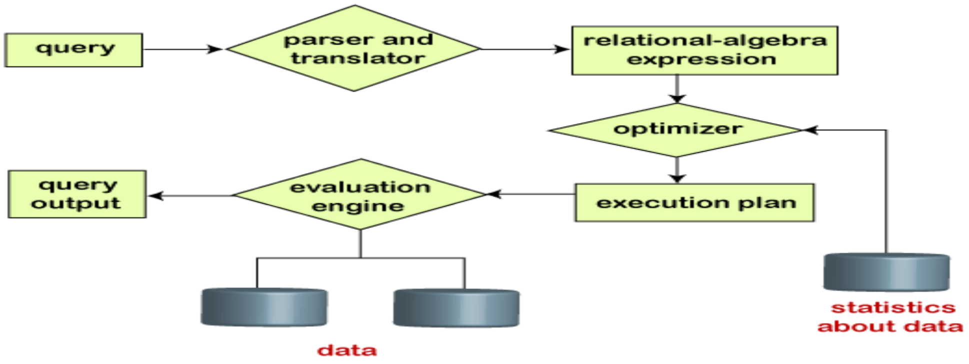
UNIT-IV

Query Processing in DBMS

Query Processing is the activity performed in extracting data from the database. In query processing, it takes various steps for fetching the data from the database. The steps involved are:

1. Parsing and translation
2. Optimization
3. Evaluation

The query processing works in the following way:



Steps in query processing

- **Step-1:**
  Parser: During parse call, the database performs the following checks- Syntax check, Semantic check and Shared pool check, after converting the query into relational algebra.
  Parser performs the following checks as (refer detailed diagram):

  1. **Syntax check** – concludes SQL syntactic validity.
     Example:
     SELECT * FORM employee

     Here error of wrong spelling of FROM is given by this check.

  2. **Semantic check** – determines whether the statement is meaningful or not. Example: query contains a table name which does not exist is checked by this check.
  3. **Shared Pool check** – Every query possess a hash code during its execution. So, this check determines existence of written hash code in shared pool if code exists in shared pool then database will not take additional steps for optimization and execution.

Hard Parse and Soft Parse

If there is a fresh query and its hash code does not exist in shared pool then that query has to pass through from the additional steps known as hard parsing otherwise if hash code exists then query does not passes through additional steps. It just passes directly to execution engine (refer detailed diagram). This is known as soft parsing.

Hard Parse includes following steps – Optimizer and Row source generation.

- **Step-2:**

  Optimizer: During optimization stage, database must perform a hard parse atleast for one unique DML statement and perform optimization during this parse. This database never optimizes DDL unless it includes a DML component such as subquery that require optimization.

  It is a process in which multiple query execution plan for satisfying a query are examined and most efficient query plan is satisfied for execution. Database catalog stores the execution plans and then optimizer passes the lowest cost plan for execution.

  Row Source Generation –

  The Row Source Generation is a software that receives a optimal execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database. The iterative plan is the binary program that when executes by the sql engine produces the result set.

- **Step-3:**

  Execution Engine: Finally runs the query and display the required result.

Measures of query cost:

Though a system can create multiple plans for a query, the chosen method should be the best of all. It can be done by comparing each possible plan in terms of their estimated cost. For calculating the net estimated cost of any plan, the cost of each operation within a plan should be determined and combined to get the net estimated cost of the query evaluation plan.

The cost estimation of a query evaluation plan is calculated in terms of various resources that include:

- Number of disk accesses
- Execution time taken by the CPU to execute a query
- Communication costs in distributed or parallel database systems.

To estimate the cost of a query evaluation plan, we use the number of blocks transferred from the disk, and the number of disks seeks. Suppose the disk has an average block access time of

t_S seconds and takes an average of t_T seconds to transfer x data blocks. The block access time is the sum of disk seeks time and rotational latency. It performs S seeks than the time taken will be **$b*t_T + S*t_S$** seconds. If t_T=0.1 ms, t_S =4 ms, the block size is 4 KB, and its transfer rate is 40 MB per second. With this, we can easily calculate the estimated cost of the given query evaluation plan.

Generally, for estimating the cost, we consider the worst case that could happen. The users assume that initially, the data is read from the disk only. But there must be a chance that the information is already present in the main memory. However, the users usually ignore this effect, and due to this, the actual cost of execution comes out less than the estimated value.

The response time, i.e., the time required to execute the plan, could be used for estimating the cost of the query evaluation plan. But due to the following reasons, it becomes difficult to calculate the response time without actually executing the query evaluation plan:

- When the query begins its execution, the response time becomes dependent on the contents stored in the buffer. But this information is difficult to retrieve when the query is in optimized mode, or it is not available also.
- When a system with multiple disks is present, the response time depends on an interrogation that in "what way accesses are distributed among the disks?". It is difficult to estimate without having detailed knowledge of the data layout present over the disk.
- Consequently, instead of minimizing the response time for any query evaluation plan, the optimizers finds it better to reduce the total **resource consumption** of the query plan. Thus to estimate the cost of a query evaluation plan, it is good to minimize the resources used for accessing the disk or use of the extra resources.

Selection Operation:

In the previous section, we understood that estimating the cost of a query plan should be done by measuring the total resource consumption.

In this section, we will understand how the selection operation is performed in the query execution plan.

