UNIT-4

1.Query Processing:

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:

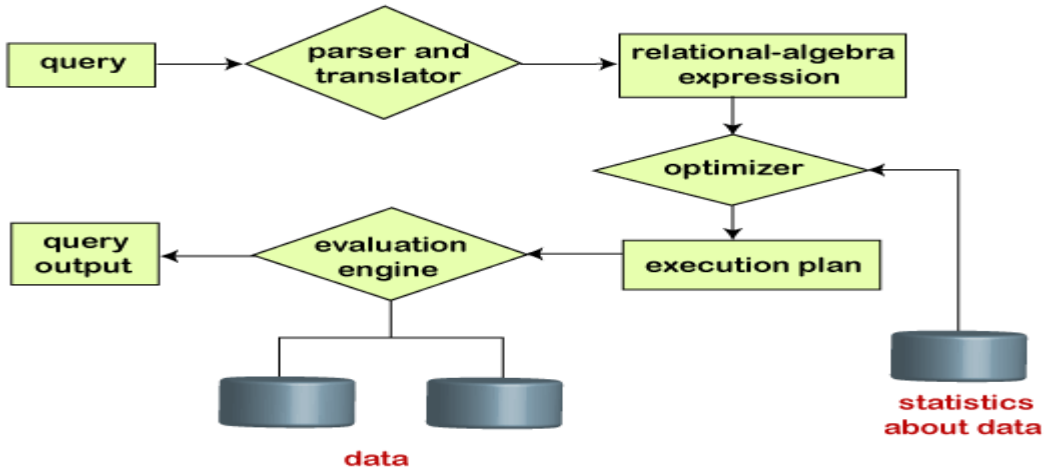
1. Parsing and translation
2. Optimization
3. Evaluation

The query processing works in the following way:

Parsing and Translation

As query processing includes certain activities for data retrieval. Initially, the given user queries get translated in high-level database languages such as SQL. It gets translated into expressions that can be further used at the physical level of the file system. After this, the actual evaluation of the queries and a variety of query -optimizing transformations and takes place. Thus before processing a query, a computer system needs to translate the query into a human-readable and understandable language. Consequently, SQL or Structured Query Language is the best suitable choice for humans. But, it is not perfectly suitable for the internal representation of the query to the system. Relational algebra is well suited for the internal representation of a query. The translation process in query processing is similar to the parser of a query. When a user executes any query, for generating the internal form of the query, the parser in the system checks the syntax of the query, verifies the name of the relation in the database, the tuple, and finally the required attribute value. The parser creates a tree of the query, known as 'parse-tree.' Further, translate it into the form of relational algebra. With this, it evenly replaces all the use of the views when used in the query.

Thus, we can understand the working of a query processing in the below-described diagram:



Steps in query processing

Suppose a user executes a query. As we have learned that there are various methods of extracting the data from the database. In SQL, a user wants to fetch the records of the employees whose salary is greater than or equal to 10000. For doing this, the following query is undertaken:

select emp_name from Employee where salary>10000;

Thus, to make the system understand the user query, it needs to be translated in the form of relational algebra. We can bring this query in the relational algebra form as:

- $\sigma_{salary>10000}$ (π_{salary} **(Employee))**
- π_{salary} ($\sigma_{salary>10000}$ **(Employee))**

After translating the given query, we can execute each relational algebra operation by using different algorithms. So, in this way, a query processing begins its working.

2.Evaluation

For this, with addition to the relational algebra translation, it is required to annotate the translated relational algebra expression with the instructions used for specifying and evaluating each operation. Thus, after translating the user query, the system executes a query evaluation plan.

Query Evaluation Plan

- In order to fully evaluate a query, the system needs to construct a query evaluation plan.
- The annotations in the evaluation plan may refer to the algorithms to be used for the particular index or the specific operations.
- Such relational algebra with annotations is referred to as **Evaluation Primitives**. The evaluation primitives carry the instructions needed for the evaluation of the operation.
- Thus, a query evaluation plan defines a sequence of primitive operations used for evaluating a query. The query evaluation plan is also referred to as **the query execution plan**.
- A **query execution engine** is responsible for generating the output of the given query. It takes the query execution plan, executes it, and finally makes the output for the user query.

