UNIT-II

Syllabus: Distributed Database Design Framework -- Database Fragmentation Design -- Fragments Allocation -- Equivalence Transformations for Queries -- Global Queries -- Distributed Grouping and Aggregate Function Evaluation -- Parametric Queries -- Access Strategies Optimization -- Query Optimization Framework -- Join Queries -- General Queries.

TOPIC:

A FRAMEWORK FOR DISTRIBUTED DATABASE DESIGN

1. Designing the “conceptual schema” which describes the integrated database (i.e., all the data which are used by the database applications).
2. Designing the “physical database” i.e., mapping the conceptual schema to storage areas and determining appropriate access methods.
3. Designing the fragmentation, i.e., determining how global relations are subdivided into horizontal, vertical, or mixed fragments.
4. Designing the allocation of fragments, i.e., determining how fragments are mapped to physical images; in this way, also the replication of fragments is determined.

->Objectives of distributed database:

The concept of distributed database management system is to store the data to different locations. There are number of goals to distribute the data.

Reliability

Distributed database management system is more reliable if any of the connected computer system is failed to perform then other computers can complete the task without delay.

Availability

If a computer server fails to perform and stopped working in any time the other computer servers can perform the task as requested.

Performance

The data and information can be accessed from distributed database management systems from different locations. It is very easy to handle and maintain.

Low communication cost

Data and information is stored locally in distributed database management system. Its communication cost and data manipulation become easy and less costly.

Modular development

Modulation in distributed database management system is so easy. More systems can be manipulated and installed by just installing and connecting with the distributed database system with no interruption and failure.

Better response

All the computer system are installed centrally and can process any query with in shortest possible time. It provides faster response due to centralized processing of database management systems.

Data recovery

Data can be easily recovered in distributed database management systems.

->TOP-DOWN AND BOTTOM DOWN APPROACHES

Top-Down Design Model:

In the top-down model, an overview of the system is formulated without going into detail for any part of it. Each part of it then refined into more details, defining it in yet more details until the entire specification is detailed enough to validate the model. if we glance at a haul as a full, it's going to appear not possible as a result of it's so complicated For example: Writing a University system program, writing a word processor. Complicated issues may be resolved victimization high down style, conjointly referred to as Stepwise refinement where,

1. We break the problem into parts,
2. Then break the parts into parts soon and now each of parts will be easy to do.

Advantages:

- Breaking problems into parts help us to identify what needs to be done.
- At each step of refinement, new parts will become less complex and therefore easier to solve.
- Parts of the solution may turn out to be reusable.
- Breaking problems into parts allows more than one person to solve the problem.

Bottom-Up Design Model:

In this design, individual parts of the system are specified in detail. The parts are linked to form larger components, which are in turn linked until a complete system is formed. Object-oriented language such as C++ or java uses a bottom-up approach where each object is identified first.

Advantage:

Make decisions about reusable low-level utilities then decide how there will be put together to create high-level construct. ,

TOPIC:
DESIGN OF DATABASE FRAGMENTATION

Data Fragmentation

The process of dividing the database into smaller multiple parts or sub−tables is called fragmentation. The smaller parts or sub−tables are called fragments and are stored at different locations. Data fragmentation should be done in a way that the reconstruction of the original parent database from the fragments is possible. The restoration can be done using UNION or JOIN operations.

Database fragmentation is of three types: Horizontal fragmentation, Vertical fragmentation, and Mixed or Hybrid fragmentation.

Horizontal Fragmentation

It divides a table horizontally into a group of rows to create multiple fragments or subsets of a table. These fragments can then be assigned to different sites in the database. Reconstruction is done using UNION or JOIN operations. In relational algebra, it is represented as σp(T) for any given table(T).

Example

In this example, we are going to see how the horizontal fragmentation looks in a table.

Input :

STUDENT

```
id      name    age     salary
1       aman    21      20000
2       naman   22      25000
3       raman   23      35000
4       sonam   24      36000
```

Example

```
SELECT * FROM student WHERE salary<35000;
SELECT * FROM student WHERE salary>35000;
```

Output

id	name	age	salary
1	aman	21	20000
2	naman	22	25000
id	name	age	salary
4	soman	24	36000

There are three types of Horizontal fragmentation: Primary, Derived, and Complete Horizontal Fragmentation

A: Primary Horizontal Fragmentation: It is a process of segmenting a single table in a row−wise manner using a set of conditions.