Generally, the selection operation is performed by the file scan. **File scans** are the search algorithms that are used for locating and accessing the data. It is the lowest-level operator used in query processing.

Let's see how selection using a file scan is performed.

Selection using File scans and Indices

In RDBMS or relational database systems, the file scan reads a relation only if the whole relation is stored in one file only. When the selection operation is performed on a relation whose tuples are stored in one file, it uses the following algorithms:

- **Linear Search:** In a linear search, the system scans each record to test whether satisfying the given selection condition. For accessing the first block of a file, it needs an initial seek. If the blocks in the file are not stored in contiguous order, then it needs some extra seeks. However, linear search is the slowest algorithm used for searching, but it is applicable in all types of cases. This algorithm does not care about the nature of selection, availability of indices, or the file sequence. But other algorithms are not applicable in all types of cases.

Selection Operation with Indexes:

The index-based search algorithms are known as **Index scans**. Such index structures are known as **access paths**. These paths allow locating and accessing the data in the file. There are following algorithms that use the index in query processing:

- **Primary index, equality on a key:** We use the index to retrieve a single record that satisfies the equality condition for making the selection. The equality comparison is performed on the key attribute carrying a primary key.
- **Primary index, equality on non key:** The difference between equality on key and non key is that in this, we can fetch multiple records. We can fetch multiple records through a primary key when the selection criteria specify the equality comparison on a non key.
- **Secondary index, equality on key or non key:** The selection that specifies an equality condition can use the secondary index. Using secondary index strategy, we can either retrieve a single record when equality is on key or multiple records when the equality condition is on non key. When retrieving a single record, the time cost is equal to the primary index. In the case of multiple records, they may reside on different blocks. This results in one I/O operation per fetched record, and each I/O operation requires a seek and a block transfer.

Selection Operations with Comparisons:

For making any selection on the basis of a comparison in a relation, we can proceed it either by using the linear search or via indices in the following ways:

- **Primary index, comparison:** When the selection condition given by the user is a comparison, then we use a primary ordered index, such as the primary B^+-tree index. **For**

- **example**, when A attribute of a relation R compared with a given value v as A>v, then we use a primary index on A to directly retrieve the tuples. The file scan starts its search from the beginning till the end and outputs all those tuples that satisfy the given selection condition.
- **Secondary index, comparison:** The secondary ordered index is used for satisfying the selection operation that involves <, >, ≤, or ≥ In this, the files scan searches the blocks of the lowest-level index.
  (< ≤): In this case, it scans from the smallest value up to the given value v.
  (>, ≥): In this case, it scans from the given value v up to the maximum value.
  However, the use of the secondary index should be limited for selecting a few records. It is because such an index provides pointers to point each record, so users can easily fetch the record through the allocated pointers. Such retrieved records may require an I/O operation as records may be stored on different blocks of the file. So, if the number of fetched records is large, it becomes expensive with the secondary index.

Implementing Complex Selection Operations

Working on more complex selection involves three selection predicates known as Conjunction, Disjunction, and Negation.

Conjunction: A conjunctive selection is the selection having the form as:

$\boldsymbol{\sigma}_{\theta1\wedge\theta2\wedge\ldots\wedge\theta n}\,\mathbf{(r)}$

A conjunction is the intersection of all records that satisfies the above selection condition.

Disjunction: A disjunctive selection is the selection having the form as:

$\boldsymbol{\sigma}_{\theta1\vee\theta2\vee\ldots\vee\theta n}\,\mathbf{(r)}$

A disjunction is the union of all records that satisfies the given selection condition $\boldsymbol{\theta}_i$.

Negation: The result of a selection $\boldsymbol{\sigma}_{\neg\theta}\mathbf{(r)}$ is the set of tuples of given relation r where the selection condition evaluates to false. But nulls are not present, and this set is only the set of tuples in relation r that are not in $\boldsymbol{\sigma}_{\theta}\mathbf{(r)}$.

Using these discussed selection predicates, we can implement the selection operations by using the following algorithms:

- **Conjunctive selection using one index:** In such type of selection operation implementation, we initially determine if any access path is available for an attribute. If found one, then algorithms based on the index will work better. Further completion of the selection operation is done by testing that each selected records satisfy the remaining simple conditions. The cost of the selected algorithm provides the cost of this algorithm.
- **Conjunctive selection via Composite index:** A composite index is the one that is provided on multiple attributes. Such an index may be present for some conjunctive selections. If the given selection operation proves true on the equality condition on two or more attributes and a composite index is present on these combined attribute fields, then directly search the index. Such type of index evaluates the suitable index algorithms.
- **Conjunctive selection via the intersection of identifiers:** This implementation involves record pointers or record identifiers. It uses indices with the record pointers on those fields which are involved in the individual selection condition. It scans each index for pointers to tuples satisfying the individual condition. Therefore, the intersection of all the retrieved pointers is the set of pointers to the tuples that satisfies the conjunctive condition. The algorithm uses these pointers to fetch the actual records. However, in absence of indices on each individual condition, it tests the retrieved records for the other remaining conditions.
- **Disjunctive selection by the union of identifiers:** This algorithm scans those entire indexes for pointers to tuples that satisfy the individual condition. But only if access paths are available on all disjunctive selection conditions. Therefore, the union of all fetched records provides pointers sets to all those tuples which satisfy or prove the disjunctive condition. Further, it makes use of pointers for fetching the actual records. Somehow, if the access path is not present for anyone condition, we need to use a linear search to find those tuples that satisfy the condition. Thus, it is good to use a linear search for determining such tests.

