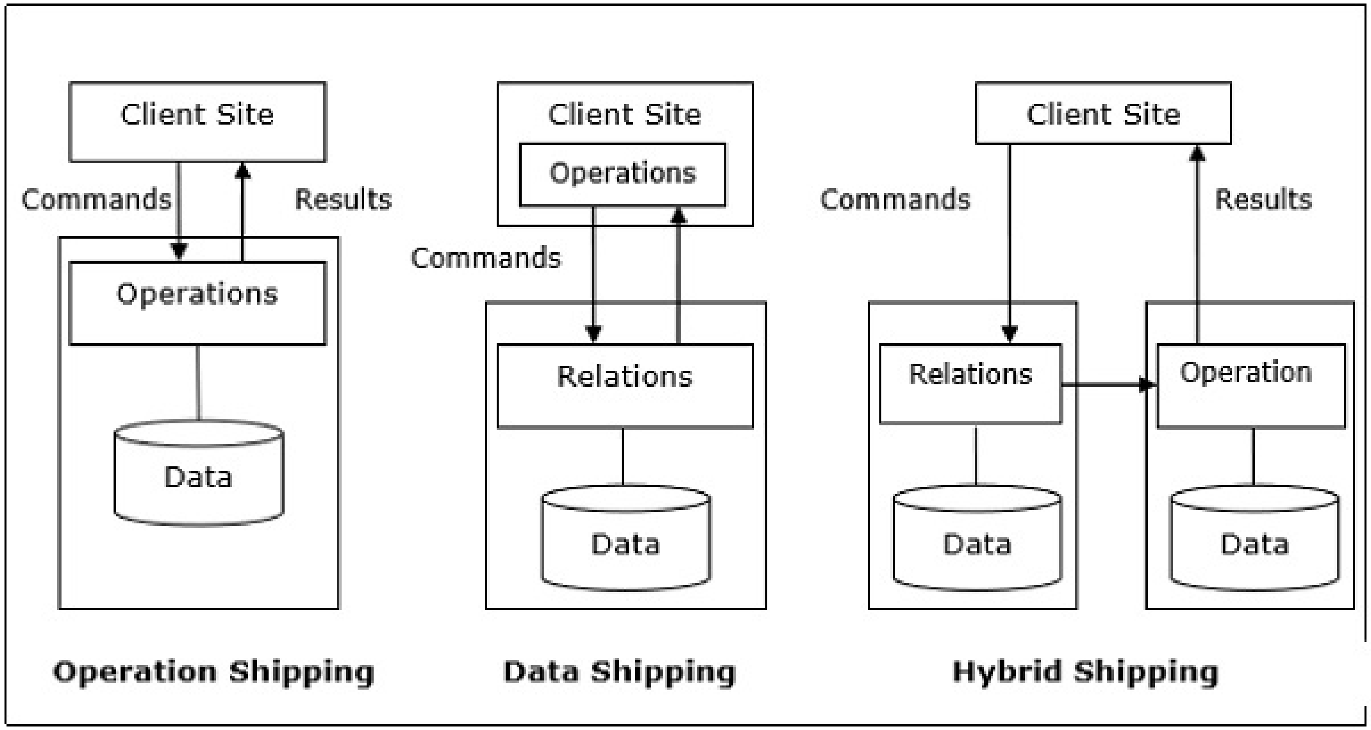
Optimal Utilization of Resources in the Distributed System

A distributed system has a number of database servers in the various sites to perform the operations pertaining to a query. Following are the approaches for optimal resource utilization −

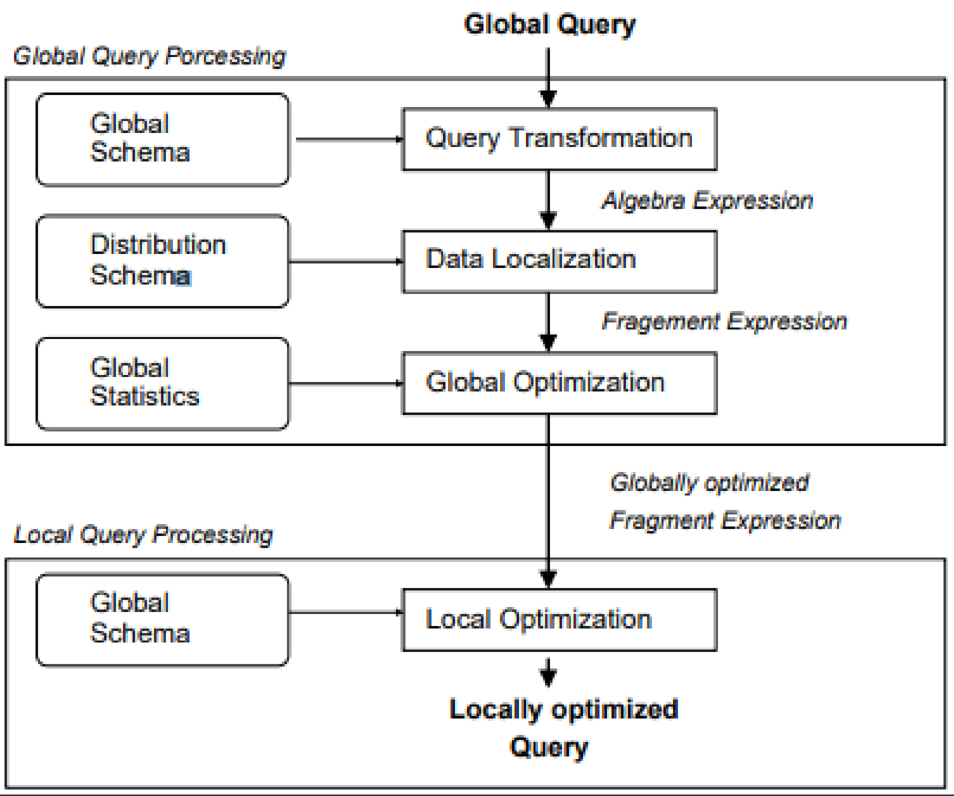
Operation Shipping − In operation shipping, the operation is run at the site where the data is stored and not at the client site. The results are then transferred to the client site. This is appropriate for operations where the operands are available at the same site. Example: Select and Project operations.

Data Shipping – In data shipping, the data fragments are transferred to the database server, where the operations are executed. This is used in operations where the operands are distributed at different sites. This is also appropriate in systems where the communication costs are low, and local processors are much slower than the client server.

Hybrid Shipping – This is a combination of data and operation shipping. Here, data fragments are transferred to the high-speed processors, where the operation runs. The results are then sent to the client site.



Phases of Query Processing



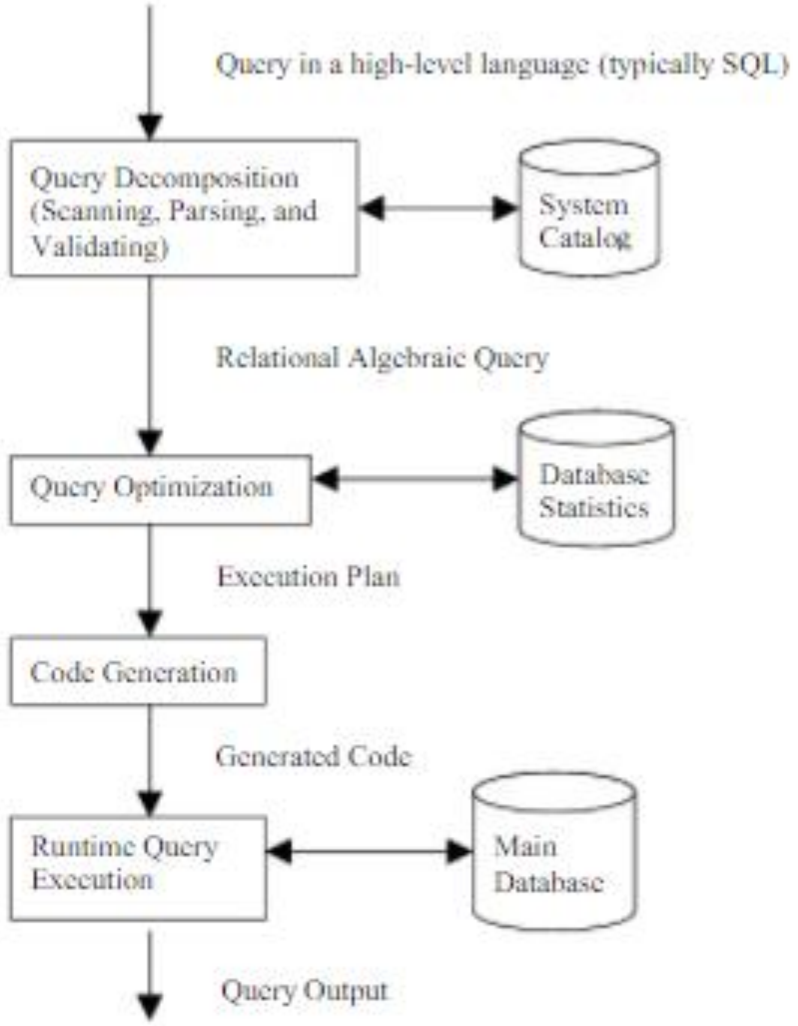
- Query transformation
  - Translation of SQL to internal representation (Relational Algebra)
  - Name resolution: object names $\rightarrow$ internal names (catalog)
  - Semantic analysis: verification of global relations and attributes, view expansion, global access control
  - Normalization: transformation to canonical format
  - Algebraic optimization: improve "‘efficiency"’ of algebra expression
- Data localization:
  - Identification of nodes with fragments of used relations (from distribution schema)
- Global optimization:
  - Selection of least expensive query plan
  - Consideration of costs (execution and communication, cardinalities of intermediate results)
  - Determination of execution order and place

- Local optimization
  - Optimization of fragment query on each node
  - Using local catalog data (statistics)
  - Usage of index structures
  - Cost-based selection of locally optimal plan
- Code-generation
  - Map query plan to executable code

Global Query Optimization. Optimization consists of selecting a strategy from a list of candidates that is closest to optimal. A list of candidate queries can be obtained by permuting the ordering of operations within a fragment query generated by the previous stage. Time is the preferred unit for measuring cost. The total cost is a weighted combination of costs such as CPU cost, I/O costs, and communication costs. Since DDBs are connected by a net-work, often the communication costs over the network are the most significant. This is especially true when the sites are connected through a wide area network (WAN).

Local Query Optimization. This stage is common to all sites in the DDB. The techniques are similar to those used in centralized systems.

Steps in Query Processing for Centralized DBMS



The query optimization is a critical performance issue in centralized database management system, because the main difficulty is to choose one execution strategy that minimizes the consumption of computing resources. Another objective of query optimization is to reduce the total execution time of the query which is the sum of the execution times of all individual operations that make up the query. There are two main techniques for query optimization in centralized DBMS, although the two methods are usually combined in practice. The first approach uses **heuristic rules** that order the operations in a query while the second approach compares different strategies based on their relative costs and selects one which requires minimum computing resources. In a distributed system, several additional factors further complicate the process of query execution. In general, the relations are fragmented in a distributed database, therefore, the distributed query processor transform a high-level query on a logical distributed database (entire relation) into a sequence of database operations (relational algebra) on relation fragments. The data accessed by the query in a distributed database must be localized so that the database operations can be performed on local data or relation fragments. Finally, the query on fragments must be extended with communication operations, and optimization should be done with respect to a cost function that minimizes the use of computing resources such as disk I/Os, CPUs and communication networks.

Let us consider the following two relations which are stored in a centralized DBMS

Employee (<u>empid</u>, ename, salary, designation, deptno)
Department (<u>deptno</u>, dname, location)

and the following query:

"Retrieve the names of all employees whose department location is 'inside' the campus"

where, **empid** and **deptno** are primary key for the relation Employee and Department respectively and **deptno** is a foreign key of the relation Employee.

Using SQL, the above query can be expressed as:

Select ename from Employee, Department where Employee.deptno = Department.deptno and location = "inside".

Two equivalent relational algebraic expressions that correspond to the above SQL statement are as follows:

(i) $\prod_{ename} (\sigma_{(location = \text{'inside'}) \wedge (Employee.deptno = Department.deptno)} (Employee \times Department))$

(ii) $\prod_{ename}(Employee \bowtie_{Employee.deptno = Department.deptno} (\sigma_{location = \text{'inside'}} (Department)))$

In the first relational algebraic expression the projection and the selection operation has been performed after calculating the Cartesian product of two relations Employee and Department, while in the second expression the Cartesian Product has been performed after performing the selection and the projection operation from individual relations. Obviously, the use of computing resources is lesser in the second expression. Thus, in a centralized DBMS, it is easier to choose the optimum execution strategy based on a number of relational algebraic expressions that are equivalent to the same query. In distributed context, the query processing is significantly more difficult because the choice of optimum execution strategy depends on some other factors such as data transfer among sites and the selection of best site for query execution. The problem of distributed query processing is discussed below with the help of an example.

2. Equivalence Transformations for Queries

Query: A query is a request for information from a database.
Query Plans: A query plan (or query execution plan) is an ordered set of steps used to access data in a SQL relational database management system.
Query Optimization: A single query can be executed through different algorithms or re-written in different forms and structures. Hence, the question of query optimization comes into the picture – Which of these forms or pathways is the most optimal? The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans.
Importance: The goal of query optimization is to reduce the system resources required to fulfill a query, and ultimately provide the user with the correct result set faster.