Optimization

- The cost of the query evaluation can vary for different types of queries. Although the system is responsible for constructing the evaluation plan, the user does need not to write their query efficiently.
- Usually, a database system generates an efficient query evaluation plan, which minimizes its cost. This type of task performed by the database system and is known as Query

Optimization.

- For optimizing a query, the query optimizer should have an estimated cost analysis of each operation. It is because the overall operation cost depends on the memory allocations to several operations, execution costs, and so on.

Finally, after selecting an evaluation plan, the system evaluates the query and produces the output of the query.

3.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 nonkey:** The difference between equality on key and nonkey 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 nonkey.
- **Secondary index, equality on key or nonkey:** 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 nonkey. 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:

$$\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$$

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:

$$\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$$

A disjunction is the union of all records that satisfies the given selection condition θ_i.

Negation: The result of a selection $\sigma_{\neg\theta}(r)$ 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 $\sigma_{\theta}(r)$.

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.

4.Cost Estimation

Here, the overall cost of the algorithm is composed by adding the cost of individual index scans and cost of fetching the records in the intersection of the retrieved lists of pointers. We can minimize the cost by sorting the list of pointers and fetching the sorted records. So, we found the following two points for cost estimation:

- We can fetch all selected records of the block using a single I/O operation because each pointer in the block appears together.
- The disk-arm movement gets minimized as blocks are read in sorted order.

Cost Estimation Chart for various Selection algorithms

Here, b_r is the number of blocks in the file.

h_i denotes the height of the index

b is the number of blocks holding records with specified search key

n is the number of fetched records

Selection Algorithms	Cost	Why So?
Linear Search	$t_s + b_r * t_T$	It needs one initial seek with b_r block transfers.
Linear Search, Equality on Key	$t_s + (b_r/2) * t_T$	It is the average case where it needs only one record condition. So as soon as it is found, the scan termina
Primary B^+-tree index, Equality on Key	$(h_i +1) * (t_r + t_s)$	Each I/O operation needs one seek and one block tr the record by traversing the height of the tree.
Primary B^+-tree index, Equality on a Nonkey	$h_i * (t_T + t_s) + b * t_T$	It needs one seek for each level of the tree, and on first block.
Secondary B^+-tree index, Equality on Key	$(h_i + 1) * (t_r + t_s)$	Each I/O operation needs one seek and one block tr the record by traversing the height of the tree.
Secondary B^+-tree index, Equality on Nonkey	$(h_i + n) * (t_r + t_s)$	It requires one seek per record because each record different block.
Primary B^+-tree index, Comparison	$h_i * (t_r + t_s) + b * t_T$	It needs one seek for each level of the tree, and on first block.
Secondary B^+-tree index, Comparison	$(h_i + n) * (t_r + t_s)$	It requires one seek per record because each record different block.

5.Estimating Query Cost

In the previous section, we understood about Query processing steps and evaluation plan. 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 $\mathbf{b*t_T + S*t_S}$ 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.

6.Hash Join Algorithm

The Hash Join algorithm is used to perform the natural join or equi join operations. The concept behind the Hash join algorithm is to partition the tuples of each given relation into sets. The partition is done on the basis of the same hash value on the join attributes. The hash function

provides the hash value. The main goal of using the hash function in the algorithm is to reduce the number of comparisons and increase the efficiency to complete the join operation on the relations.

For example, suppose there are two tuples a and b where both of them satisfy the join condition. It means they have the same value for the join attributes. Suppose that both a and b tuples consist of a hash value as i. It implies that tuple a should be in a_i, and tuple b should be in b_i. Thus, we only compare a tuples in a_i with b tuples of b_i. We do not need to compare the b tuples in any other partition. Therefore, in this way, the hash join operation works.

Hash Join Algorithm

```
//Partition s//
for each tuple ts in s do begin
i = h(ts [JoinAttrs]);
Hsi = Hsi U {ts};
end
//Partition r//
for each tuple tr in r do begin
i = h(tr[JoinAttrs]);
Hri = Hri U {tr};
end
//Perform the join operation on each partition//
for i= 0 to nh do begin
        read Hsi and build an in-memory hash index on it;
        for each tuple tr in Hri do begin
          probe the hash index on Hsi to locate all tuples
              such that ts[JoinAttrs] = tr[JoinAttrs];
            for each matching tuple ts in Hsi do begin
                    add tr ⋈ ts to the result;
            end
        end
end
```

It is the Hash join algorithm in which we have computed the natural join of two given relations r and s. In the algorithm, there are various terms used:

$t_r \bowtie t_s$: It defines the concatenation of the attributes of tuple t_r and t_s, which is further followed by projecting out the repeated attributes

t_r and t_s: These are the tuples of relations r and s, respectively.