Example

This example shows how the Select statement is used with a condition to provide output.

```
SELECT * FROM student SALARY<30000;
```

Output

id	name	age	salary
1	aman	21	20000
2	naman	22	25000

B: Derived Horizontal Fragmentation: Fragmentation that is being derived from primary relation.

Example

This example shows how the Select statement is used with the where clause to provide output.

```
SELECT * FROM student WHERE age=21 AND salary<30000;
```

Output

id	name	age	salary
1	aman	21	20000

C: Complete horizontal fragmentation: It derives a set of horizontal fragments to make the table have at least one partition.

Vertical Fragmentation

It divides a table vertically into a group of columns to create multiple fragments or subsets of a table. These fragments can then be assigned to different sites in the database. Reconstruction is done using full outer join operation.

Example

This example shows how the Select statement is used to do the fragmentation and to provide the output.

Input Table :

STUDENT

id	name	age	salary
1	aman	21	20000
2	naman	22	25000
3	raman	23	35000
4	sonam	24	36000

Example

```
SELECT * FROM name;#fragmentation 1
SELECT * FROM id, age;#fragmentation 2
```

Output

name

aman
naman
raman
sonam
age
21
22
23
24

Mixed or Hybrid Fragmentation

It is done by performing both horizontal and vertical partitioning together. It is a group of rows and columns in relation.

Example

This example shows how the Select statement is used with the where clause to provide the output.

```
SELECT * FROM name WHERE age=22;
```

Output

```
name    age
naman   22
```

Data Replication

Data replication means a replica is made i. e. data is copied at multiple locations to improve the availability of data. It is used to remove inconsistency between the same data which result in a distributed database so that users can do their task without interrupting the work of other users.

Types of data replication :

Transactional Replication

It makes a full copy of the database along with the changed data. Transactional consistency is guaranteed because the

order of data is the same when copied from publisher to subscriber database. It is used in server−to−server environments by consistently and accurately replicating changes in the database.

Snapshot Replication

It is the simplest type that distributes data exactly as it appears at a particular moment regardless of any updates in data. It copies the 'snapshot' of the data. It is useful when the database changes infrequently. It is slower to Transactional Replication because the data is sent in bulk from one end to another. It is generally used in cases where subscribers do not need the updated data and are operating in read−only mode.

Merge Replication

It combines data from several databases into a single database. It is the most complex type of replication because both the publisher and subscriber can do database changes. It is used in a server−to−client environment and has changes sent from one publisher to multiple subscribers.

Data Allocation

It is the process to decide where exactly you want to store the data in the database. Also involves the decision as to which data type of data has to be stored at what particular location. Three main types of data allocation are centralized, partitioned, and replicated.

Centralises: Entire database is stored at a single site. No data distribution occurs

Partitioned: The database gets divided into different fragments which are stored at several sites.

Replicated: Copies of the database are stored at different locations to access the data.

Conclusion

This article consists of three parts. The first one has data fragmentation which divides data into sub-tables and stores them at different locations. Types of data fragmentation are horizontal Fragmentation which divides the table horizontally into groups of rows, followed by vertical fragmentation which divides the table vertically into groups of columns and last one is mixed fragmentation which is done by performing both horizontal and vertical partitioning together. The second part comes with data replication in which data is copied at multiple locations. Types of replication are Transactional Replication which makes a full copy of the database along with the changes that occur, followed by Snapshot Replication which copies the snapshot of data to distribute and last one is Merge Replication which combines data into a single database. third part comes with Data Allocation which tells us where to store the data.

TOPIC:

FRAGMENTS ALLOCATION

Each fragment or each copy of a fragment is stored at a particular site in the distributed system with an "optimal" distribution. This process is called data distribution (or data allocation). The choice of sites and the degree of replication depend on the performance and availability goals of the system and on the types and frequencies of transactions submitted at each site.

Example: If high availability is required, transactions can be submitted at any site, and most transactions are retrieved only, a fully replicated database is a good choice. However, if certain transactions that access particular parts of the

database are mostly submitted at a particular site, the corresponding set of fragments can be allocated at that site only. Data that is accessed at multiple sites can be replicated at those sites. If any updates are performed, it may be useful to limit replication. Finding an optimal or even a good solution to distributed data allocation is a complex optimization problem.