SORTING:

Till now, we saw that sorting is an important term in any database system. It means arranging the data either in ascending or descending order. We use sorting not only for generating a sequenced output but also for satisfying conditions of various database algorithms. In query processing, the sorting method is used for performing various relational operations such as joins, etc. efficiently.

But the need is to provide a sorted input value to the system. For sorting any relation, we have to build an index on the sort key and use that index for reading the relation in sorted order. However, using an index, we sort the relation logically, not physically. Thus, sorting is performed for cases that include:

Case 1: Relations that are having either small or medium size than main memory.

Case 2: Relations having a size larger than the memory size.

In Case 1, the small or medium size relations do not exceed the size of the main memory. So, we can fit them in memory. So, we can use standard sorting methods such as quicksort, merge sort, etc., to do so.

For Case 2, the standard algorithms do not work properly. Thus, for such relations whose size exceeds the memory size, we use the External Sort-Merge algorithm.

The sorting of relations which do not fit in the memory because their size is larger than the memory size. Such type of sorting is known as **External Sorting**. As a result, the external-sort merge is the most suitable method used for external sorting.

External Sort-Merge Algorithm

Here, we will discuss the external-sort merge algorithm stages in detail:

In the algorithm, M signifies the number of disk blocks available in the main memory buffer for sorting.

Stage 1: Initially, we create a number of sorted runs. Sort each of them. These runs contain only a few records of the relation.

```
i = 0;

repeat
    read either M blocks or the rest of the relation having a smaller size;
    sort the in-memory part of the relation;
    write the sorted data to run file Ri;
    i =i+1;

Until the end of the relation
```

In Stage 1, we can see that we are performing the sorting operation on the disk blocks. After completing the steps of Stage 1, proceed to Stage 2.

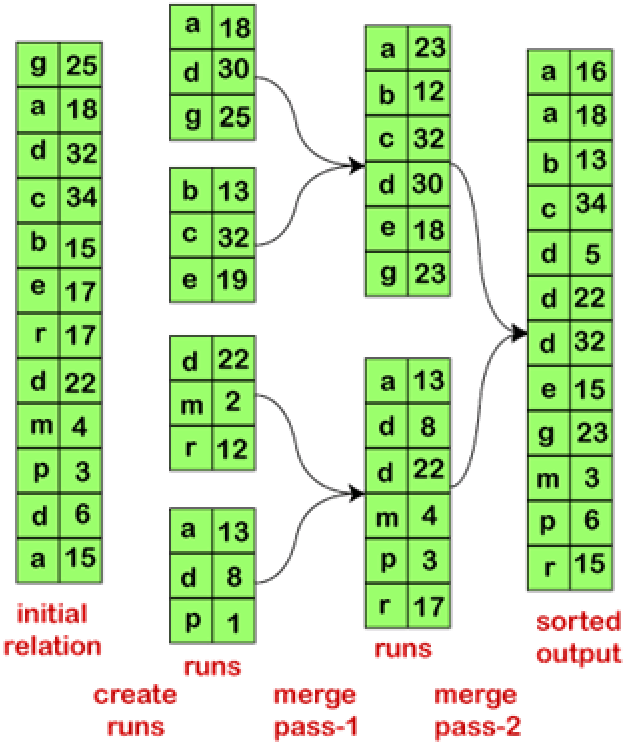
Stage 2: In Stage 2, we merge the runs. Consider that total number of runs, i.e., N is less than M. So, we can allocate one block to each run and still have some space left to hold one block of output. We perform the operation as follows:

```
read one block of each of N files Ri into a buffer block in memory;
repeat
    select the first tuple among all buffer blocks (where selection is made in sorted order);
    write the tuple to the output, and then delete it from the buffer block;
            if the buffer block of any run Ri is empty and not EOF(Ri)
    then read the next block of Ri into the buffer block;
Until all input buffer blocks are empty
```

After completing Stage 2, we will get a sorted relation as an output. The output file is then buffered for minimizing the disk-write operations. As this algorithm merges N runs, that's why it is known as an **N-way merge.**

However, if the size of the relation is larger than the memory size, then either M or more runs will be generated in Stage 1. Also, it is not possible to allocate a single block for each run while processing Stage 2. In such a case, the merge operation process in multiple passes. As M-1 input buffer blocks have sufficient memory, each merge can easily hold M-1 runs as its input. So, the initial phase works in the following way:

- It merges the first M-1 runs for getting a single run for the next one.
- Similarly, it merges the next M-1 runs. This step continues until it processes all the initial runs. Here, the number of runs has a reduced M-1 value. Still, if this reduced value is greater than or equal to M, we need to create another pass. For this new pass, the input will be the runs created by the first pass.
- The work of each pass will be to reduce the number of runs by M-1 value. This job repeats as many times as needed until the number of runs is either less than or equal to M.
- Thus, a final pass produces the sorted output.



External sorting using sort-merge

Estimating cost for External Sort-Merge Method

The cost analysis of external sort-merge is performed using the above-discussed stages in the algorithm:

- Assume b_r denote number of blocks containing records of relation r.
- In the first stage, it reads each block of the relation and writes them back. It takes a total of $2b_r$ block transfers.
- Initially, the value of the number of runs is $[b_r/M]$. As the number of runs decreases by M-1 in each merge pass, so it needs a total number of $[\log_{M-1}(b_r/M)]$ merge passes.