❖First, it provides the user with faster results, which makes the application seem faster to the user.
❖Secondly, it allows the system to service more queries in the same amount of time, because each request takes less time than unoptimized queries.
❖Thirdly, query optimization ultimately reduces the amount of wear on the hardware (e.g. disk drives), and allows the server to run more efficiently (e.g. lower power consumption, less memory usage).

2. Equivalence Transformations for Queries

Equivalence Rule :

❖The equivalence rule says that expressions of two forms are the same or equivalent because both expressions produce the same outputs on any legal database instance.

❖It means that we can possibly replace the expression of the first form with that of the second form and replace the expression of the second form with an expression of the first form.

❖Thus, the optimizer of the query-evaluation plan uses such an equivalence rule or method for transforming expressions into the logically equivalent one.

❖The optimizer uses various equivalence rules on relational-algebra expressions for transforming the relational expressions. For describing each rule, we will use the following symbols:

$\theta, \theta_1, \theta_2$ … : Used for denoting the predicates.

L_1, L_2, L_3 … : Used for denoting the list of attributes.

E, E_1, E_2 …. : Represents the relational-algebra expressions.

There are broadly two ways a query can be optimized:
Analyze and transform equivalent relational expressions:
Try to minimize the tuple and column counts of the intermediate and final query processes (discussed here).
Using different algorithms for each operation:
These underlying algorithms determine how tuples are accessed from the data structures they are stored in, indexing, hashing, data retrieval and hence influence the number of disk and block accesses (discussed in query processing).

Analyze and transform equivalent relational expressions
To analyze equivalent expression, listed are a set of equivalence rules. These generate equivalent expressions for a query written in relational algebra. To optimize a query, we must convert the query into its equivalent form as long as an equivalence rule is satisfied.

1. **Conjunctive selection operations can be written as a sequence of individual selections. This is called a sigma-cascade.**

   $$\sigma_{\theta_1 \Lambda \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

   Explanation: Applying condition θ_1 intersection θ_2 is expensive. Instead, filter out tuples satisfying condition θ_2 (inner selection) and then apply condition θ_1 (outer selection) to the then resulting fewer tuples. This leaves us with less tuples to process the second time. This can be extended for two or more intersecting selections. Since we are breaking a single condition into a series of selections or cascades, it is called a "cascade".

2. **Selection is commutative.**

   $$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

   Explanation: σ condition is commutative in nature. This means, it does not matter whether we apply σ_1 first or σ_2 first. In practice, it is better and more optimal to apply that selection first which yields a fewer number of tuples. This saves time on our outer selection.

3. **All following projections can be omitted, only the first projection is required. This is called a pi-cascade.**

$$\pi_{L_1}(\pi_{L_2}(...(\pi_{L_n}(E))...)) = \pi_{L_1}(E)$$

Explanation: A cascade or a series of projections is meaningless. This is because in the end, we are only selecting those columns which are specified in the last, or the outermost projection. Hence, it is better to collapse all the projections into just one i.e. the outermost projection.

4. **Selections on Cartesian Products can be re-written as Theta Joins.**
   - **Equivalence 1**

     $$\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$$

     Explanation: The cross product operation is known to be very expensive. This is because it matches each tuple of E1 (total m tuples) with each tuple of E2 (total n tuples). This yields m*n entries. If we apply a selection operation after that, we would have to scan through m*n entries to find the suitable tuples which satisfy the condition θ. Instead of doing all of this, it is more optimal to use the Theta Join, a join specifically designed to select only those entries in the cross product which satisfy the Theta condition, without evaluating the entire cross product first.

   - **Equivalence 2**

     $$\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \Lambda \theta_2} E_2$$

     Explanation: Theta Join radically decreases the number of resulting tuples, so if we apply an intersection of both the join conditions i.e. θ_1 and θ_2 into the Theta Join itself, we get fewer scans to do. On the other hand, a σ_1 condition outside unnecessarily increases the tuples to scan.

5. **Theta Joins are commutative.**

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

Explanation: Theta Joins are commutative, and the query processing time depends to some extent which table is used as the outer loop and which one is used as the inner loop during the join process (based on the indexing structures and blocks).

6. **Join operations are associative.**
   - **Natural Join**

     $$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

     Explanation: Joins are all commutative as well as associative, so one must join those two tables first which yield less number of entries, and then apply the other join.

   - **Theta Join**

     $$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \Lambda \theta_3} E_3 = E_1 \bowtie_{\theta_1 \Lambda \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

     Explanation: Theta Joins are associative in the above manner, where θ_2 involves attributes from only E2 and E3.

7. **Selection operation can be distributed.**

- **Equivalence 1**
  $$\sigma_{\theta_1 \Lambda \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$
  Explanation: Applying a selection after doing the Theta Join causes all the tuples returned by the Theta Join to be monitored after the join. If this selection contains attributes from only E1, it is better to apply this selection to E1 (hence resulting in a fewer number of tuples) and then join it with E2.

- **Equivalence 2**
  $$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$
  Explanation: This can be extended to two selection conditions, θ_1 and θ_2, where Theta1 contains the attributes of only E1 and θ_2 contains attributes of only E2. Hence, we can individually apply the selection criteria before joining, to drastically reduce the number of tuples joined.

8. **Projection distributes over the Theta Join.**

- **Equivalence 1**

  $$\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\pi_{L_1}(E_1)) \bowtie_\theta (\pi_{L_2}(E_2))$$

  Explanation: The idea discussed for selection can be used for projection as well. Here, if L1 is a projection that involves columns of only E1, and L2 another projection that involves the columns of only E2, then it is better to individually apply the projections on both the tables before joining. This leaves us with a fewer number of columns on either side, hence contributing to an easier join.

- **Equivalence 2**

  $$\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \pi_{L_1 \cup L_2}((\pi_{L_1 \cup L_3})) \bowtie_\theta (\pi_{L_2 \cup L_4}(E_2)))$$

  Explanation: Here, when applying projections L1 and L2 on the join, where L1 contains columns of only E1 and L2 contains columns of only E2, we can introduce another column E3 (which is common between both the tables). Then, we can apply projections L1 and L2 on E1 and E2 respectively, along with the added column L3. L3 enables us to do the join.

9. **Union and Intersection are commutative.**
$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$
Explanation: Union and intersection are both distributive; we can enclose any tables in parentheses according to requirement and ease of access.

10. **Union and Intersection are associative.**
$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$
Explanation: Union and intersection are both distributive; we can enclose any tables in parentheses according to requirement and ease of access.

11. **Selection operation distributes over the union, intersection, and difference operations.**
$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$$
Explanation: In set difference, we know that only those tuples are shown which belong to table E1 and do not belong to table E2. So, applying a selection condition on the entire set difference is equivalent to applying the selection condition on the individual tables and then applying set difference. This will reduce the number of comparisons in the set difference step.

12. **Projection operation distributes over the union operation.**

$$\pi_L(E_1 \cup E_2) = (\pi_L(E_1)) \cup (\pi_L(E_2))$$

Explanation: Applying individual projections before computing the union of E1 and E2 is more optimal than the left expression, i.e. applying projection after the union step.

Minimality

A set of equivalence rules is said to be minimal if no rule can be derived from any combination of the others. A query is said to be optimal when it is minimal.

Examples

Assume the following tables:

```
instructor(ID, name, dept_name, salary)
teaches(ID, course_id, sec_id, semester, year)
course(course_id, title, dept_name, credits)
```

Query 1: Find the names of all instructors in the Music department, along with the titles of the courses that they teach

$$\pi_{name,title}(\sigma_{dept\_name=\text{“}Music\text{”}}(instructor \bowtie (teaches \bowtie \pi_{course\_id,title}(course))))$$

Here, dept_name is a field of only the instructor table. Hence, we can select out the Music instructors before joining the tables, hence reducing query time.

Optimized Query:
Using rule 7a, and Performing the selection as early as possible reduces the size of the relation to be joined.

$$\pi_{name,title}((\sigma_{dept\_name=\text{“}Music\text{”}}(instructor) \bowtie (teaches \bowtie \pi_{course\_id,title}(course)))$$

Query 2: Find the names of all instructors in the Music department who have taught a course in 2009, along with the titles of the courses that they taught

$$\sigma_{dept\_name=\text{“}CSE\text{”}}(\sigma_{year=2009}(instructor \bowtie teaches))$$

Optimized Query:

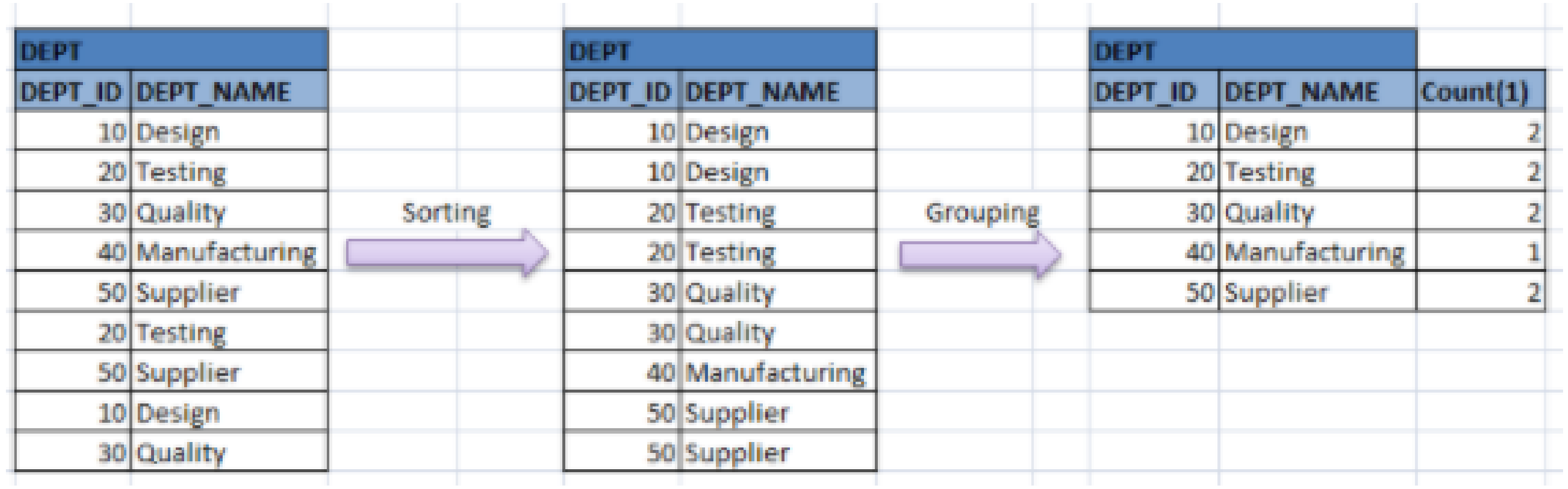
We can perform an "early selection", hence the optimized query becomes:

$$\sigma_{dept\_name=\text{“}CSE\text{”}}(instructor) \bowtie \sigma_{year=2009}(teaches)$$

3. Distributed Grouping and Aggregate Function Evaluation

❖This is also implemented similar to duplicate elimination. Sorting or hashing techniques are used to get similar groups of records together.
❖Then grouping functions like SUM, AVG etc are applied on it.
Suppose we want to see number duplicate records entered for each department in DEPT table.
❖Since the table does not have any keys (hence it allowed entering duplicate records), the records are not sorted and are placed in the table as they are entered.
❖In typical case, it has to pick one record and search the whole table for rest of the record to count the number for that record.
❖Then it has to repeat the same step with rest of the records.
This is a costly method of checking the counts.
❖Hence what it does is, first sorts the records of DEPT table (either by sorting or hashing method).
❖Then it counts the number of similar records and displays the results as below.