There are four alternative strategies regarding the placement of data: centralized, fragmented, complete replication, and selective replication.

1. Centralized

This strategy consists of a single database and DBMS stored at one site with users distributed across the network. The locality of reference is at its lowest as all sites, except the central site, have to use the network for all data access. This also means that communication costs are high. Reliability and availability are low, as a failure of the central site results in the loss of the entire database system.

2.Fragmented (or Partitioned)

This strategy partitions the database into disjoint fragments, with each fragment assigned to one site. If data items are located at the site where they are used most frequently, the locality of reference is high. As there is no replication, storage costs are low; similarly, reliability and availability are low, although they are higher than in the centralized case, as the failure of a site results in the loss of only that site's data. Performance should be good and communications cost low if the distribution is designed properly.

3.Complete Replication

This strategy consists of maintaining a complete copy of the database at each site. Therefore, the locality of reference, reliability, availability, and performance is maximized. However, storage costs and communication costs for updates are the most expensive. To overcome some of these problems, snapshots are sometimes used. A snapshot is a copy of the data at a given time. The copies are updated periodically-for example, hourly or weekly-so they may not be always up to date. Snapshots are also sometimes used to implement views in a distributed database to improve the time it takes to perform a database operation on a view.

4.Selective Replication

This strategy is a combination of fragmentation, replication, and centralization. Some data items are fragmented to achieve a high locality of reference, and others that are used at many sites and are not frequently updated are replicated; otherwise, the data items are centralized. The objective of this strategy is to have all the advantages of the other approaches but none of the disadvantages. This is the most commonly used strategy, because of its flexibility.

TOPIC:

DISTRIBUTED GROUPING AND AGGREGATE FUNCTION

Overview

Using Group Functions

Using Group-By Clause

Using Having-Clause

Nesting Group Functions

Overview

Group functions are mathematical functions to operate on sets of rows to give one result per set. The types of group functions (also called aggregate functions) are:

AVG, that calculates the average of the specified columns in a set of rows,

COUNT, calculating the number of rows in a set.

MAX, calculating the maximum,

MIN, calculating the minimum,

STDDEV, calculating the standard deviation,

SUM, calculating the sum,

VARIANCE, calculating the variance.

The general syntax for using Group functions is :

```
SELECT <column, ...>, group_function(column)
FROM <table>
WHERE <condition>
[GROUP BY <column>]
[ORDER BY <column>]
```

Note that the column on which the group function is applied must exist in the SELECT column list.

For example, the following selects the maximum salary from the employee table.

SQL> SELECT dept, MAX(sal)

FROM employee;

Using Group Functions

You can use AVG and SUM for numeric data. MIN and MAX can be applied on any data type including numbers, dates and strings. The order is defined accordingly.

Group functions ignore the NULL values in the column. To enforce the group functions ti include the NULL value, use NVL function.

For example, the following statement forces to take the average for all salary values even including 0 (although it is an odd case, let's assume this is possible.)

```
SQL> SELECT dept, AVG(NVL(sal,0))
    FROM employee;
```

Using Group-By Clause

Note that ALL columns in the SELECT list that are not in group functions must be in the GROUP-By clause.

For example,

```
SQL> SELECT dept, job, MAX(sal)
     FROM employee
     GROUP BY dept, job;
```

The following statement is thus illegal, because column dept is not in an aggregate function and is not in GROUP-BY clause.

```
SQL> SELECT dept, MAX(sal)
     FROM employee;
```

The correct statement which selects the maximum salary for each department, should be:

```
SQL> SELECT dept, MAX(sal)
     FROM employee
     GROUP BY dept;
```

Using Having-Clause

To exclude some group results, use Having-clause.

For example,

```
SQL> SELECT dept, MAX(sal)
     FROM employee
     GROUP BY dept
    HAVING MAX(sal) > 2000;
```

Note that you CANNOT use WHERE clause to restrict groups. For example, the following is illegal:

```
SQL> SELECT dept, MAX(sal)
     FROM employee
     WHERE MAX(sal) > 2000
     GROUP BY dept;
```

This is because the group function cannot be used in WHERE clause.