Every pass read and write each block of the relation only once. But with two exceptions:

- The final pass can give a sorted output without writing its result to the disk
- There might be chances that some runs may not be read or written during the pass.

Neglecting such small exceptions, the total number of block transfers for external sorting comes out:

$$\mathbf{b_r\,(2\ \Gamma \log_{M-1} (b_r/M) \rceil + 1)}$$

We need to add the disk seek cost because each run needs seeks reading and writing data for them. If in Stage 2, i.e., the merge phase, each run is allocated with b_b buffer blocks or each run reads b_b data at a time, then each merge needs $[b_r /b_b]$ seeks for reading the data. The output is written sequentially, so if it is on the same disk as input runs, the head will need to move between the writes of consecutive blocks. Therefore, add a total of $2[b_r /b_b]$ seeks for each merge pass and the total number of seeks comes out:

$$\mathbf{2 \lceil b_r /M \rceil + \lceil b_r /b_b \rceil (2\lceil \log_{M-1}(b_r /M)\rceil - 1)}$$

Thus, we need to calculate the total number of disk seeks for analyzing the cost of the External merge-sort algorithm.

Example of External Merge-sort Algorithm

Let's understand the working of the external merge-sort algorithm and also analyze the cost of the external sorting with the help of an example.

<u>JOIN OPERATION:</u>

The join operation is the most time-consuming operation in query processing. The Database Engine supports the following three different join processing techniques, so the optimizer can choose one of them depending on the statistics for both tables:

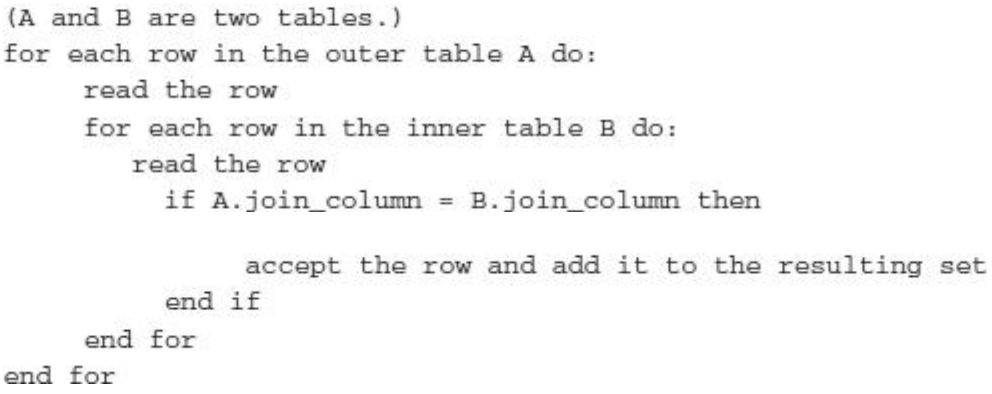
- Nested loop
- Merge join
- Hash join

The following subsections describe these techniques.

<u>Nested loop:</u>

Nested loop is the processing technique that works by “brute force.” In other words, for each row of the outer table, each row from the inner table is retrieved and compared. The pseudo-code in Algorithm 1 demonstrates the nested loop processing technique for two tables.

ALGORITHM



```
(A and B are two tables.)
for each row in the outer table A do:
     read the row
     for each row in the inner table B do:
        read the row
          if A.join_column = B.join_column then

               accept the row and add it to the resulting set
          end if
     end for
end for
```

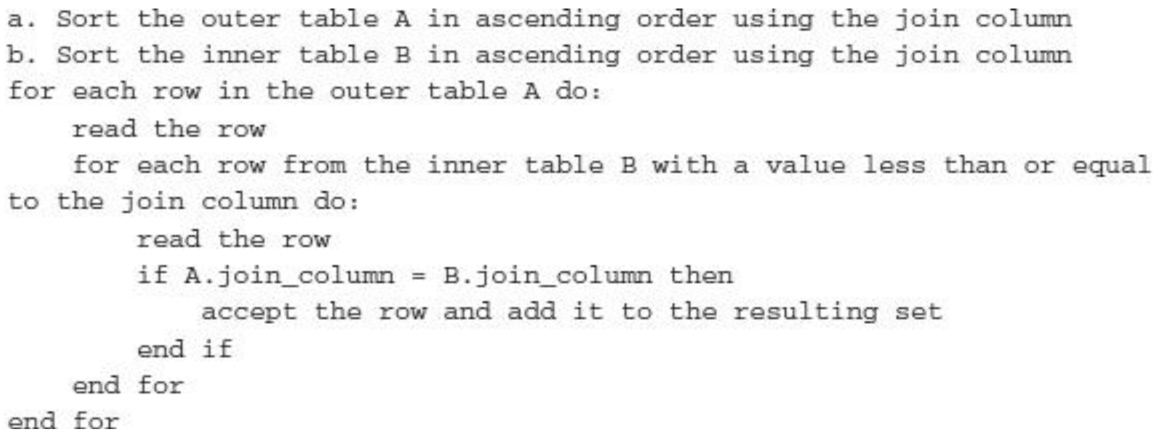
In Algorithm 1, every row selected from the outer table (table A) causes the access of all rows of the inner table (table B). After that, the comparison of the values in the join columns is performed and the row is added to the result set if the values in both columns are equal.