SELECT DEPT_ID, DEPT_NAME, COUNT (1) FROM DEPT GROUP BY DEPT_ID, DEPT_NAME;

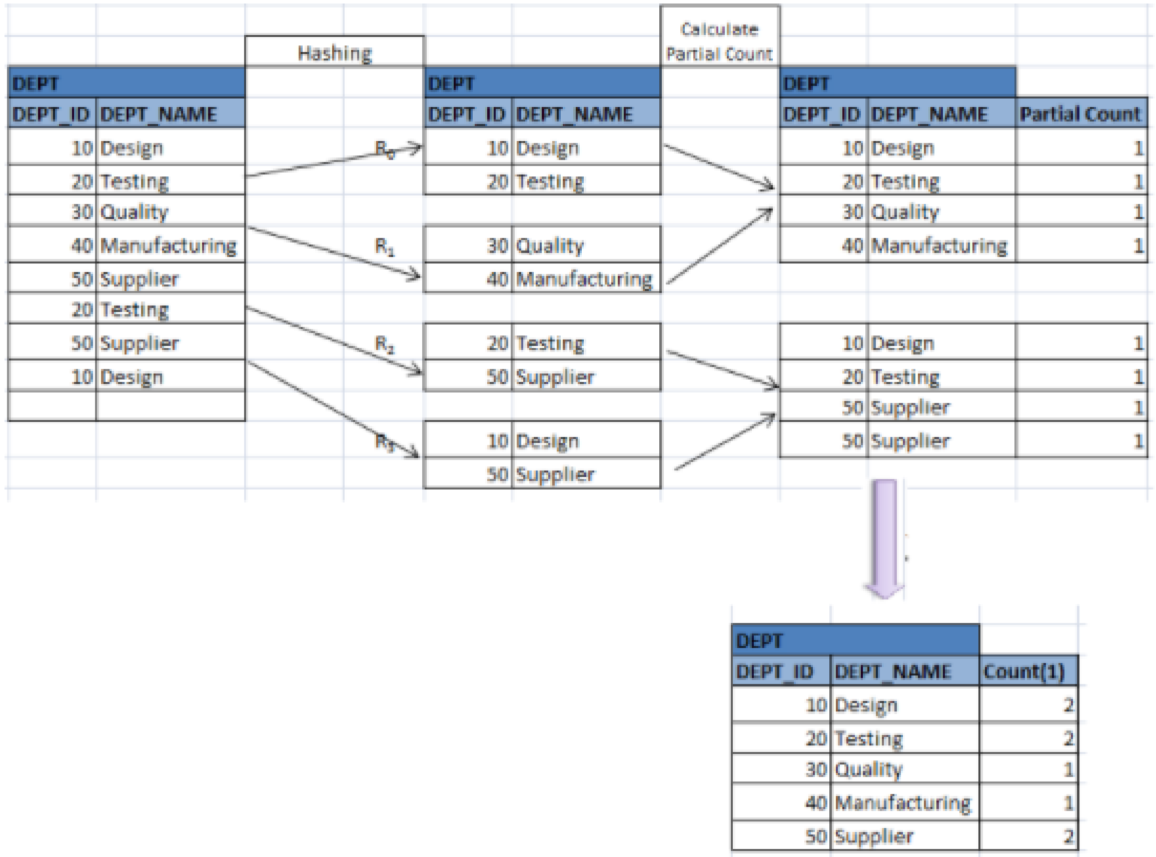


DEPT	
DEPT_ID	DEPT_NAME
10	Design
20	Testing
30	Quality
40	Manufacturing
50	Supplier
20	Testing
50	Supplier
10	Design
30	Quality

DEPT	
DEPT_ID	DEPT_NAME
10	Design
10	Design
20	Testing
20	Testing
30	Quality
30	Quality
40	Manufacturing
50	Supplier
50	Supplier

DEPT		
DEPT_ID	DEPT_NAME	Count(1)
10	Design	2
20	Testing	2
30	Quality	2
40	Manufacturing	1
50	Supplier	2

In hashing method, sorting can even optimized during run generation or merge passes. During run or merge passes, it can calculate the partial aggregate values in that block and can pass it to next merge pass.
For example, say we have one run and one merge pass in above case, and block size is 2 with one record each.



DEPT

DEPT_ID	DEPT_NAME
10	Design
20	Testing
30	Quality
40	Manufacturing
50	Supplier
20	Testing
50	Supplier
10	Design

DEPT

DEPT_ID	DEPT_NAME
10	Design
20	Testing
30	Quality
40	Manufacturing
20	Testing
50	Supplier
10	Design
50	Supplier

DEPT

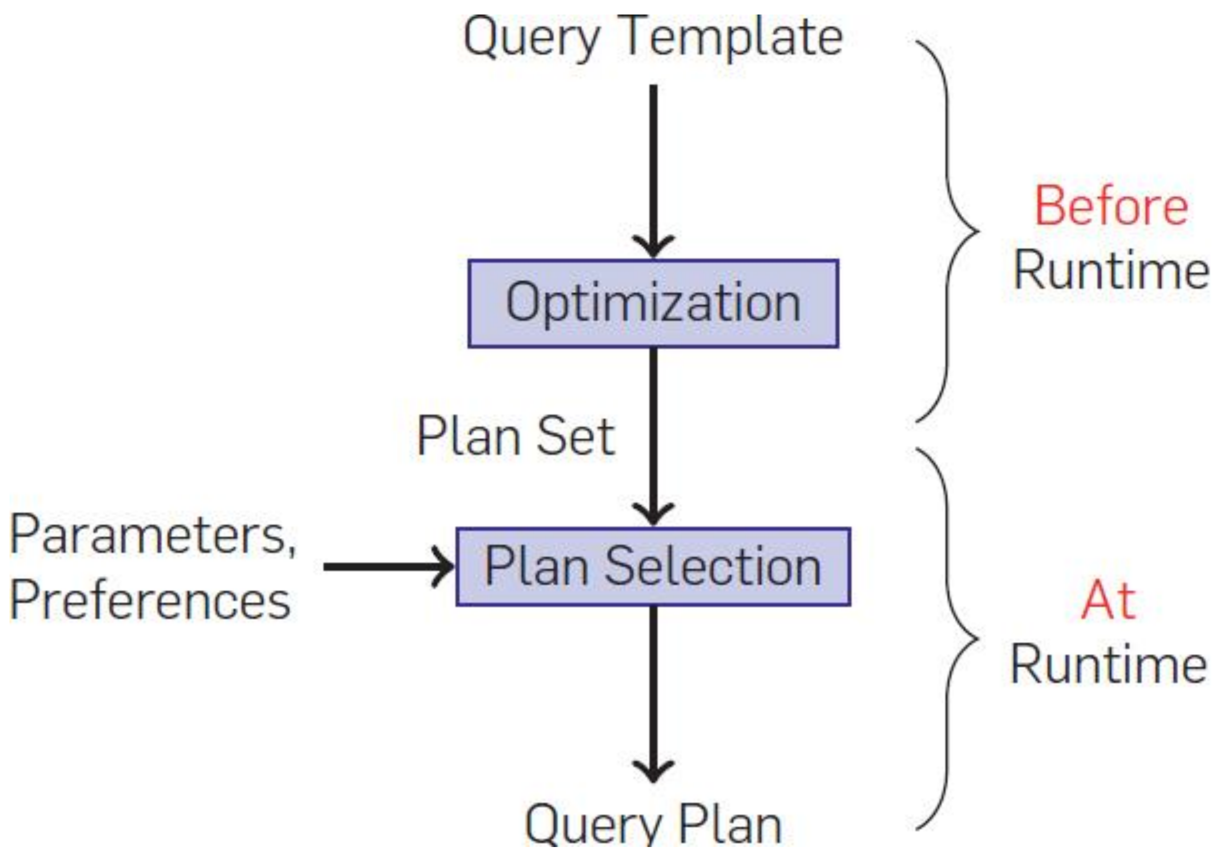
DEPT_ID	DEPT_NAME	Partial Count
10	Design	1
20	Testing	1
30	Quality	1
40	Manufacturing	1
10	Design	1
20	Testing	1
50	Supplier	1
50	Supplier	1

DEPT

DEPT_ID	DEPT_NAME	Count(1)
10	Design	2
20	Testing	2
30	Quality	1
40	Manufacturing	1
50	Supplier	2

- ❖The task of grouping is distributed among the sorting steps and hence the cost of grouping at the final step is reduced drastically.
- ❖Hence for better performance hashing method follows below steps while grouping.
- ❖When MAX, MIN, SUM, COUNT are used in a query, partial value in each block is calculated and kept.
- ❖It is passed to next level to calculate next partial value and so on.
- ❖When AVG function is used in the query, SUM and COUNT are calculated in each step and at the end SUM is divided by the COUNT.

4. Parametric Queries



parametric query optimization

1. Optimization of Access Strategies

The **query optimizer** (called simply the **optimizer**) is built-in database software that determines the most efficient method for a SQL statement to access requested data.

Purpose of the Query Optimizer
The optimizer attempts to generate the most optimal execution plan for a SQL statement.
The optimizer choose the plan with the lowest cost among all considered candidate plans. The optimizer uses available statistics to calculate cost. For a specific query in a given environment, the cost computation accounts for factors of query execution such as I/O, CPU, and communication.
For example, a query might request information about employees who are managers. If the optimizer statistics indicate that 80% of employees are managers, then the optimizer may decide that a full table scan is most efficient. However, if statistics indicate that very few employees are managers, then reading an index followed by a table access by rowid may be more efficient than a full table scan.
Because the database has many internal statistics and tools at its disposal, the optimizer is usually in a better position than the user to determine the optimal method of statement execution. For this reason, all SQL statements use the optimizer.

The objective of a query optimization routine is to minimize the total cost associated with the execution of a request.

The costs associated with a request are a function of the:

- Access time (I/O) cost involved in accessing the physical data stored on disk.

- Communication cost associated with the transmission of data among nodes in distributed database systems.

- CPU time cost associated with the processing overhead of managing distributed transactions.

Cost-Based 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)** to contrast it with the legacy rule-based optimizer (RBO).

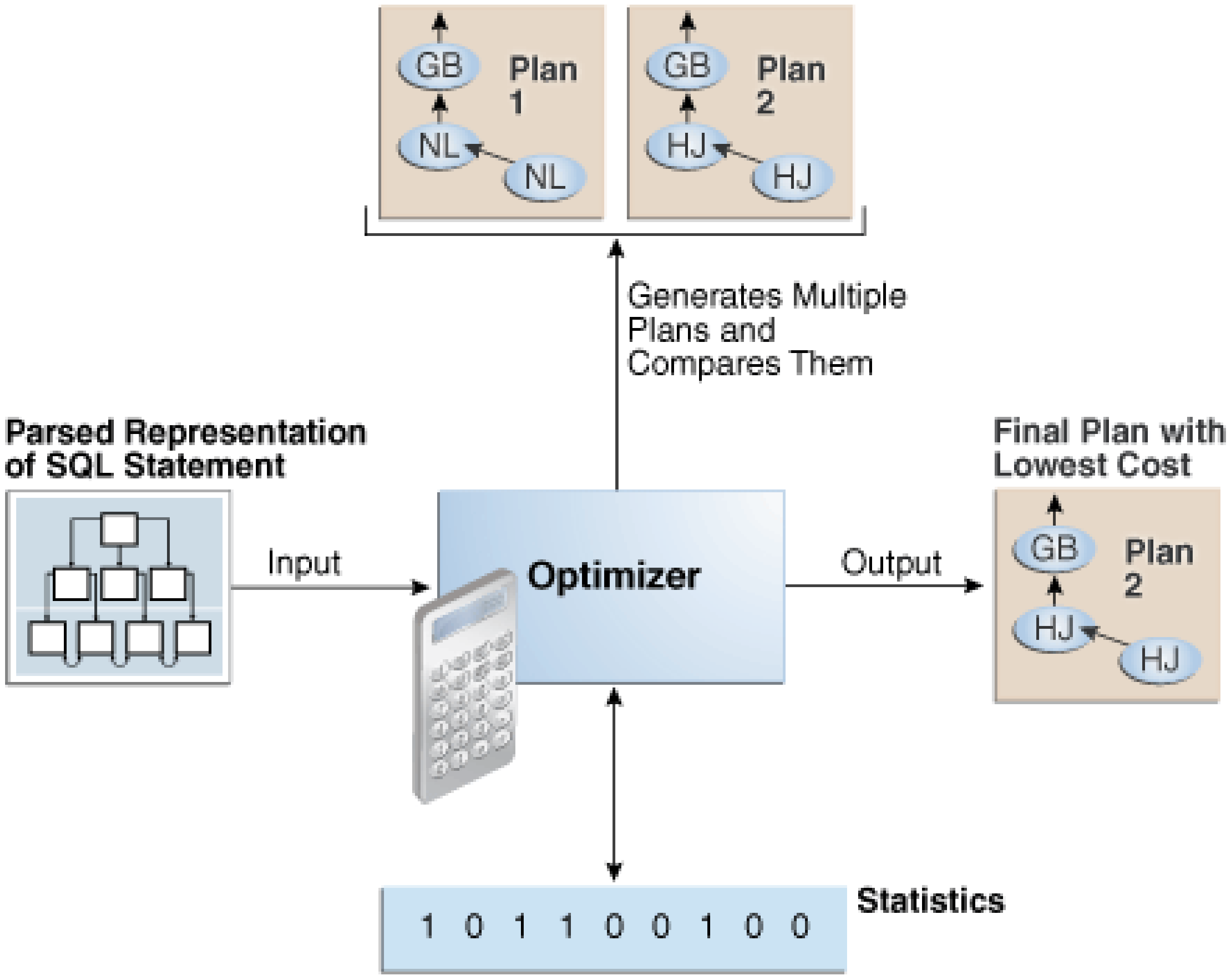
Execution Plans

An **execution plan** describes a recommended method of execution for a SQL statement.

The plan shows the combination of the steps Oracle Database uses to execute a SQL statement. Each step either retrieves rows of data physically from the database or prepares them for the user issuing the statement.

An execution plan displays the cost of the entire plan, indicated on line 0, and each separate operation. The cost is an internal unit that the execution plan only displays to allow for plan comparisons. Thus, you cannot tune or change the cost value.

In the following graphic, the optimizer generates two possible execution plans for an input SQL statement, uses statistics to estimate their costs, compares their costs, and then chooses the plan with the lowest cost.


GB
Plan 1
NL
NL
GB
Plan 2
HJ
HJ
Generates Multiple Plans and Compares Them
Parsed Representation of SQL Statement
Input
Optimizer
Output
Final Plan with Lowest Cost
GB
Plan 2
HJ
HJ
1 0 1 1 0 0 1 0 0
Statistics

Query Blocks

The input to the optimizer is a parsed representation of a SQL statement.

Each `SELECT` block in the original SQL statement is represented internally by a **query block**. A query block can be a top-level statement, subquery, or unmerged view.