Nesting Group Functions

You can nest group function calls. For example, the following example to show the maximum of the average salary of all departments,

```
SQL> SELECT MAX(AVE(sal))
    FROM emp
    GROUP BY dept_no;
```

TOPIC:

Parametric Queries

What is a Parametric Query?

A parametric query is a type of query that involves parameters or variables. Instead of specifying exact values directly in the query, you use placeholders that get replaced with actual values during query execution.

These placeholders allow for flexibility and reusability. Parametric queries are commonly used in various database systems, including DDBMS.

Parametric Queries in DDBMS:

- In a distributed environment, parametric queries play a crucial role in optimizing performance and minimizing data transfer across sites.
- Here's how they work:
- Parameterized Fragments: In a DDBMS, data is distributed across multiple sites (servers). Each site holds a fragment of the overall data.
- When executing a query, the system identifies the relevant fragments based on the query parameters.
- For example, consider a distributed database storing customer information. A parametric query to retrieve customer details might include parameters like customer ID, date range, or location.

- The query optimizer determines which fragments need to be accessed based on these parameters.
- The actual values for the parameters are provided during query execution.
- This approach minimizes unnecessary data transfer and ensures that only relevant fragments are involved in processing the query.

Benefits of Parametric Queries:

Reduced Network Traffic: By fetching only the necessary data fragments, parametric queries minimize network communication.

Improved Performance: Query execution is more efficient because it focuses on relevant data.

Adaptability: Parametric queries can handle varying conditions without modifying the query structure.

Security: Parameters can be used to enforce access control and data privacy.

Example:

- Let's say we have a DDBMS with customer data distributed across three sites: Site A, Site B, and Site C.
- A parametric query to retrieve orders placed by a specific customer within a given date range might look like this:

```
SELECT order_id, order_date, total_amount
FROM orders
WHERE customer_id = :param_customer_id
  AND order_date BETWEEN :param_start_date AND
:param_end_date;
```

- During execution, the actual customer ID, start date, and end date are substituted for the placeholders (:param_customer_id, :param_start_date, and :param_end_date).

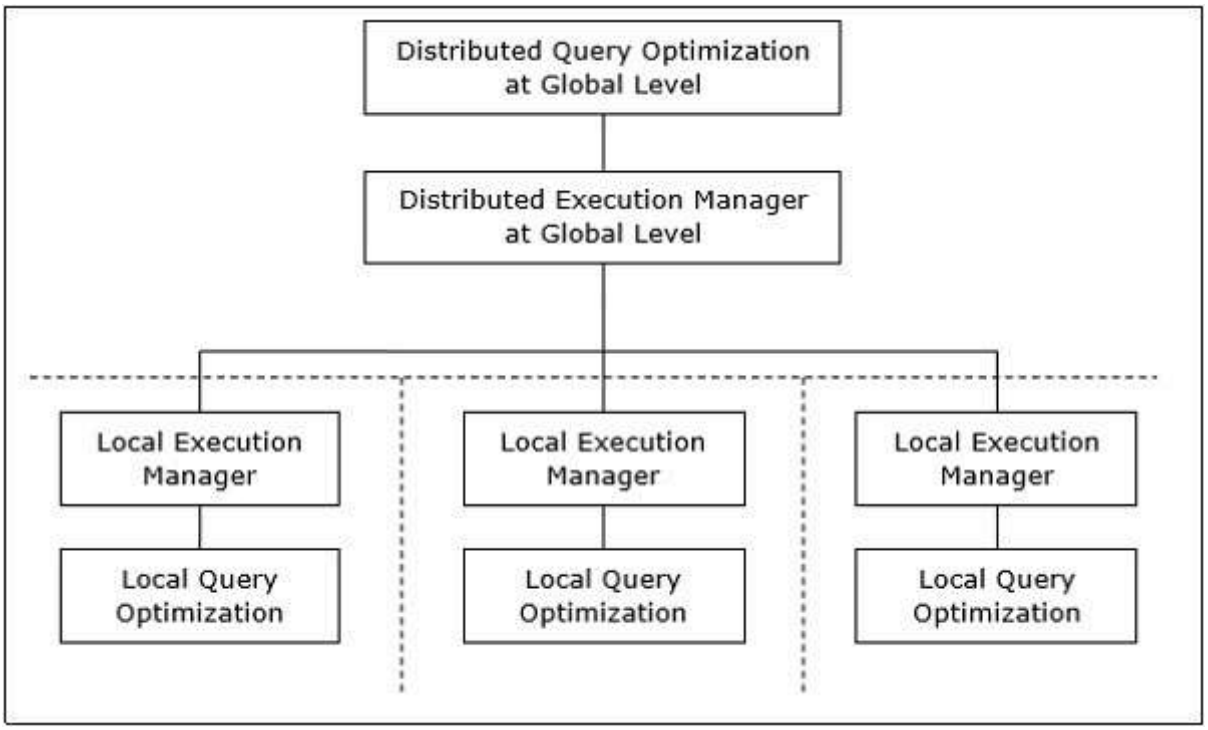
Query Optimization in Distributed Systems