The nested loop technique is very slow if there is no index for the join column of the inner table. Without such an index, the Database Engine would have to scan the outer table once and the inner table n times, where n is the number of rows of the outer table. Therefore, the query optimizer usually chooses this method if the join column of the inner table is indexed, so the inner table does not have to be scanned for each row in the outer table.

<u>Merge join</u>

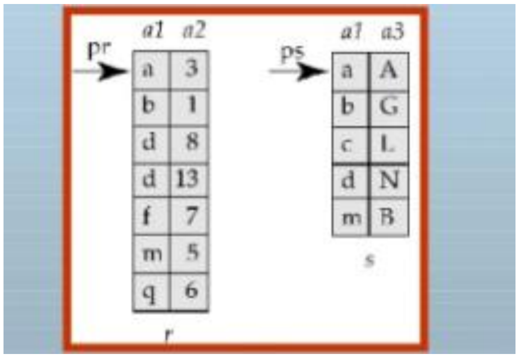
The merge join technique provides a cost-effective alternative to constructing an index for nested loop. The rows of the joined tables must be physically sorted using the values of the join column. Both tables are then scanned in order of the join columns, matching the rows with the same value for the join columns. The pseudo-code in Algorithm 2 demonstrates the merge join processing technique for two tables.

ALGORITHM 2



```
a. Sort the outer table A in ascending order using the join column
b. Sort the inner table B in ascending order using the join column
for each row in the outer table A do:
    read the row
    for each row from the inner table B with a value less than or equal
to the join column do:
        read the row
        if A.join_column = B.join_column then
            accept the row and add it to the resulting set
        end if
    end for
end for
```

<u>EXAMPLE:</u>



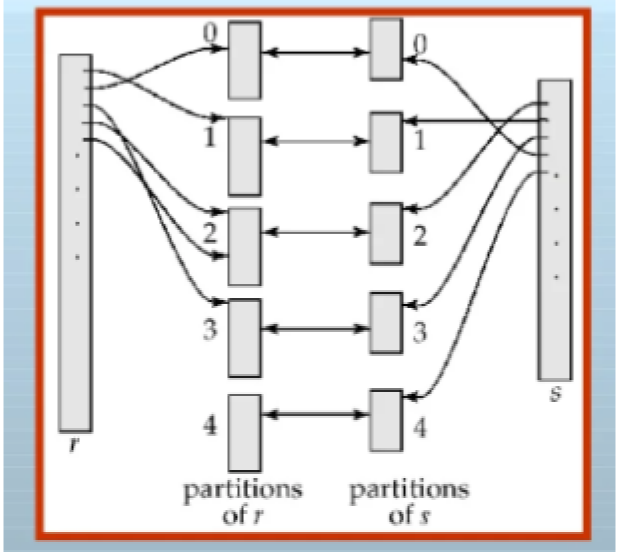
a1	a2
a	3
b	1
d	8
d	13
f	7
m	5
q	6

a1	a3
a	A
b	G
c	L
d	N
m	B

The merge join processing technique has a high overhead if the rows from both tables are unsorted. However, this method is preferable when the values of both join columns are sorted in advance. (This is always the case when both join columns are primary keys of corresponding tables, because the Database Engine creates by default the clustered index for the primary key of a table.)

<u>Hash join:</u>

The hash join technique is usually used when there are no indices for join columns. In the case of the hash join technique, both tables that have to be joined are considered as two inputs: the build input and the probe input. (The smaller table usually represents the build input.) The process works as follows:



1. The value of the join column of a row from the build input is stored in a hash bucket depending on the number returned by the hashing algorithm.
2. Once all rows from the build input are processed, the processing of the rows from the probe input starts.
3. Each value of the join column of a row from the probe input is processed using the same hashing algorithm.
4. The corresponding rows in each bucket are retrieved and used to build the result set.

EVALUATION OF EXPRESSIONS:

In our previous sections, we understood various concepts in query processing. We learned about the query processing steps, selection operations, and also several types of algorithms used for performing the join operation with their cost estimations.

We are already aware of computing and representing the individual relational operations for the given user query or expression. Here, we will get to know how to compute and evaluate an expression with multiple operations.

For evaluating an expression that carries multiple operations in it, we can perform the computation of each operation one by one. However, in the query processing system, we use two methods for evaluating an expression carrying multiple operations. These methods are:

1. Materialization
2. Pipelining

Materialization

In this method, the given expression evaluates one relational operation at a time. Also, each operation is evaluated in an appropriate sequence or order. After evaluating all the operations, the outputs are materialized in a temporary relation for their subsequent uses. It leads the materialization method to a disadvantage. The disadvantage is that it needs to construct those temporary relations for materializing the results of the evaluated operations, respectively. These temporary relations are written on the disks unless they are small in size.

Pipelining

Pipelining is an alternate method or approach to the materialization method. In pipelining, it enables us to evaluate each relational operation of the expression simultaneously in a pipeline. In this approach, after evaluating one operation, its output is passed on to the next operation, and the chain continues till all the relational operations are evaluated thoroughly. Thus, there is no requirement of storing a temporary relation in pipelining. Such an advantage of pipelining makes it a better approach as compared to the approach used in the materialization method. Even the costs of both approaches can have subsequent differences in-between. But, both approaches perform the best role in different cases. Thus, both ways are feasible at their place.