Example 4-1 Query Blocks

The following SQL statement consists of two query blocks. The subquery in parentheses is the inner query block. The outer query block, which is the rest of the SQL statement, retrieves names of employees in the departments whose IDs were supplied by the subquery. The query form determines how query blocks are interrelated.

```
SELECT first_name, last_name
FROM   hr.employees
WHERE  department_id
IN     (SELECT department_id
        FROM   hr.departments
        WHERE  location_id = 1800);
```

Query Subplans:

For each query block, the optimizer generates a query subplan.

The database optimizes query blocks separately from the bottom up. Thus, the database optimizes the innermost query block first and generates a subplan for it, and then generates the outer query block representing the entire query.

The number of possible plans for a query block is proportional to the number of objects in the FROM clause. This number rises exponentially with the number of objects. For example, the possible plans for a join of five tables are significantly higher than the possible plans for a join of two tables.

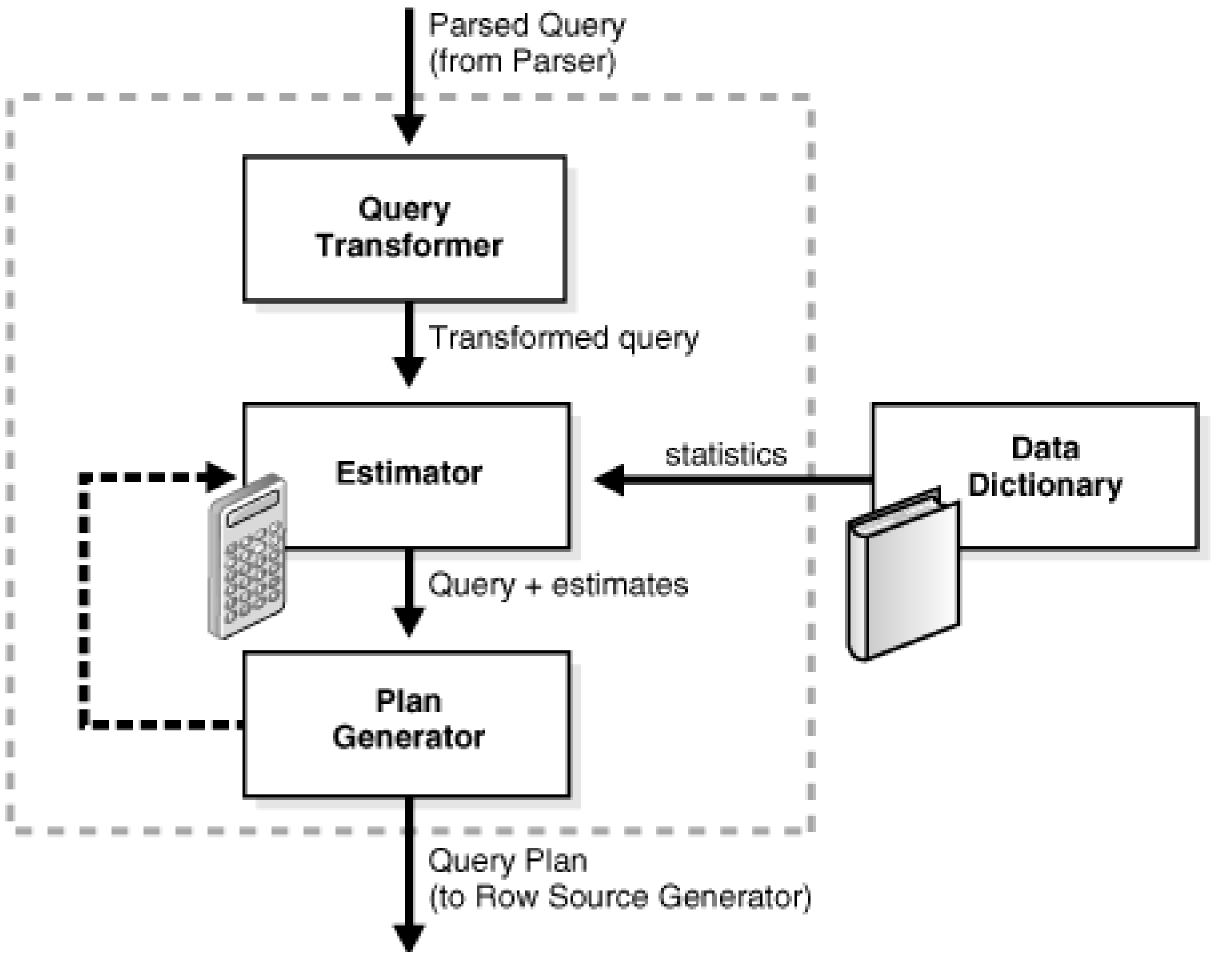
Analogy for the Optimizer:

One analogy for the optimizer is an online trip advisor.

A cyclist wants to know the most efficient bicycle route from point A to point B. A query is like the directive "I need the most efficient route from point A to point B" or "I need the most efficient route from point A to point B by way of point C." The trip advisor uses an internal algorithm, which relies on factors such as speed and difficulty, to determine the most efficient route. The cyclist can influence the trip advisor's decision by using directives such as "I want to arrive as fast as possible" or "I want the easiest ride possible."

In this analogy, an execution plan is a possible route generated by the trip advisor. Internally, the advisor may divide the overall route into several subroutes (subplans), and calculate the efficiency for each subroute separately. For example, the trip advisor may estimate one subroute at 15 minutes with medium difficulty, an alternative subroute at 22 minutes with minimal difficulty, and so on.
The advisor picks the most efficient (lowest cost) overall route based on user-specified goals and the available statistics about roads and traffic conditions. The more accurate the statistics, the better the advice. For example, if the advisor is not frequently notified of traffic jams, road closures, and poor road conditions, then the recommended route may turn out to be inefficient (high cost).

About Optimizer Components
The optimizer contains three components: the transformer, estimator, and plan generator.
The following graphic illustrates the components..


Parsed Query
(from Parser)
Query Transformer
Transformed query
Estimator
statistics
Data Dictionary
Query + estimates
Plan Generator
Query Plan
(to Row Source Generator)

A set of query blocks represents a parsed query, which is the input to the optimizer. The following table describes the optimizer operations.

Table 4-1 Optimizer Operations

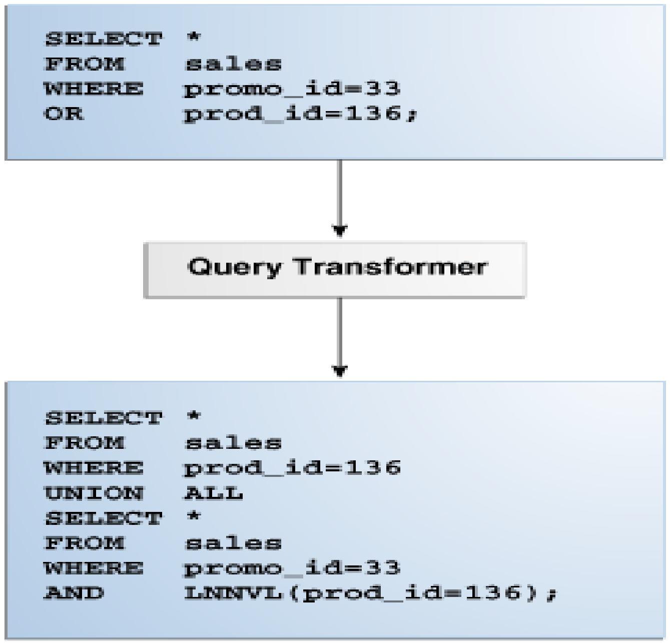
Phase	Operation	Description	To Learn More
1	Query Transformer	The optimizer determines whether it is helpful to change the form of the query so that the optimizer can generate a better execution plan.	"Query Transformer"
2	Estimator	The optimizer estimates the cost of each plan based on statistics in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

Query Transformer

Query Transformer

For some statements, the query transformer determines whether it is advantageous to rewrite the original SQL statement into a semantically equivalent SQL statement with a lower cost.

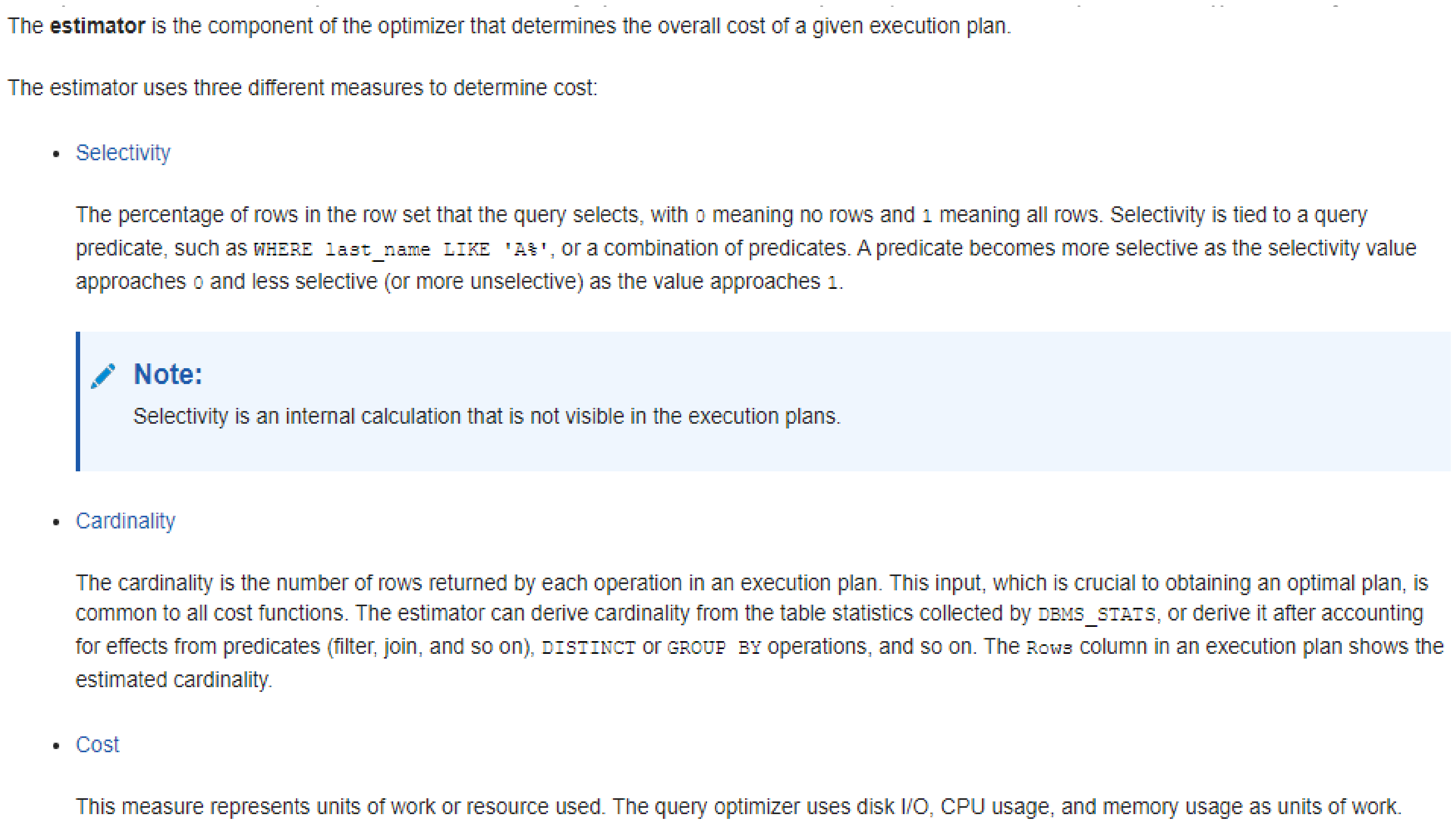
When a viable alternative exists, the database calculates the cost of the alternatives separately and chooses the lowest-cost alternative. The following graphic shows the query transformer rewriting an input query that uses OR into an output query that uses UNION ALL.



```
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

  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`.

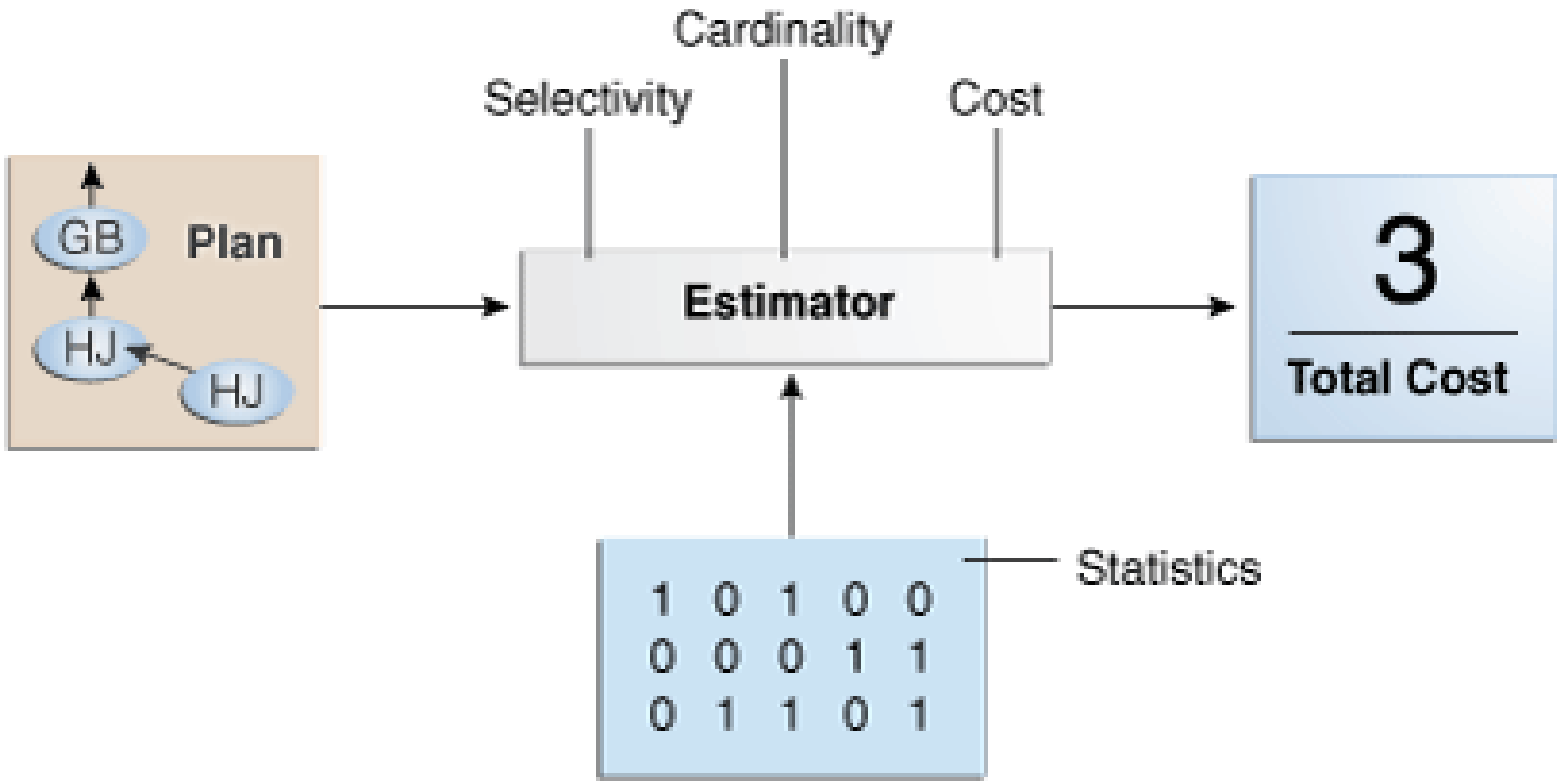
  > **Note:**
  >
  > Selectivity is an internal calculation that is not visible in the execution plans.

- 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.


Cardinality
Selectivity
Cost
GB
Plan
HJ
HJ
Estimator
3
Total Cost
1 0 1 0 0
0 0 0 1 1
0 1 1 0 1
Statistics

For the query shown in Example 4-1, the estimator uses selectivity, estimated cardinality (a total return of 10 rows), and cost measures to produce its to
cost estimate of 3:

```
--------------------------------------------------------------------------------
| Id| Operation                    |Name              |Rows|Bytes|Cost %CPU|Time|
--------------------------------------------------------------------------------
| 0 | SELECT STATEMENT             |                  | 10| 250| 3 (0)| 00:00:01|
| 1 |  NESTED LOOPS                |                  |   |    |      |         |
| 2 |   NESTED LOOPS               |                  | 10| 250| 3 (0)| 00:00:01|
|*3 |    TABLE ACCESS FULL         |DEPARTMENTS       |  1|   7| 2 (0)| 00:00:01|
|*4 |    INDEX RANGE SCAN          |EMP_DEPARTMENT_IX| 10|    | 0 (0)| 00:00:01|
| 5 |   TABLE ACCESS BY INDEX ROWID|EMPLOYEES         | 10| 180| 1 (0)| 00:00:01|
--------------------------------------------------------------------------------
```

Selectivity

The **selectivity** represents a fraction of rows from a row set.

The row set can be a base table, a view, or the result of a join. The selectivity is tied to a query predicate, such as `last_name = 'Smith'`, or a combination of predicates, such as `last_name = 'Smith' AND job_id = 'SH_CLERK'`.

A predicate filters a specific number of rows from a row set. Thus, the selectivity of a predicate indicates how many rows pass the predicate test. Selectivity ranges from 0.0 to 1.0. A selectivity of 0.0 means that no rows are selected from a row set, whereas a selectivity of 1.0 means that all rows are selected. A predicate becomes more selective as the value approaches 0.0 and less selective (or more unselective) as the value approaches 1.0.

The optimizer estimates selectivity depending on whether statistics are available:

- Statistics not available

  Depending on the value of the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter, the optimizer either uses **dynamic statistics** or an internal default value. The database uses different internal defaults depending on the predicate type. For example, the internal default for an equality predicate (`last_name = 'Smith'`) is lower than for a range predicate (`last_name > 'Smith'`) because an equality predicate is expected to return a smaller fraction of rows.

- Statistics available

  When statistics are available, the estimator uses them to estimate selectivity. Assume there are 150 distinct employee last names. For an equality predicate `last_name = 'Smith'`, selectivity is the reciprocal of the number *n* of distinct values of `last_name`, which in this example is .006 because the query selects rows that contain 1 out of 150 distinct values.

  If a histogram exists on the `last_name` column, then the estimator uses the histogram instead of the number of distinct values. The histogram captures the distribution of different values in a column, so it yields better selectivity estimates, especially for columns that have **data skew**.

Cardinality

The **cardinality** is the number of rows returned by each operation in an execution plan.

For example, if the optimizer estimate for the number of rows returned by a full table scan is 100, then the cardinality estimate for this operation is 100. The cardinality estimate appears in the `Rows` column of the execution plan.

The optimizer determines the cardinality for each operation based on a complex set of formulas that use both table and column level statistics, or dynamic statistics, as input. The optimizer uses one of the simplest formulas when a single equality predicate appears in a single-table query, with no histogram. In this case, the optimizer assumes a uniform distribution and calculates the cardinality for the query by dividing the total number of rows in th table by the number of distinct values in the column used in the `WHERE` clause predicate.

For example, user `hr` queries the `employees` table as follows:

```
SELECT first_name, last_name
FROM   employees
WHERE  salary='10200';
```

The `employees` table contains 107 rows. The current database statistics indicate that the number of distinct values in the `salary` column is `58`. Therefore, the optimizer estimates the cardinality of the result set as `2`, using the formula `107/58=1.84`.

Cardinality estimates must be as accurate as possible because they influence all aspects of the execution plan. Cardinality is important when the optimizer determines the cost of a join. For example, in a nested loops join of the `employees` and `departments` tables, the number of rows in `employee` determines how often the database must probe the `departments` table. Cardinality is also important for determining the cost of sorts.

Cost

The **optimizer cost model** accounts for the machine resources that a query is predicted to use.

The **cost** is an internal numeric measure that represents the estimated resource usage for a plan. The cost is *specific* to a query in an optimizer environment. To estimate cost, the optimizer considers factors such as the following:

- System resources, which includes estimated I/O, CPU, and memory
- Estimated number of rows returned (cardinality)
- Size of the initial data sets
- Distribution of the data
- Access structures

The execution time is a function of the cost, but cost does not equate directly to time. For example, if the plan for query *A* has a lower cost than the plan for query *B*, then the following outcomes are possible:

- *A* executes faster than *B*.
- *A* executes slower than *B*.
- *A* executes in the same amount of time as *B*.

Therefore, you cannot compare the costs of different queries with one another. Also, you cannot compare the costs of semantically equivalent queries that use different optimizer modes.

Example 4-2 Cost in a Sample Execution Plan

The execution plan displays the cost of the entire plan, which is indicated on line `0`, and each individual operation. For example, the following plan shows an overall cost of `14`.

```
EXPLAINED SQL STATEMENT:
------------------------
SELECT prod_category, AVG(amount_sold) FROM   sales s, products p WHERE
 p.prod_id = s.prod_id GROUP BY prod_category

Plan hash value: 4073170114


----------------------------------------------------------------------
| Id  | Operation                  | Name                  | Cost (%CPU)|
----------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                       |    14 (100)|
|   1 |  HASH GROUP BY             |                       |    14  (22)|
|   2 |   HASH JOIN                |                       |    13  (16)|
|   3 |    VIEW                    | index$_join$_002      |     7  (15)|
|   4 |     HASH JOIN              |                       |            |
|   5 |      INDEX FAST FULL SCAN| PRODUCTS_PK           |     4   (0)|
|   6 |      INDEX FAST FULL SCAN| PRODUCTS_PROD_CAT_IX |     4   (0)|
|   7 |    PARTITION RANGE ALL     |                       |     5   (0)|
|   8 |     TABLE ACCESS FULL      | SALES                 |     5   (0)|
----------------------------------------------------------------------
```

The access path determines the number of units of work required to get data from a base table. To determine the overall plan cost, the optimizer assigns a cost to each access path:

The access path determines the number of units of work required to get data from a base table. To determine the overall plan cost, the optimizer assigns a cost to each access path:

- Table scan or fast full index scan

  During a table scan or fast full index scan, the database reads multiple blocks from disk in a single I/O. The cost of the scan depends on the number of blocks to be scanned and the multiblock read count value.

- Index scan

  The cost of an index scan depends on the levels in the B-tree, the number of index leaf blocks to be scanned, and the number of rows to be fetched using the rowid in the index keys. The cost of fetching rows using rowids depends on the **index clustering factor**.

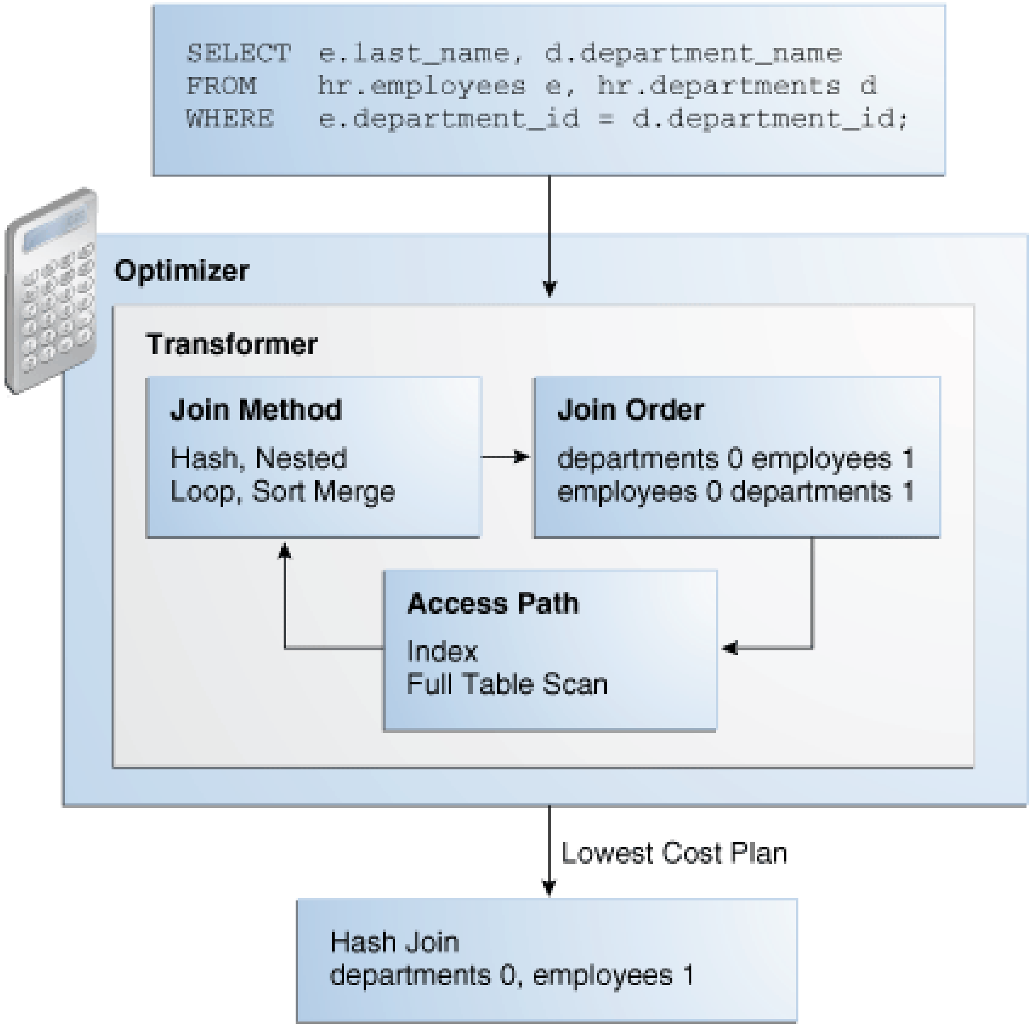
The **join cost** represents the combination of the individual access costs of the two row sets being joined, plus the cost of the join operation.

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 th lowest cost.

The following graphic shows the optimizer testing different plans for an input query.


```
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

Tuning Optimizer

The optimizer performs different operations depending on how it is invoked.

The database provides the following types of optimization:

Normal optimization

The optimizer compiles the SQL and generates an execution plan. The normal mode generates a reasonable plan for most SQL statements. Under normal mode, the optimizer operates with strict time constraints, usually a fraction of a second, during which it must find an optimal plan.