This chapter discusses query optimization in distributed database system.

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 –



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

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.

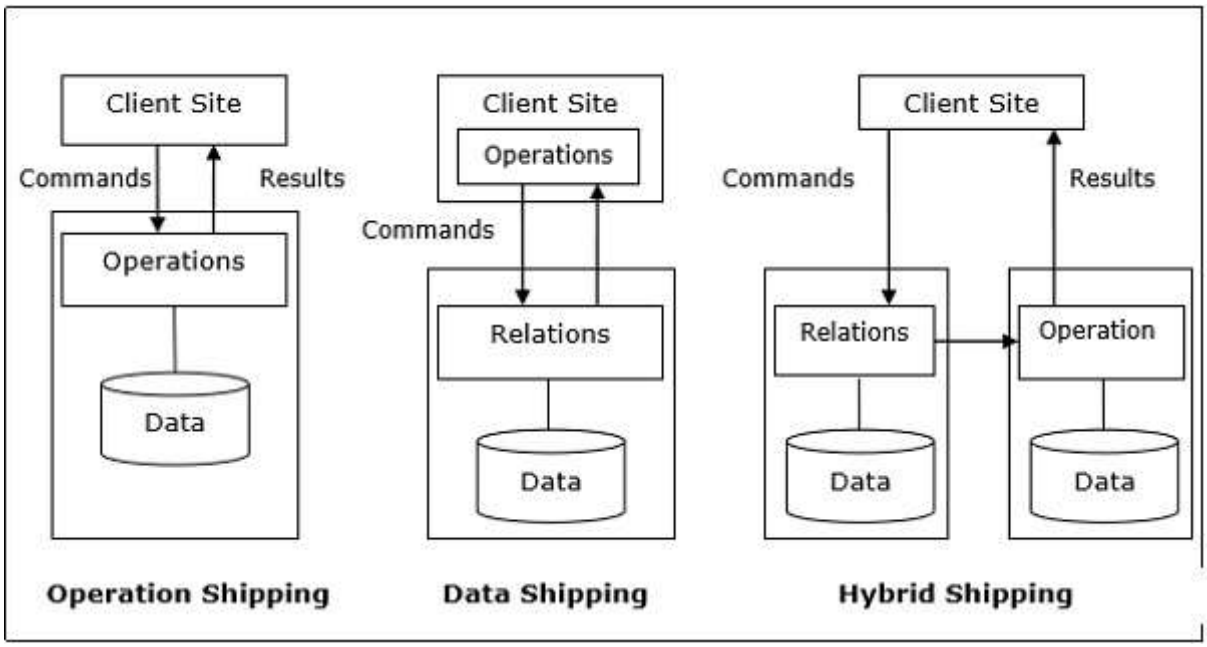
Optimal Utilization of Resources in the Distributed System

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.



Query Trading

In query trading algorithm for distributed database systems, the controlling/client site for a distributed query is called the buyer and the sites where the local queries execute are called sellers. The buyer formulates a number of alternatives for choosing sellers and for reconstructing the global results. The target of the buyer is to achieve the optimal cost.

The algorithm starts with the buyer assigning sub-queries to the seller sites. The optimal plan is created from local optimized query plans proposed by the sellers combined with the communication cost for reconstructing the final result. Once the global optimal plan is formulated, the query is executed.