Query optimization:

Query optimization is of great importance for the performance of a relational database, especially for the execution of complex SQL statements. A query optimizer decides the best methods for implementing each query.

The query optimizer selects, for instance, whether or not to use indexes for a given query, and which join methods to use when joining multiple tables. These decisions have a tremendous effect on SQL performance, and query optimization is a key technology for every application, from operational Systems to data warehouse and analytical systems to content management systems.

There is the various principle of Query Optimization are as follows −

- **Understand how your database is executing your query** − The first phase of query optimization is understanding what the database is performing. Different databases have different commands for this. For example, in MySQL, one can use the “EXPLAIN [SQL

Query]" keyword to see the query plan. In Oracle, one can use the "EXPLAIN PLAN FOR [SQL Query]" to see the query plan.

- **Retrieve as little data as possible** − The more information restored from the query, the more resources the database is required to expand to process and save these records. For example, if it can only require to fetch one column from a table, do not use 'SELECT *'.
- **Store intermediate results** − Sometimes logic for a query can be quite complex. It is possible to produce the desired outcomes through the use of subqueries, inline views, and UNION-type statements. For those methods, the transitional results are not saved in the database but are directly used within the query. This can lead to achievement issues, particularly when the transitional results have a huge number of rows.

There are various query optimization strategies are as follows −

- **Use Index** − It can be using an index is the first strategy one should use to speed up a query.
- **Aggregate Table** − It can be used to pre-populating tables at higher levels so less amount of information is required to be parsed.
- **Vertical Partitioning** − It can be used to partition the table by columns. This method reduces the amount of information a SQL query required to process.
- **Horizontal Partitioning** − It can be used to partition the table by data value, most often time. This method reduces the amount of information a SQL query required to process.
- **De-normalization** − The process of de-normalization combines multiple tables into a single table. This speeds up query implementation because fewer table joins are required.
- **Server Tuning** − Each server has its parameters and provides tuning server parameters so that it can completely take benefit of the hardware resources that can significantly speed up query implementation.

Transforming Relational Expressions

The first step of the optimizer says to implement such expressions that are logically equivalent to the given expression. For implementing such a step, we use the equivalence rule that describes the method to transform the generated expression into a logically equivalent expression.

Although there are different ways through which we can express a query, with different costs. But for expressing a query in an efficient manner, we will learn to create alternative as well as equivalent expressions of the given expression, instead of working with the given expression.

Two relational-algebra expressions are equivalent if both the expressions produce the same set of tuples on each legal database instance. A **legal database instance** refers to that database system which satisfies all the integrity constraints specified in the database schema. However, the sequence of the generated tuples may vary in both expressions, but they are considered equivalent until they produce the same tuples set.

Equivalence Rules

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.

Let's discuss a number of equivalence rules:

Rule 1: Cascade of σ

This rule states the deconstruction of the conjunctive selection operations into a sequence of individual selections. Such a transformation is known as a **cascade of σ**.

$\sigma_{\theta 1 \wedge \theta 2} (E) = \sigma_{\theta 1} (\sigma_{\theta 2} (E))$

Rule 2: Commutative Rule

a) This rule states that selections operations are commutative.

$\sigma_{\theta 1} (\sigma_{\theta 2} (E)) = \sigma_{\theta 2} (\sigma_{\theta 1} (E))$

b) Theta Join (θ) is commutative.

$E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$ ***(θ is in subscript with the join symbol)***

However, in the case of theta join, the equivalence rule does not work if the order of attributes is considered. Natural join is a special case of Theta join, and natural join is also commutative.

However, in the case of theta join, the equivalence rule does not work if the order of attributes is considered. Natural join is a special case of Theta join, and natural join is also commutative.

Rule 3: Cascade of ∏

This rule states that we only need the final operations in the sequence of the projection operations, and other operations are omitted. Such a transformation is referred to as a **cascade of** ∏.

$\prod L1 (\prod L2 (. . . (\prod Ln (E)) . . .)) = \prod L1 (E)$

Rule 4: We can combine the selections with Cartesian products as well as theta joins

Rule 4: We can combine the selections with Cartesian products as well as theta joins

1. $\boldsymbol{\sigma_{\theta} (E_1 \text{ x } E_2) = E_{1\theta} \bowtie E_2}$
2. $\boldsymbol{\sigma_{\theta 1} (E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2}$

Rule 5: Associative Rule

a) This rule states that natural join operations are associative.

$\boldsymbol{(E1 \bowtie E2) \bowtie E3 = E1 \bowtie (E2 \bowtie E3)}$

b) Theta joins are associative for the following expression:

$\boldsymbol{(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)}$

In the theta associativity, θ2 involves the attributes from E2 and E3 only. There may be chances of empty conditions, and thereby it concludes that Cartesian Product is also associative.

Rule 6: Distribution of the Selection operation over the Theta join.

Under two following conditions, the selection operation gets distributed over the theta-join operation:

a) When all attributes in the selection condition θ_0 include only attributes of one of the expressions which are being joined.

$\boldsymbol{\sigma_{\theta 0} (E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta 0} (E_1)) \bowtie_{\theta} E_2}$

b) When the selection condition θ_1 involves the attributes of E_1 only, and θ_2 includes the attributes of E_2 only.