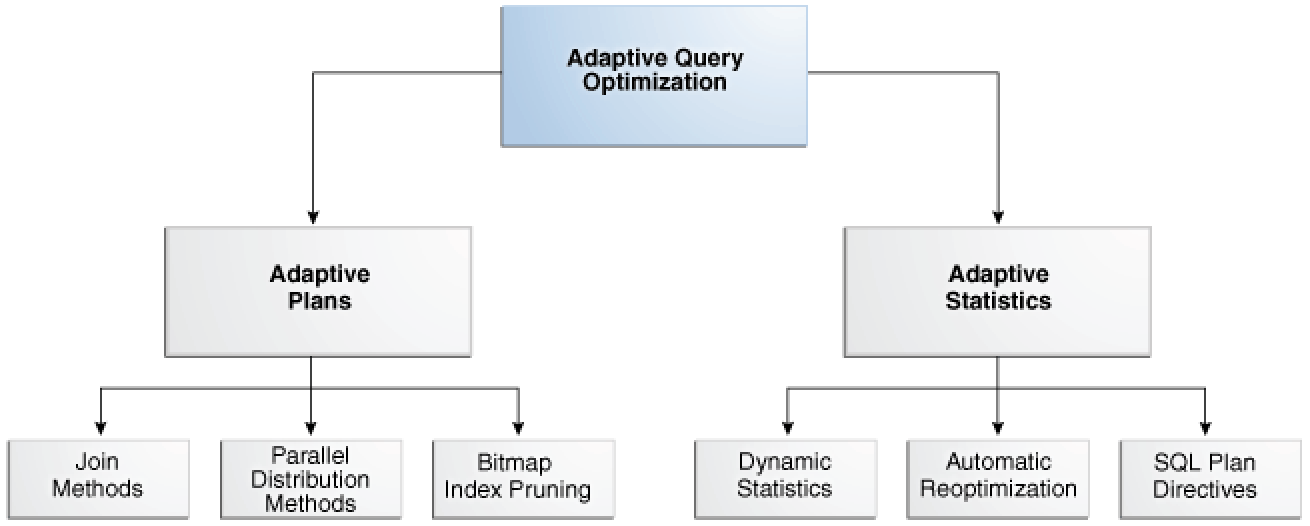
SQL Tuning Advisor optimization

When SQL Tuning Advisor invokes the optimizer, the optimizer is known as **Automatic Tuning Optimizer**. In this case, the optimizer performs additional analysis to further improve the plan produced in normal mode. The optimizer output is not an execution plan, but a series of actions, along with their rationale and expected benefit for producing a significantly better plan.

Adaptive Query Optimization

In Oracle Database, **adaptive query optimization** enables the optimizer to make run-time adjustments to execution plans and discover additional information that can lead to better statistics.

Adaptive optimization is helpful when existing statistics are not sufficient to generate an optimal plan. The following graphic shows the feature set for adaptive query optimization.



Parameterized Queries

An introduction to parameterized queries and query templates.

Another piece of data.world-specific functionality is the ability to declare parameters for a query. Queries with parameters allow you to easily vary the results based on input values without changing the content of the underlying query.

Queries which have parameter values declared are often referred to as **parameterized queries.** In the data.world user interface, you'll also see them referred to as **query templates.**

Parameters are defined at the beginning of a query using a declare statement. Declare statements start with the keyword `DECLARE`, followed by the name of the parameter (starting with a question mark) followed by the type of the parameter and an optional default value. The default value must be a literal value, either `STRING`, `NUMERIC`, `BOOLEAN`, `DATE`, or `TIME`. Once you've declared a parameter, you can then use that parameter in your query, just as though it were any other value.

Queries may declare multiple parameters. If a parameter is declared without a default value, a value must be provided at run-time. If a parameter is declared without a type, the type is assumed to be `STRING`.

In the CRM dataset we have been using, we have a table called `central_office_orders` which lists every order taken by the central office. A common thing to do with such data is to ask for the total orders placed by a given account. For example, we might want to know the total orders for account `"Rundofase"`. Here's how we would do that with a non-parameterized query.

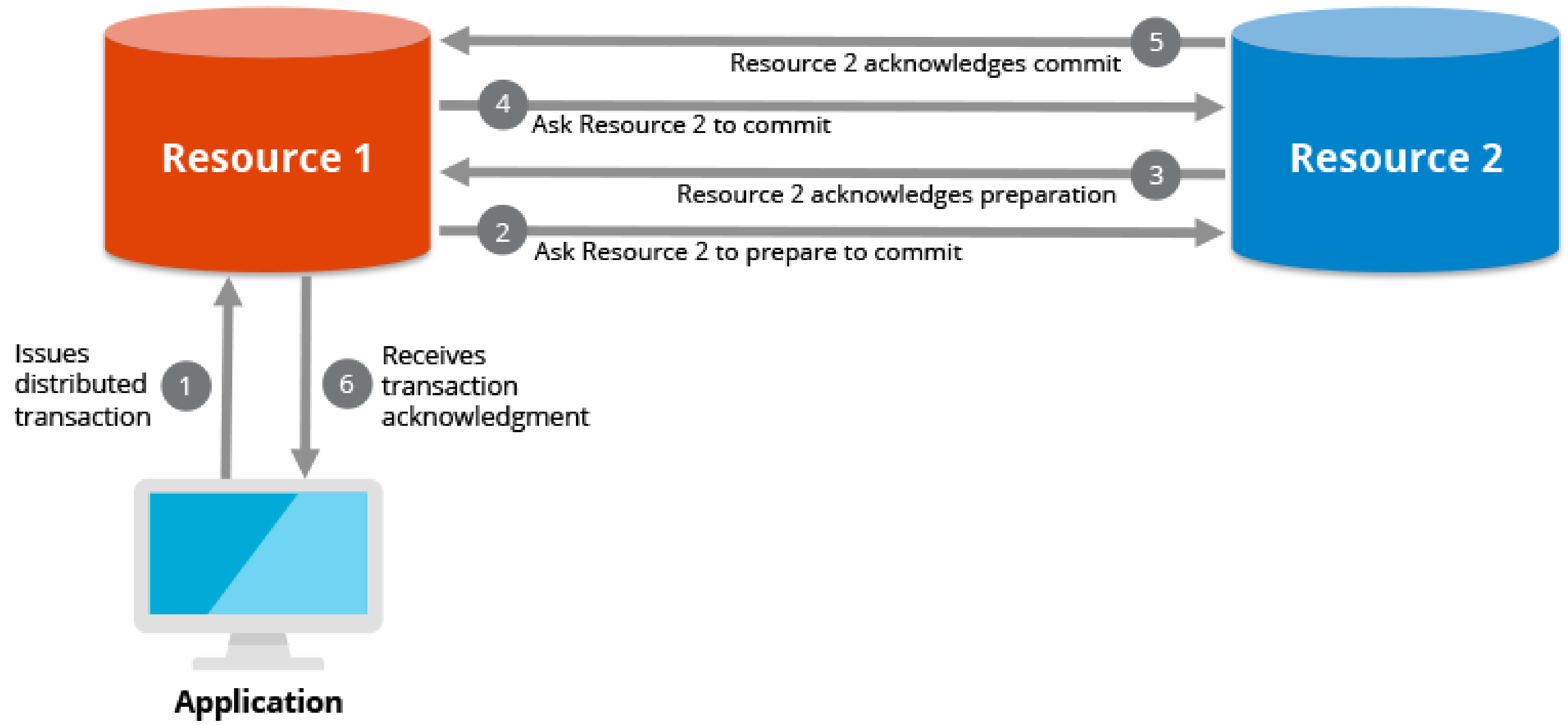
```
SELECT account, SUM(order_total)
  FROM central_office_orders
 WHERE account = "Rundofase"
 GROUP BY account
```

9. Introduction to Distributed Transactions.

✓A **distributed transaction** is a set of operations on data that is performed across two or more data repositories (especially databases).
✓It is typically coordinated across separate nodes connected by a network, but may also span multiple databases on a single server.

✓There are two possible outcomes:
✓1) all operations successfully complete, or
✓2) none of the operations are performed at all due to a failure somewhere in the system. In the latter case, if some work was completed prior to the failure, that work will be reversed to ensure no net work was done.
✓This type of operation is in compliance with the "ACID" (atomicity-consistency-isolation-durability) principles of databases that ensure data integrity.
✓ACID is most commonly associated with transactions on a single database server, but distributed transactions extend that guarantee across multiple databases.

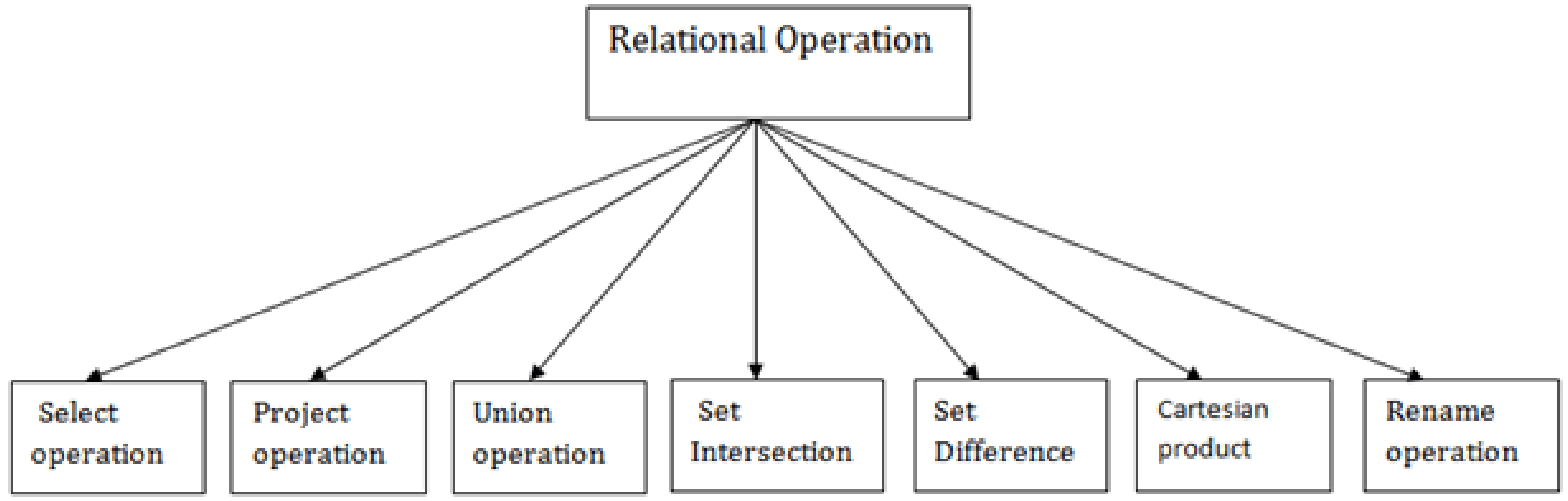
✓The operation known as a "two-phase commit" (2PC) is a form of a distributed transaction.
✓"XA transactions" are transactions using the XA protocol, which is one implementation of a two-phase commit operation.


Resource 1
Resource 2
5 Resource 2 acknowledges commit
4 Ask Resource 2 to commit
3 Resource 2 acknowledges preparation
2 Ask Resource 2 to prepare to commit
Issues distributed transaction 1
6 Receives transaction acknowledgment
Application

Relational Algebra

Relational algebra is a procedural query language. It gives a step by step process to obtain the result of the query. It uses operators to perform queries.

Types of Relational operation



1. Select Operation:
The select operation selects tuples that satisfy a given predicate.
It is denoted by sigma (σ).

Notation: σ p(r)

Where:
σ is used for selection prediction
r is used for relation
p is used as a propositional logic formula which may use connectors like: AND OR and NOT. These relational can use as relational operators like =, ≠, ≥, <, >, ≤.

For example: LOAN Relation

BRANCH_NAME	LOAN_NO	AMOUNT
Downtown	L-17	1000
Redwood	L-23	2000
Perryride	L-15	1500
Downtown	L-14	1500
Mianus	L-13	500
Roundhill	L-11	900
Perryride	L-16	1300

Input:

σ BRANCH_NAME="perryride" (LOAN)

Output:

BRANCH_NAME	LOAN_NO	AMOUNT
Perryride	L-15	1500
Perryride	L-16	1300

2. Project Operation:
This operation shows the list of those attributes that we wish to appear in the result. Rest of the attributes are eliminated from the table.
It is denoted by ∏.
Notation: ∏ A1, A2, An (r)
Where
A1, **A2**, **A3** is used as an attribute name of relation **r**.
Example: CUSTOMER RELATION

NAME	STREET	CITY
Jones	Main	Harrison
Smith	North	Rye
Hays	Main	Harrison
Curry	North	Rye
Johnson	Alma	Brooklyn
Brooks	Senator	Brooklyn

Input:
1.∏ NAME, CITY (CUSTOMER)
Output:

NAME	CITY
Jones	Harrison
Smith	Rye
Hays	Harrison
Curry	Rye
Johnson	Brooklyn
Brooks	Brooklyn

3. Union Operation:
Suppose there are two tuples R and S. The union operation contains all the tuples that are either in R or S or both in R & S.
It eliminates the duplicate tuples. It is denoted by ∪.
Notation: R ∪ S
A union operation must hold the following condition:
R and S must have the attribute of the same number.
Duplicate tuples are eliminated automatically.
Example:
DEPOSITOR RELATION

BORROW RELATION

CUSTOMER_NAME	ACCOUNT_NO
Johnson	A-101
Smith	A-121
Mayes	A-321
Turner	A-176
Johnson	A-273
Jones	A-472
Lindsay	A-284

CUSTOMER_NAME	LOAN_NO
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17

Input:

∏ CUSTOMER_NAME (BORROW) ∪ ∏ CUSTOMER_NAME (DEPOSITOR)

Output:

CUSTOMER_NAME
Johnson
Smith
Hayes
Turner
Jones
Lindsay
Jackson
Curry
Williams
Mayes

4. Set Intersection:
Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in both R & S.
It is denoted by intersection ∩.
Notation: R ∩ S
Example: Using the above DEPOSITOR table and BORROW table
Input:
∏ CUSTOMER_NAME (BORROW) ∩ ∏ CUSTOMER_NAME (DEPOSITOR)
Output:

CUSTOMER_NAME
Smith
Jones

5. Set Difference:
Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in R but not in S.
It is denoted by intersection minus (-).