Reduction of Solution Space of the Query

Optimal solution generally involves reduction of solution space so that the cost of query and data transfer is reduced. This can be achieved through a set of heuristic rules, just as heuristics in centralized systems.

Following are some of the rules –

- Perform selection and projection operations as early as possible. This reduces the data flow over communication network.
- Simplify operations on horizontal fragments by eliminating selection conditions which are not relevant to a particular site.

- In case of join and union operations comprising of fragments located in multiple sites, transfer fragmented data to the site where most of the data is present and perform operation there.
- Use semi-join operation to qualify tuples that are to be joined. This reduces the amount of data transfer which in turn reduces communication cost.
- Merge the common leaves and sub-trees in a distributed query tree.

TOPIC:

QUERY OPTIMIZATION FRAMEWORK:

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).

There are broadly two ways a query can be optimized:

1. Analyze and transform equivalent relational expressions: Try to minimize the tuple and column counts of the intermediate and final query processes (discussed here).
2. 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.
Here, we shall talk about generating minimal equivalent 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.**

   Explanation: Applying condition intersection is expensive. Instead, filter out tuples satisfying condition (inner selection) and then apply condition (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.**

   Explanation: condition is commutative in nature. This means, it does not matter whether we apply first or 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.**

   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**

     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 . 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**

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

5. **Theta Joins are commutative.**

   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**

     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**

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

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

   - **Equivalence 1**

     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**

  Explanation: This can be extended to two selection conditions, and , where Theta1 contains the attributes of only E1 and 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**

     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**

     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.**

   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.**

    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.**

    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.**

    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

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.

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

Optimized Query:
We can perform an “early selection”, hence the optimized query becomes:

Features :

Cost-Based Optimization: In cost-based optimization, the query optimizer estimates the cost of executing each possible query execution plan and selects the plan with the lowest cost. The cost of a query execution plan is usually measured in terms of the number of disk I/O operations or CPU cycles required to execute the plan.
Plan Space Exploration: The query optimizer explores the plan space, which is the set of all possible query execution plans, to find the optimal plan. This can be a complex and computationally expensive process, especially for complex queries involving multiple tables and join operations.
Query Rewriting: The query optimizer may rewrite the original query to create an equivalent query with a more efficient execution plan. For example, the optimizer may reorder the join operations or apply filter conditions earlier in the query execution plan to reduce the amount of data that needs to be processed.
Statistics: The query optimizer relies on statistics about the data in the tables being queried to estimate the cost of different query execution plans. These statistics may include information about the number of rows in each table, the distribution of values in each column, and the presence of indexes or constraints.
Index Selection: The query optimizer may select different indexes on the tables being queried to improve the efficiency of the query execution plan. This involves choosing the best index for each table based on the query’s filter and join conditions.
Caching: The query optimizer may cache the results of commonly executed queries to improve performance. This can reduce the need to execute the same query multiple times, especially in applications with frequent user queries.

Advantages:

Improved Query Performance: The main advantage of query optimization is that it can significantly improve the performance of queries. By selecting the most efficient query plan, the RDBMS can execute the query in the shortest possible time.
Cost Reduction: The optimization process reduces the cost of query execution by minimizing the use of system resources, such as CPU and memory, while still delivering the same results.
Query Tuning: Query optimization provides a way to fine-tune queries to make them more efficient, even in complex and large-scale databases.

Reduced Complexity: Query optimization makes the query execution process more efficient and reduces the complexity of the system. By using a streamlined approach, the RDBMS can improve its overall performance.

Disadvantages:

Increased Overhead: Query optimization requires additional processing time and system resources. This overhead may be significant for large databases and can impact system performance.
Limited Scope: Query optimization may not work for all queries, especially those that are ad-hoc and not well-defined. In such cases, the RDBMS may not be able to optimize the query, leading to suboptimal performance.
Algorithmic Complexity: The process of query optimization itself can be complex, especially for large and complex queries. This can result in a higher computational cost and may require specialized expertise to implement.
Maintenance: Query optimization requires ongoing maintenance to keep the query optimizer up to date with changes in the database schema, data distribution, and query workload.

1. All

2. **Database Management System (DBMS)**

What are the main challenges of query optimization in a distributed DBMS?