$\boldsymbol{\sigma_{\theta 1 \wedge \theta 2} (E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta 1} (E_1)) \bowtie_{\theta} ((\sigma_{\theta 2} (E_2))}$

Rule 7: Distribution of the projection operation over the theta join.

Under two following conditions, the selection operation gets distributed over the theta-join operation:

a) Assume that the join condition θ includes only in L_1 υ L_2 attributes of E_1 and E_2 Then, we get the following expression:

$$\prod_{L1\upsilon L2} (E_1 \bowtie_\theta E_2) = (\prod_{L1} (E_1)) \bowtie_\theta (\prod_{L2} (E_2))$$

b) Assume a join as $E_1 \bowtie E_2$. Both expressions E_1 and E_2 have sets of attributes as L_1 and L_2. Assume two attributes L_3 and L_4 where L_3 be attributes of the expression E_1, involved in the θ join condition but not in L_1 υ L_2 Similarly, an L_4 be attributes of the expression E_2 involved only in the θ join condition and not in L_1 υ L_2 attributes. Thus, we get the following expression:

$$\prod_{L1\upsilon L2} (E_1 \bowtie_\theta E_2) = \prod_{L1\upsilon L2} ((\prod_{L1\upsilon L3} (E_1)) \bowtie_\theta ((\prod_{L2\upsilon L4} (E_2)))$$

Rule 8: The union and intersection set operations are commutative.

$$E_1 \upsilon E_2 = E_2 \upsilon E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

However, set difference operations are not commutative.

Rule 9: The union and intersection set operations are associative.

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

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

Rule 10: Distribution of selection operation on the intersection, union, and set difference operations.

The below expression shows the distribution performed over the set difference operation.

$$\sigma_p (E_1 - E_2) = \sigma_p(E_1) - \sigma_p(E_2)$$

We can similarly distribute the selection operation on υ and ∩ by replacing with -. Further, we get:

$$\sigma_p (E_1 - E_2) = \sigma_p(E_1) - E_2$$

Rule 11: Distribution of the projection operation over the union operation.

This rule states that we can distribute the projection operation on the union operation for the given expressions.

$$\prod_L (E_1 \upsilon E_2) = (\prod_L (E_1)) \upsilon (\prod_L (E_2))$$

Estimating Statistics of Expression results in DBMS

In order to determine ideal plan for evaluating the query, it checks various details about the tables that are stored in the data dictionary. These information's about tables are collected when a table is created and when various DDL / DML operations are performed on it. The optimizer checks data dictionary for:

- Total number of records in a table, nr. This will help to determine which table needs to be accessed first. Usually smaller tables are executed first to reduce the size of the intermediary tables. Hence it is one of the important factors to be checked.
- Total number of records in each block, fr. This will be useful in determining blocking factor and is required to determine if the table fits in the memory or not.
- Total number of blocks assigned to a table, br. This is also an important factor to calculate number of records that can be assigned to each block. Suppose we have 100 records in a table and total number of blocks are 20, then fr can be calculated as nr/b r = 100/20 = 5.
- Total length of the records in the table, l r. This is an important factor when the size of the records varies significantly between any two tables in the query. If the record length is fixed, there is no significant affect. But when a variable length records are involved in the query, average length or actual length needs to be used depending upon the type of operations.
- Number of unique values for a column, d Ar. This is useful when a query uses aggregation operation or projection. It will provide an estimate on distinct number of columns selected while projection. Number groups of records can be determined using this when Aggregation operation is used in the query. E.g.; SUM, MAX, MIN, COUNT etc.
- Levels of index, x. This data provides the information like whether the single level of index like primary key index, secondary key indexes are used or multi-level indexes like B+ tree index, merge-sort index etc are used. These index levels will provide details about number of block access required to retrieve the data.
- Selection cardinality of a column, s A. This is the number of records present with same column value as A. This is calculated as nr/d Ar. i.e.; total number of records with distinct value of A. For example, suppose EMP table has 500 records and DEPT_ID has 5 distinct values. Then the selection cardinality of DEPT_ID in EMP table is 500/ 5 = 100. That means, on an average 100 employees are distributed among each department. This is helpful in determining average number of records that would satisfy selection criteria.

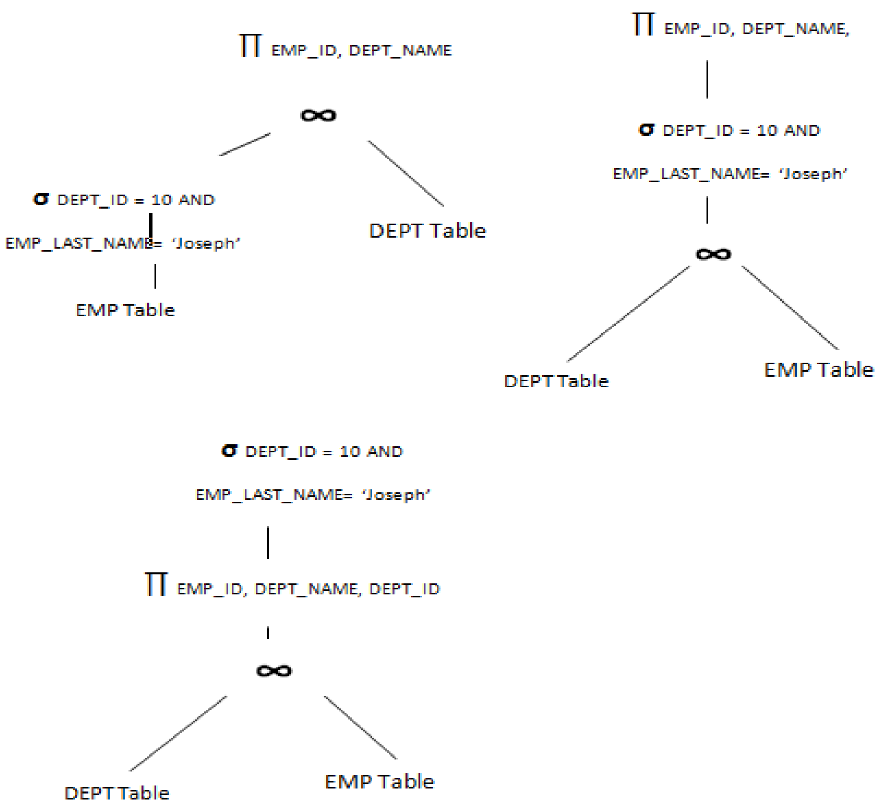
- There many other factors too like index type, data file type, sorting order, type of sorting etc.