Notation: R - S

Example: Using the above DEPOSITOR table and BORROW table

Input:

∏ CUSTOMER_NAME (BORROW) - ∏ CUSTOMER_NAME (DEPOSITOR)

Output:

CUSTOMER_NAME
Jackson
Hayes
Willians
Curry

6. Cartesian product

The Cartesian product is used to combine each row in one table with each row in the other table. It is also known as a cross product.

It is denoted by X.

Notation: E X D

Example:

EMPLOYEE

EMP_ID	EMP_NAME	EMP_DEPT
1	Smith	A
2	Harry	C
3	John	B

DEPARTMENT

DEPT_NO	DEPT_NAME
A	Marketing
B	Sales
C	Legal

Input:
EMPLOYEE X DEPARTMENT
Output:

EMP_ID	EMP_NAME	EMP_DEPT	DEPT_NO	DEPT_NAME
1	Smith	A	A	Marketing
1	Smith	A	B	Sales
1	Smith	A	C	Legal
2	Harry	C	A	Marketing
2	Harry	C	B	Sales
2	Harry	C	C	Legal
3	John	B	A	Marketing
3	John	B	B	Sales
3	John	B	C	Legal

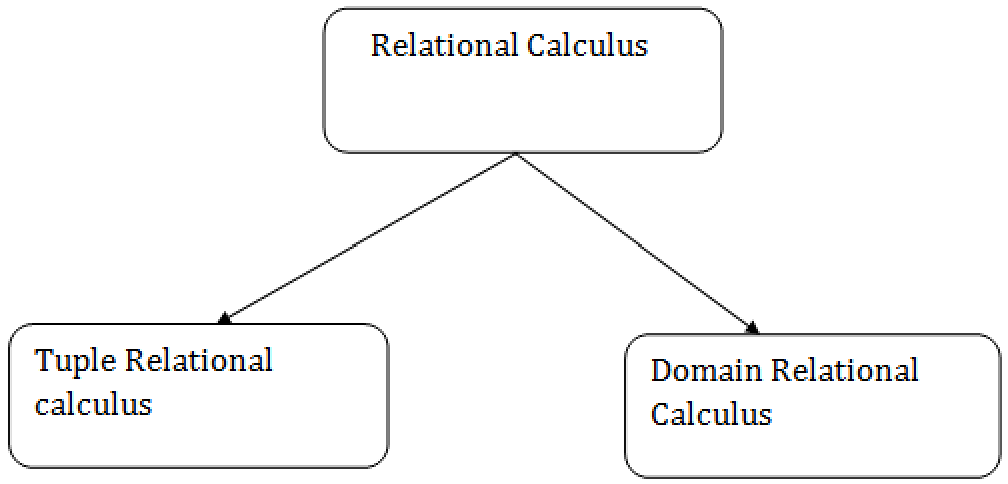
7. Rename Operation:
The rename operation is used to rename the output relation. It is denoted by **rho** (ρ).
Example: We can use the rename operator to rename STUDENT relation to STUDENT1.
ρ(STUDENT1, STUDENT)

Relational Calculus

Relational calculus is a non-procedural query language. In the non-procedural query language, the user is concerned with the details of how to obtain the end results.

The relational calculus tells what to do but never explains how to do.

Types of Relational calculus:



1. Tuple Relational Calculus (TRC)
The tuple relational calculus is specified to select the tuples in a relation. In TRC, filtering variable uses the tuples of a relation.
The result of the relation can have one or more tuples.
Notation:
{T | P (T)}   or {T | Condition (T)}
Where
T is the resulting tuples

P(T) is the condition used to fetch T.
For example:
{ T.name | Author(T) AND T.article = 'database' }
OUTPUT: This query selects the tuples from the AUTHOR relation. It returns a tuple with 'name' from Author who has written an article on 'database'.
TRC (tuple relation calculus) can be quantified. In TRC, we can use Existential (∃) and Universal Quantifiers (∀).
For example:
{ R| ∃T ∈ Authors(T.article='database' AND R.name=T.name)}
Output: This query will yield the same result as the previous one.

2. Domain Relational Calculus (DRC)
The second form of relation is known as Domain relational calculus. In domain relational calculus, filtering variable uses the domain of attributes.
Domain relational calculus uses the same operators as tuple calculus. It uses logical connectives ∧ (and), ∨ (or) and ¬ (not).
It uses Existential (∃) and Universal Quantifiers (∀) to bind the variable.
Notation:
{ a1, a2, a3, ..., an | P (a1, a2, a3, ... ,an)}
Where
a1, a2 are attributes
P stands for formula built by inner attributes
For example:
{< article, page, subject > | ∈ javatpoint ∧ subject = 'database'}
Output: This query will yield the article, page, and subject from the relational javatpoint, where the subject is a database.

Join Operations:

A Join operation combines related tuples from different relations, if and only if a given join condition is satisfied. It is denoted by ⋈.

Example:

EMPLOYEE

EMP_CODE	EMP_NAME
101	Stephan
102	Jack
103	Harry

SALARY

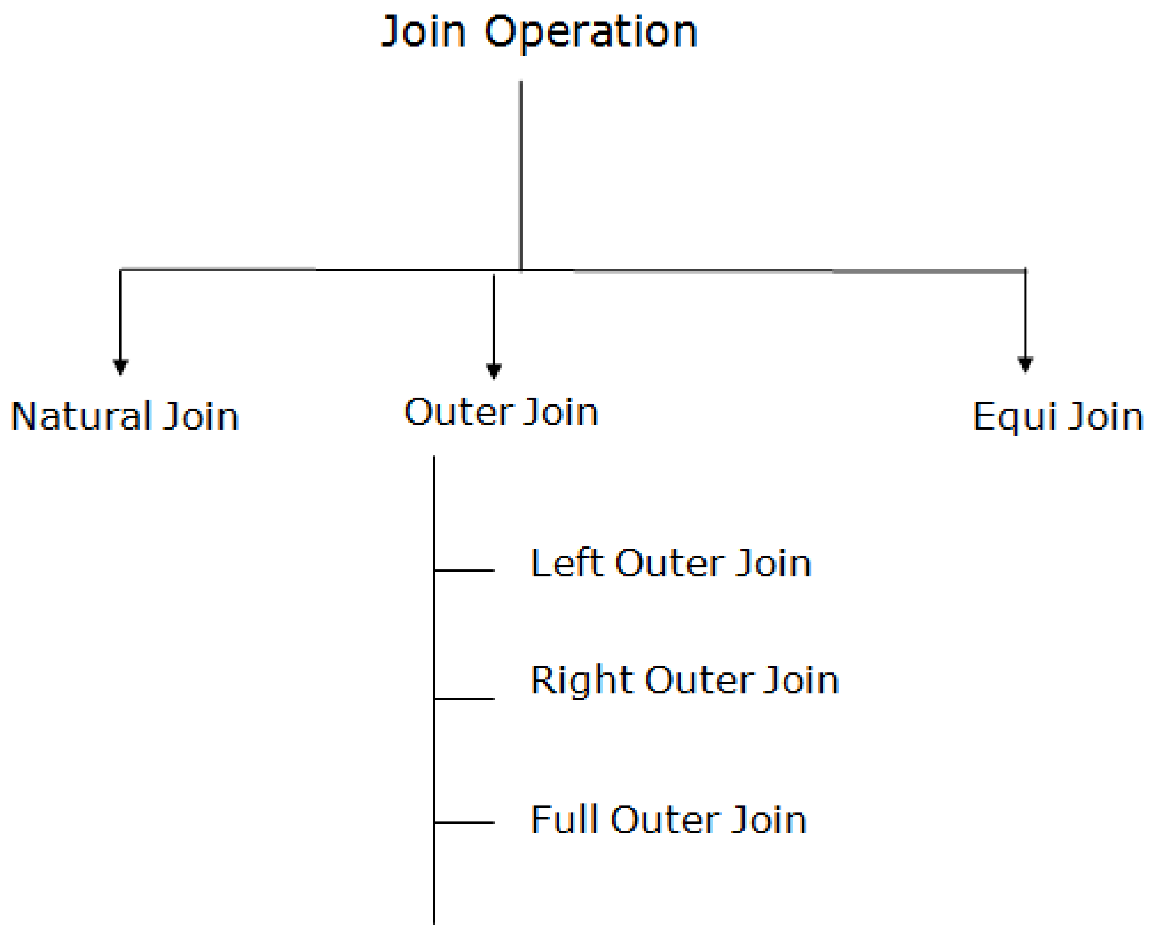
EMP_CODE	SALARY
101	50000
102	30000
103	25000

Operation: (EMPLOYEE ⋈ SALARY)

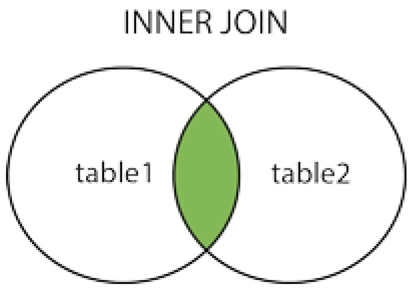
Result:

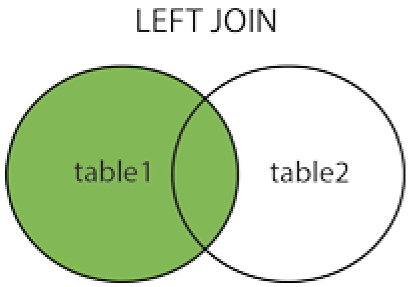
EMP_CODE	EMP_NAME	SALARY
101	Stephan	50000
102	Jack	30000
103	Harry	25000

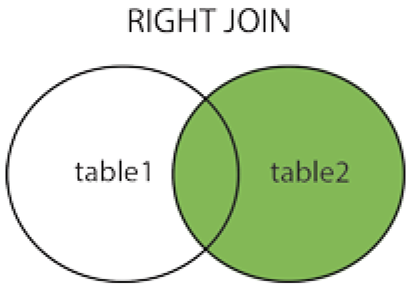
Types of Join operations:

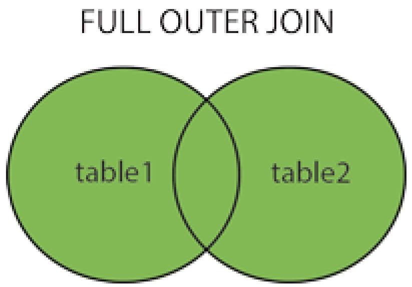

Join Operation
Natural Join
Outer Join
Equi Join
Left Outer Join
Right Outer Join
Full Outer Join

(INNER) JOIN: Returns records that have matching values in both tables
LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table









1. Natural Join:

A natural join is the set of tuples of all combinations in R and S that are equal on their common attribute names.

It is denoted by ⋈.

Example: Let's use the above EMPLOYEE table and SALARY table:

Input:

∏EMP_NAME, SALARY (EMPLOYEE ⋈ SALARY)

Output:

EMP_NAME	SALARY
Stephan	50000
Jack	30000
Harry	25000

2. Outer Join:

The outer join operation is an extension of the join operation. It is used to deal with missing information.

Example:

EMPLOYEE

EMP_NAME	STREET	CITY
Ram	Civil line	Mumbai
Shyam	Park street	Kolkata
Ravi	M.G. Street	Delhi
Hari	Nehru nagar	Hyderabad

FACT_WORKERS

EMP_NAME	BRANCH	SALARY
Ram	Infosys	10000
Shyam	Wipro	20000
Kuber	HCL	30000
Hari	TCS	50000

Input:
(EMPLOYEE ⋈ FACT_WORKERS)
Output:

EMP_NAME	STREET	CITY	BRANCH	SALARY
Ram	Civil line	Mumbai	Infosys	10000
Shyam	Park street	Kolkata	Wipro	20000
Hari	Nehru nagar	Hyderabad	TCS	50000

An outer join is basically of three types:
Left outer join
Right outer join
Full outer join
a. Left outer join:
Left outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.
In the left outer join, tuples in R have no matching tuples in S.
It is denoted by ⟕.
Example: Using the above EMPLOYEE table and FACT_WORKERS table
Input:

EMPLOYEE ⟕ FACT_WORKERS

EMP_NAME	STREET	CITY	BRANCH	SALARY
Ram	Civil line	Mumbai	Infosys	10000
Shyam	Park street	Kolkata	Wipro	20000
Hari	Nehru street	Hyderabad	TCS	50000
Ravi	M.G. Street	Delhi	NULL	NULL

b. Right outer join:

Right outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.

In right outer join, tuples in S have no matching tuples in R.

It is denoted by ⟖.