Let's understand the hash join algorithm with the following steps:

Step 1: In the algorithm, firstly, we have partitioned both relations r and s.

Step 2: After partitioning, we perform a separate indexed nested-loop join on each of the partition pairs i using for loop as i = 0 to n_h.

Step 3: For performing the nested-loop join, it initially creates a hash index on each s_i and then probes with tuples from r_i. In the algorithm, relation r is the **probe input**, and relation s is the **build input**.

There is a benefit of using the Hash Join algorithm i.e., the hash index on si is built-in memory, so for fetching the tuples, we do not need to access the disk. It is good to use smaller input relations as the build relations.

Recursive Partitioning in Hash Join

Recursive partitioning is the one in which the system repeats the partitioning of the input until each partition of the build input fits into the memory. The recursive partitioning is needed when the value of n_h is greater than or equal to the number of memory blocks. It becomes difficult to split the relation in one pass since there can be insufficient buffer blocks. So, it's better to split the relation in repeated passes. In one pass, we can split the input as several partitions because there are sufficient blocks available to be used as output buffers. Each bucket build by the pass is read separately and further partitioned in the next pass so as to create smaller partitions. Also, the hash functions are different in different passes. So, it is better to use recursive partitioning for handling such cases.

Overflows in Hash Join

The overflow condition in hash-table occurs in any partition i of the build relation s due to the following cases:

Case 1: When the hash index on s_i is greater than the main memory, the overflow condition occurs.

Case 2: When there are multiple tuples in the build relation with the same values for the join attributes.

Case 3: When the hash function does not hold randomness and uniformity characteristics.

Case 4: When some of the partitions have more tuples than the average and others have fewer tuples, then such type of partitioning is known as **skewed**.

Handling the Overflows

We can handle such cases of hash-table overflows using various methods.

- **Using Fudge Factor**

We can handle a small amount of skew by increasing the number of partitions with the use of the **fudge factor**. The fudge factor is a small value that increases the number of partitions. So, it

will help to reduce the expected size of each partition, including their hash index less than the memory size. Unfortunately, the use of a fudge factor makes the user conservative on the size of the partitions. Thus, the chances of overflow are still possible. However, the use of the fudge factor is suitable for handling small overflows, but it is not sufficient for handling large overflows in the hash-table.

As a result, we have two more methods for handling the overflows.

The overflow resolution method is applied during the build phase when a hash index overflow is detected. The overflow resolution works in the following way:

It finds s_i for any partition i if having size larger than the memory size. It again partitions such build relation s_i into smaller partitions through a different hash function. Similarly, it partitions the probe relation r_i through the new hash function, and only those tuples are joined, which are having matching partitions. But, it is a less careful approach because this method waits for such conditions to occur, and then take the necessary actions to resolve the problem.

2. Overflow Avoidance

The overflow avoidance method uses a careful approach while partitioning in order to avoid the occurrence of overflow in the build phase. The overflow avoidance works in the following way:

It initially partitions the build relation s into several small partitions and then combines some of the partitions. These partitions are combined in such a way that each combined partition fits in the memory. Similarly, it partitions the probe relation r as the combined partitions on s. But, the size of r_i does not matter in this method.

Both overflow resolution and overflow avoidance methods may fail on some partitions if a large number of tuples in s have the same value for the join attributes. In such a case, it is better to use block nested-loop join rather than applying the hash join technique for completing the join operation on those partitions.

7.Cost Analysis of Hash Join

For analyzing the cost of a hash join, we consider that no overflow occurs in the hash join. We will consider only two cases where:

1. Recursive partitioning is not needed

We need to read and write relations r and s completely for partitioning them. For this, a total of $\mathbf{2(b_r + b_s)}$ block transfers are required. The term $\mathbf{b_r}$ **a**nd $\mathbf{b_s}$ are the number of blocks holding records of relations r and s. Both relations read each partition once for more $b_r + b_s$ blocks transfers. However, the partitions might have occupied slightly more number of blocks than $b_r + b_s$, which results in partially filled blocks. To access such partially filled blocks can include the overhead of $2n_h$ approximately for each relation. Thus, a hash join cost estimates need:

Number of block transfers = $3(b_r + b_s) + 4n_h$

Here, we can neglect the overhead value of $4n_h$ since it is much smaller than $b_r + b_s$ value.