Choice of Evaluation Plans in DBMS

- So far we saw how a query is parsed and traversed, how they are evaluated using different methods and what the different costs are when different methods are used. Now the important phase while evaluating a query is deciding which evaluation plan has to be selected so that it can be traversed efficiently. It collects all the statistics, costs, access/ evaluation paths, relational trees etc. It then analyses them and chooses the best evaluation path.
- Like we saw in the beginning of this article, same query is written in different forms of relational algebra. Corresponding trees for them too is drawn by DBMS. Statistics for them based on cost based evaluation and heuristic methods are collected. It checks the costs based on the different techniques that we have seen so far. It checks for the operator, joining type, indexes, number of records, selectivity of records, distinct values etc from the data dictionary. Once all these informations are collected, it picks the best evaluation plan.
- **Have look at below relational algebra and tree for EMP and DEPT.**
  ∏ EMP_ID, DEPT_NAME **(**σ DEPT_ID = 10 AND EMP_LAST_NAME = 'Joseph' **(**EMP**)** ∞DEPT**)**
-

  Or
- ∏ EMP_ID, DEPT_NAME **(**σ DEPT_ID = 10 AND EMP_LAST_NAME = 'Joseph' **(**EMP ∞DEPT**))**
-

  Or
- σ DEPT_ID = 10 AND EMP_LAST_NAME = 'Joseph' **(**∏ EMP_ID, DEPT_NAME, DEPT_ID **(**EMP ∞DEPT**))**
-



What can be observed here? First tree reduces the number of records for joining and seems to be efficient. But what happens if we have index on DEPT_ID? Then the join between EMP and EMP can also be more efficient. But we see the filter condition on EMP table, we have DEPT_ID = 10, which is index column. Hence first applying selection condition and then join will reduce the number of records as well as make the join more efficient than without index. Next are the projected columns – EMP_ID and DEPT_NAME. they are all distinct values. There cannot be duplicate values for them. But we are selecting those values for DEPT_ID = 10, hence DEPT_NAME has only one value. Hence their selectivity is same as number of employees working for DEPT_ID = 10. But we are selecting only those employees whose last name is 'Joseph'. Hence the selectivity is min (distinct (employee (DEPT_10)), distinct (employee (DEPT_10, JOSEPH)). Obliviously distinct (employee (DEPT_10, JOSEPH)) would have lesser value. The optimizer decides all these factors for above 3 trees and then decides first tree would be more efficient. Hence it evaluates the query using first tree.

This is how when any query is submitted to DB is traversed and evaluated.

MATERIALIZED VIEWS:

A materialized view is a view whose contents are computed and stored. Materialized view is also a logical virtual table, but in this case the result of the query is stored in the table or the disk. The performance of the materialized view is better than normal view since the data is stored in the disk.

It's also called indexed views since the table created after the query is indexed and can be accessed faster and efficiently.

Example

Consider the view given below −

Create view branchloan(branch-name, total-loan) as select branch-name , sum(amount) from loan groupby branch-name;

Materializing the above view would be especially useful if the total loan amount is required frequently.

It saves the effort of finding multiple tuples and adding up their amounts.

The task of keeping a materialized view up-to-date with the underlying data is known as materialised view maintenance. It can be maintained by recompilation on every update.

A better option is to use incremental view maintenance. It changes to database relations are used to compute changes to materialized view, which is then updated.

View maintenance can be done by following −

- Manually defining triggers on insert, delete, and update of each relation in the view definition.
- Manually written code to update the view whenever database relations are updated.
- Supported directly by the database.

Incremental view maintenance

The changes like insert and delete operations to a relation or expressions are referred to as its differential, the set of tuples inserted to and deleted from r are denoted ir and dr.

To simplify our description, let’s consider inserts and deletes, we replace updates to a tuple by deletion of the tuple followed by insertion of the updated tuple.

We describe how to compute the change to the result of each relational operation, given changes to its inputs. We then outline how to handle relational algebra expressions.

Materialized view selection

The materialized view selection decision must be made on the basis of the system workload. Indices are just like materialized views, the problem of index selection is closely related to that of materialized view selection, although it is simpler.

Some database systems provide tools to help the database administrator with index and materialized view selection.