Example: Using the above EMPLOYEE table and FACT_WORKERS Relation

Input:

EMPLOYEE ⟖ FACT_WORKERS

Output:

EMP_NAME	BRANCH	SALARY	STREET	CITY
Ram	Infosys	10000	Civil line	Mumbai
Shyam	Wipro	20000	Park street	Kolkata
Hari	TCS	50000	Nehru street	Hyderabad
Kuber	HCL	30000	NULL	NULL

c. Full outer join:
Full outer join is like a left or right join except that it contains all rows from both tables.
In full outer join, tuples in R that have no matching tuples in S and tuples in S that have no matching tuples in R in their common attribute name.
It is denoted by ⟗.
Example: Using the above EMPLOYEE table and FACT_WORKERS table
Input:
EMPLOYEE ⟗ FACT_WORKERS
Output:

EMP_NAME	STREET	CITY	BRANCH	SALARY
Ram	Civil line	Mumbai	Infosys	10000
Shyam	Park street	Kolkata	Wipro	20000
Hari	Nehru street	Hyderabad	TCS	50000
Ravi	M.G. Street	Delhi	NULL	NULL
Kuber	NULL	NULL	HCL	30000

3. Equi join:
It is also known as an inner join. It is the most common join. It is based on matched data as per the equality condition. The equi join uses the comparison operator(=).
Example:
CUSTOMER RELATION

CLASS_ID	NAME
1	John
2	Harry
3	Jackson

PRODUCT

PRODUCT_ID	CITY
1	Delhi
2	Mumbai
3	Noida

Input:

CUSTOMER ⋈ PRODUCT

Output:

CLASS_ID	NAME	PRODUCT_ID	CITY
1	John	1	Delhi
2	Harry	2	Mumbai
3	Harry	3	Noida

Exercise:

Insert the missing parts in the `JOIN` clause to join the two tables `Orders` and `Customers`, using the `CustomerID` field in both tables as the relationship between the two tables.

```
SELECT *
FROM Orders
LEFT JOIN Customers
ON Orders.CustomerID=Customers.CustomerID;
```

Exercise:

Choose the correct `JOIN` clause to select all records from the two tables where there is a match in both tables.

```
SELECT *
FROM Orders
INNER JOIN Customers
ON Orders.CustomerID=Customers.CustomerID;
```

The SQL JOIN syntax

The general syntax is

```
SELECT column-names
  FROM table-name1 JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

The general syntax with INNER is:

```
SELECT column-names
  FROM table-name1 INNER JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

SQL JOIN Examples

Problem: List all orders with customer information

ORDER	
Id	🔑
OrderDate	
OrderNumber	
CustomerId	
TotalAmount	

CUSTOMER	
Id	🔑
FirstName	
LastName	
City	
Country	
Phone	

```sql
SELECT OrderNumber, TotalAmount, FirstName, LastName, City, Country
  FROM [Order] JOIN Customer
    ON [Order].CustomerId = Customer.Id
```

In this example using table Aliases for [Order] and Customer might have been useful.

Result: 830 records.

Problem: List all orders with product names, quantities, and prices

ORDER
Id 🔑
OrderDate
OrderNumber
CustomerId
TotalAmount

ORDERITEM
Id 🔑
OrderId
ProductId
UnitPrice
Quantity

PRODUCT
Id 🔑
ProductName
SupplierId
UnitPrice
Package
IsDiscontinued

```
SELECT O.OrderNumber, CONVERT(date,O.OrderDate) AS Date,
       P.ProductName, I.Quantity, I.UnitPrice
  FROM [Order] O
  JOIN OrderItem I ON O.Id = I.OrderId
  JOIN Product P ON P.Id = I.ProductId
ORDER BY O.OrderNumber
```

The SQL LEFT JOIN syntax

The general LEFT JOIN syntax is

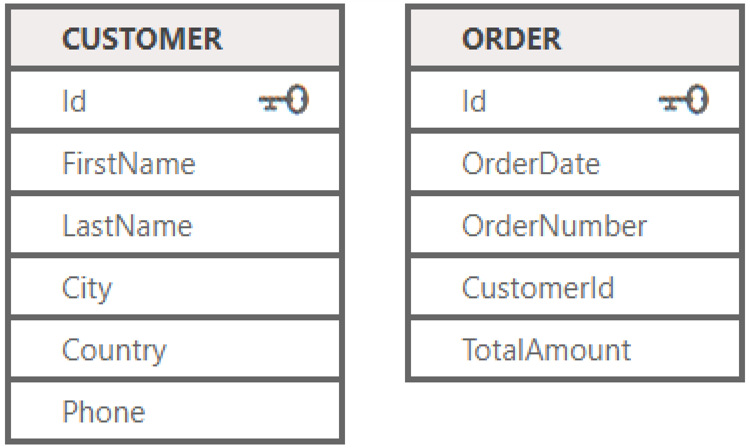
```
SELECT column-names
  FROM table-name1 LEFT JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

The general LEFT OUTER JOIN syntax is

```
SELECT column-names
  FROM table-name1 LEFT OUTER JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

SQL LEFT JOIN Example

Problem: List all customers and the total amount they spent irrespective whether they placed any orders or not.



```sql
SELECT OrderNumber, TotalAmount, FirstName, LastName, City, Country
  FROM Customer C LEFT JOIN [Order] O
    ON O.CustomerId = C.Id
 ORDER BY TotalAmount
```

The SQL RIGHT JOIN syntax

The general syntax is

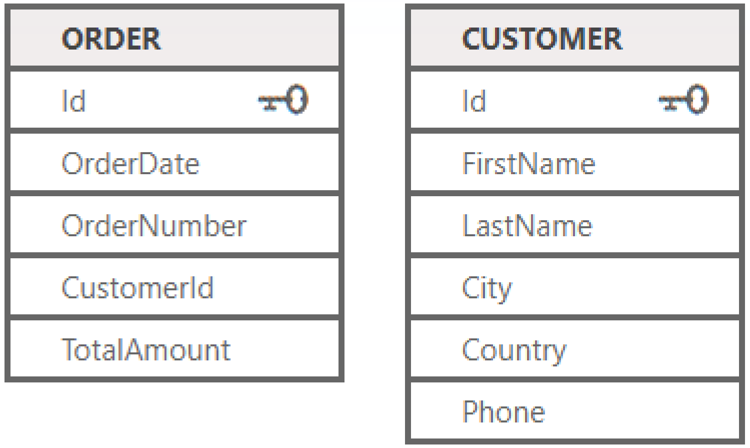
```
SELECT column-names
  FROM table-name1 RIGHT JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

The general RIGHT OUTER JOIN syntax is:

```
SELECT column-names
  FROM table-name1 RIGHT OUTER JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

SQL RIGHT JOIN Examples

Problem: List customers that have not placed orders



```sql
SELECT TotalAmount, FirstName, LastName, City, Country
  FROM [Order] O RIGHT JOIN Customer C
    ON O.CustomerId = C.Id
WHERE TotalAmount IS NULL
```

Problem: List all customers (with or without orders) and a count of the orders that include a *2kg box with Konbu* (product with Id = 13).

ORDERITEM	
Id	🔑
OrderId	
ProductId	
UnitPrice	
Quantity	

ORDER	
Id	🔑
OrderDate	
OrderNumber	
CustomerId	
TotalAmount	

CUSTOMER	
Id	🔑
FirstName	
LastName	
City	
Country	
Phone	

```
SELECT DISTINCT (C.Id), Firstname, LastName, COUNT(O.Id) AS Orders
  FROM [Order] O
  JOIN OrderItem I ON O.Id = I.OrderId AND I.ProductId = 13
 RIGHT JOIN Customer C ON C.Id = O.CustomerId
 GROUP BY C.Id, FirstName, LastName
 ORDER BY COUNT(O.Id)
```

The SQL FULL JOIN syntax

The general syntax is:

```
SELECT column-names
  FROM table-name1 FULL JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

The general FULL OUTER JOIN syntax is:

```
    SELECT column-names
  FROM table-name1 FULL OUTER JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

SQL FULL JOIN Examples

Problem: Match all customers and suppliers by country

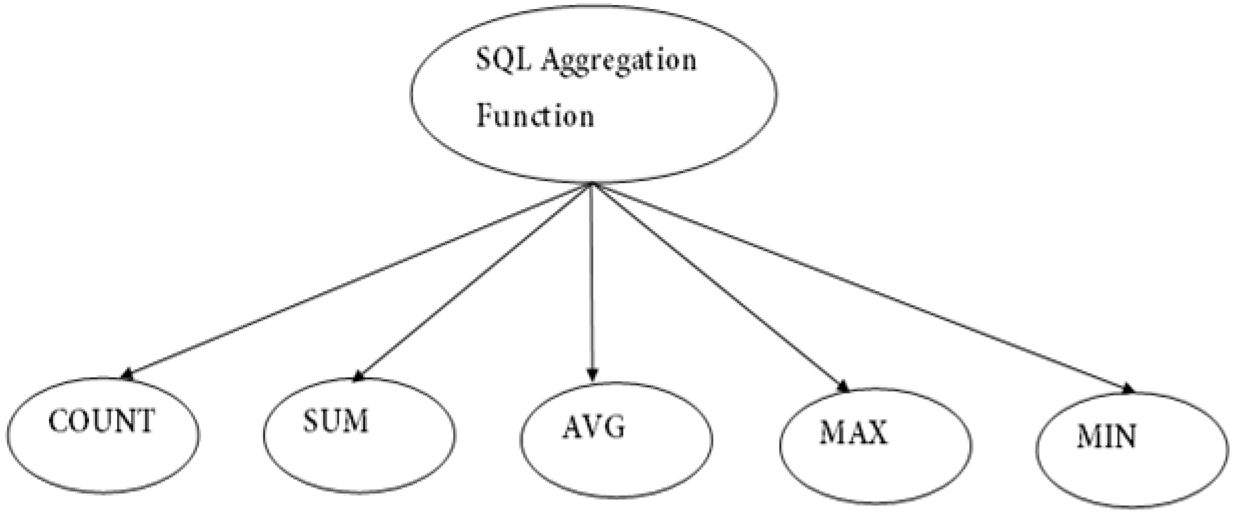
CUSTOMER
Id
FirstName
LastName
City
Country
Phone

SUPPLIER
Id
CompanyName
ContactName
City
Country
Phone
Fax

```sql
SELECT C.FirstName, C.LastName, C.Country AS CustomerCountry,
       S.Country AS SupplierCountry, S.CompanyName
  FROM Customer C FULL JOIN Supplier S
    ON C.Country = S.Country
 ORDER BY C.Country, S.Country
```

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
(Estd U/S(3) of UGC Act, 1956) Accredited with Grade "A" by NAAC

SQL Aggregate Functions
SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
It is also used to summarize the data.
Types of SQL Aggregation Function



1. COUNT FUNCTION

COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.

COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table. COUNT(*) considers duplicate and Null.

Syntax

COUNT(*)

or

COUNT([ALL|DISTINCT] expression)

Sample table:

PRODUCT_MAST

PRODUCT	COMPANY	QTY	RATE	COST
Item1	Com1	2	10	20
Item2	Com2	3	25	75
Item3	Com1	2	30	60
Item4	Com3	5	10	50
Item5	Com2	2	20	40
Item6	Cpm1	3	25	75
Item7	Com1	5	30	150
Item8	Com1	3	10	30
Item9	Com2	2	25	50
Item10	Com3	4	30	120

Example: COUNT()

```
1.SELECT COUNT(*)
2.FROM PRODUCT_MAST;
```

Output:

```
10
```

Example: COUNT with WHERE

```
SELECT COUNT(*)
FROM PRODUCT_MAST;
WHERE RATE>=20;
```

Output:

```
7
```

Example: COUNT() with DISTINCT

```
SELECT COUNT(DISTINCT COMPANY)
FROM PRODUCT_MAST;
```

Output:

```
3
```

Example: COUNT() with GROUP BY

```
SELECT COMPANY, COUNT(*)
FROM PRODUCT_MAST
GROUP BY COMPANY;
```

Output:

```
Com1   5
Com2   3
Com3   2
```

Example: COUNT() with HAVING

```
SELECT COMPANY, COUNT(*)
FROM PRODUCT_MAST
GROUP BY COMPANY
HAVING COUNT(*)>2;
```

Output:

```
Com1 5
 Com2 3
```

2. SUM Function

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

Syntax

```
SUM()
or
SUM( [ALL|DISTINCT] expression )
```

Example: SUM()

```
SELECT SUM(COST)
FROM PRODUCT_MAST;
```

Output:

670

Example: SUM() with WHERE

```
SELECT SUM(COST)
FROM PRODUCT_MAST
WHERE QTY>3;
```

Output: 320

Example: SUM() with GROUP BY

```
SELECT SUM(COST)
FROM PRODUCT_MAST
WHERE QTY>3
GROUP BY COMPANY;
```

Output: 150, 170

Example: SUM() with HAVING

```
SELECT COMPANY, SUM(COST)
FROM PRODUCT_MAST
GROUP BY COMPANY
HAVING SUM(COST)>=170;
```

Output: 335, 170

3. AVG function

The AVG function is used to calculate the average value of the numeric type.
AVG function returns the average of all non-Null values.

Syntax

AVG()

or

AVG([ALL|DISTINCT] expression)

Example:

```
SELECT AVG(COST)
FROM PRODUCT_MAST;
```

Output: 67.00

4. MAX Function

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

Syntax

MAX()

or

MAX([ALL|DISTINCT] expression)

Example:

```
SELECT MAX(RATE)
FROM PRODUCT_MAST;
```

Out put: 30

5. MIN Function

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

Syntax

MIN()

or

MIN([ALL|DISTINCT] expression)

Example:

SELECT MIN(RATE)
FROM PRODUCT_MAST;

Output: 10