Query optimization is the process of finding the most efficient way to execute a query on a database. It involves analyzing various factors, such as the query structure, the data distribution, the available resources, and the network conditions. In a distributed database management system (DBMS), where the data is stored across multiple nodes or sites, query optimization becomes even more challenging. In this article, we will discuss some of the main challenges of query optimization in a distributed DBMS and how they can be addressed.

1) Data fragmentation

One of the challenges of query optimization in a distributed DBMS is data fragmentation, which refers to how the data is partitioned and allocated among the nodes. Data fragmentation can affect the performance of queries, as it may require more data transfers, joins, or replication. For example, if a query involves data that is horizontally fragmented, meaning that different rows of a table are stored at different nodes, it may need to access multiple nodes to retrieve the relevant data. On the other hand, if a query involves data that is vertically fragmented, meaning that different columns of a table are stored at different nodes, it may need to join the data from different nodes to reconstruct the table. Therefore, query optimization in a distributed DBMS needs to consider the data fragmentation scheme and choose the best node or nodes to process the query.

Add your perspective

2) Data localization

Another challenge of query optimization in a distributed DBMS is data localization, which refers to the degree to which the data needed for a query is available at the node where the query is initiated. Data localization can affect the performance of queries, as it may reduce or increase the amount of data transfers, communication, or synchronization. For example, if a query is initiated at a node that has most or all of the data needed for the query, it can be executed locally without much overhead. On the other hand, if a query is initiated at a node that has little or none of the data needed for the query, it may need to send requests to other nodes or fetch data from them, which can incur more costs and delays. Therefore, query optimization in a distributed DBMS

needs to consider the data localization factor and choose the best node or nodes to initiate the query.

3) Data replication

A third challenge of query optimization in a distributed DBMS is data replication, which refers to the process of creating and maintaining copies of data at different nodes. Data replication can improve the availability, reliability, and scalability of the distributed DBMS, as it can provide backup, load balancing, and fault tolerance. However, data replication can also introduce complexity and overhead for query optimization, as it may create inconsistency, redundancy, or conflicts. For example, if a query involves data that is replicated at multiple nodes, it may need to decide which copy of the data to use, how to ensure that the copies are consistent, and how to handle updates or transactions that affect the replicated data. Therefore, query optimization in a distributed DBMS needs to consider the data replication strategy and choose the best node or nodes to access or update the data.

Add your perspective

4) Network heterogeneity

A fourth challenge of query optimization in a distributed DBMS is network heterogeneity, which refers to the variation in the network characteristics among the nodes. Network heterogeneity can affect the performance of queries, as it may cause different levels of latency, bandwidth, reliability, or congestion. For example, if a query involves data that is stored at nodes that have different network speeds, distances, or qualities, it may need to account for the network delays, costs, or failures. Therefore, query optimization in a distributed DBMS needs to consider the network heterogeneity

factor and choose the best node or nodes to communicate or transfer data.

5) Query decomposition and allocation

A fifth challenge of query optimization in a distributed DBMS is query decomposition and allocation, which refers to the process of breaking down a query into subqueries and assigning them to different nodes for execution. Query decomposition and allocation can improve the performance of queries, as it can exploit the parallelism, concurrency, and locality of the distributed DBMS. However, query decomposition and allocation can also pose challenges for query optimization, as it may involve trade-offs, dependencies, or coordination. For example, if a query is decomposed into subqueries that are allocated to different nodes, it may need to balance the workload, minimize the data transfers, and synchronize the results. Therefore, query optimization in a distributed DBMS needs to consider the query decomposition and allocation problem and choose the best subqueries and nodes to execute them.

TOPIC:

JOIN QUERIES AND GENERAL QUERIES

Query processing in a distributed database management system requires the transmission of data between the computers in a network. A distribution strategy for a query is the ordering of data transmissions and local data processing in a database system. Generally, a query in Distributed DBMS requires data from multiple sites, and this need for data from different sites is called the transmission of data that causes communication costs. Query processing in DBMS is different from query processing in centralized DBMS due to the communication cost of data transfer over the network. The transmission cost is low when sites are connected through high-speed Networks and is quite significant in other networks.

The process used to retrieve data from a database is called query processing. Several processes are involved in query processing to retrieve data from the database. The actions to be taken are:

- Costs (Transfer of data) of Distributed Query processing
- Using Semi join in Distributed Query processing

Costs (Transfer of Data) of Distributed Query Processing

In Distributed Query processing, the data transfer cost of distributed query processing means the cost of transferring intermediate files to other sites for processing and therefore the cost of transferring the ultimate result files to the location where that result is required. Let's say that a user sends a query to site S1, which requires data from its own and also from another site S2. Now, there are three strategies to process this query which are given below:

1. We can transfer the data from S2 to S1 and then process the query
2. We can transfer the data from S1 to S2 and then process the query
3. We can transfer the data from S1 and S2 to S3 and then process the query. So the choice depends on various factors like the size of relations and the results, the communication cost between different sites, and at which the site result will be utilized.

Commonly, the data transfer cost is calculated in terms of the size of the messages. By using the below formula, we can calculate the data transfer cost:

Data transfer cost = C * Size

Where C refers to the cost per byte of data transferring and Size is the no. of bytes transmitted.

Example: Consider the following table EMPLOYEE and DEPARTMENT.

Site1: **EMPLOYEE**

EID	NAME	SALARY	DID

EID- 10 bytes
SALARY- 20 bytes
DID- 10 bytes
Name- 20 bytes
Total records- 1000
Record Size- 60 bytes

Site2: **DEPARTMENT**

DID	DNAME

DID- 10 bytes
DName- 20 bytes
Total records- 50
Record Size- 30 bytes

Example:

1. Find the name of employees and their department names. Also, find the amount of data transfer to execute this query when the query is submitted to Site 3.

Answer: Considering the query is submitted at site 3 and neither of the two relations is an EMPLOYEE and the DEPARTMENT not available at site 3. So, to execute this query, we have three strategies:

- Transfer both the tables that are EMPLOYEE and DEPARTMENT at SITE 3 then join the tables there. The total cost in this is **1000 * 60 + 50 * 30 = 60,000 + 1500 = 61500** bytes.
- Transfer the table EMPLOYEE to SITE 2, join the table at SITE 2 and then transfer the result at SITE 3. The total cost in this is **60 * 1000 + 60 * 1000 = 120000** bytes since we have to transfer 1000 tuples having NAME and DNAME from site 1,
- Transfer the table DEPARTMENT to SITE 1, join the table at SITE 2 join the table at site1 and then transfer the result at site3. The total cost is **30 * 50 + 60 * 1000 = 61500** bytes since we have to transfer 1000 tuples having NAME and DNAME from site 1 to site 3 which is 60 bytes each.

Now, If the Optimisation criteria are to reduce the amount of data transfer, we can choose either 1 or 3 strategies from the above.

Using Semi-Join in Distributed Query Processing

The semi-join operation is used in distributed query processing to reduce the number of tuples in a table before transmitting it to another site. This reduction in the number of tuples reduces the number and the total size of the transmission ultimately reducing the total cost of data transfer. Let's say that we have two tables R1, R2 on Site S1, and S2. Now, we will forward the joining column of one table say R1 to the site where the other table say R2 is located. This column is joined with R2 at that site. The decision whether to reduce R1 or R2 can only be made after comparing the advantages of reducing R1 with that of reducing R2. Thus, semi-join is a well-organized solution to reduce the transfer of data in distributed query processing.

Example: Find the amount of data transferred to execute the same query given in the above example using a semi-join operation.

Answer: The following strategy can be used to execute the query.

- Select all (or Project) the attributes of the EMPLOYEE table at site 1 and then transfer them to site 3. For this, we will transfer NAME, DID(EMPLOYEE) and the size is 30 * 1000 = 30000 bytes.
- Transfer the table DEPARTMENT to site 3 and join the projected attributes of EMPLOYEE with this table. The size of the DEPARTMENT table is 30 * 50 = 1500

Applying the above scheme, the amount of data transferred to execute the query will be **30000 + 1500 = 31500** bytes.

Conclusion

In Conclusion, query processing in a distributed database management system (DBMS) is a complex procedure that tackles issues with transaction management, data dissemination, optimization, and fault tolerance. Distributed database systems'

performance, scalability, and dependability depend on effective concurrency management, optimization, and query decomposition techniques.