Number of disk seeks = $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) + 2n_h$

Here, we have assumed that each input buffers are allocated with b_b blocks, and the build, as well as probe phase, needs only one seek for each n_h partition of the relation, as we can read each partition sequentially.

2. Recursive partition is required

In this case, each pass reduces the size of each partition by M-1 expected factor, and also passes are repeated until it makes the size of each partition as M blocks at most. Therefore, for partitioning the relation s, we need:

Number of passes = $\lceil \log_{M-1}(b_s) - 1 \rceil$

The number of passes required in the partitioning of the build and probe relations is the same. As in each pass, each block of s is read and written out and needs a total of $2b_s\lceil \log_{M-1}(b_s) - 1 \rceil$ block transfers for splitting relation s. Thus, a hash join cost estimates need:

Number of block transfers = $2(b_r + b_s)\lceil \log_{M-1}(b_s) - 1 \rceil + b_r + b_s$

Number of disk seeks = $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)\lceil \log_{M-1}(b_s) - 1 \rceil$

Here, we assume that for buffering each partition we allocate b_b blocks to them. Also, we have neglected a relatively small number of seeks during the build and probe phase.

As a result, the hash join algorithm can be further improved if the size of the main memory increases or is large.

Hybrid Hash Join

It is a type of hash join that is useful for performing the join operations in which the memory size is relatively large. But still, the build relation does not fit in the memory completely. So, the hybrid hash join algorithm resolves the drawback of the hash join algorithm.

Materialization in Query Processing

In the previous, we took a brief introduction about materialization and how to evaluate multiple operations of an expression.

Materialization is an easy approach for evaluating multiple operations of the given query and storing the results in the temporary relations. The result can be the output of any join condition, selection condition, and many more. Thus, materialization is the process of creating and setting a view of the results of the evaluated operations for the user query. It is similar to the cache memory where the searched data get settled temporarily. We can easily understand the working of materialization through the pictorial representation of the expression. An operator tree is used for representing an expression.

The materialization uses the following approach for evaluating operations of the given expression:

- In the operator tree, we begin from the lowest-level operations (at the bottom of the tree) in the expression. The inputs to the lowest level operations are stored in the form of relations in the database. **For example,** suppose we want to fetch the name of the student as 'John' from the 'Student' relation.
  The relation expression will be:
  $\sigma_{\text{name= "John"}}$ (Student)
  In this example, there is only one operation of selecting the name from the given relation. Also, this operation is the lowest-level operation. So, we will begin by evaluating this selection operation.
- Now, we will use an appropriate algorithm which is suitable for evaluating the operation. Like in our example, we will use an appropriate selection algorithm for retrieving the name from the Student relation.
- Then, store the result of the operation in the temporary relations.
- We use these temporary relations for evaluating the next-level operation in the operator tree. The result works as an input for every next level up in the tree.
- Repeat these steps until all operators at the root of tree will be evaluated, and the final result of the expression will be generated.

8.Cost Estimation of Materialized Evaluation

The process of estimating the cost of the materialized evaluation is different from the process of estimating the cost of an algorithm. It is because in analyzing the cost of an algorithm, we do not include the cost of writing the results on to the disks. But in the evaluation of an expression, we not only compute the cost of all operations but also include the cost of writing the result of currently evaluated operation to disk.

To estimate the cost of the materialized evaluation, we consider that results are stored in the buffer, and when the buffer fills completely, the results are stored to the disk.

Let, a total of b_r number of blocks are written. Thus, we can estimate b_r as:

$b_r = n_r/f_r$.

Here, n_r is the estimated number of tuples in the result relation r and fr is the number of records of relation r that fits in a block. Thus, fr is a blocking factor of the resultant relation r.

With this, we also need to calculate the transfer time by estimating the number of required disks. It is so because the disk head may have moved in-between the successive writes of the block. Thus, we can estimate:

Number of seeks = $\lceil b_r / b_b \rceil$

Here, b_b defines the size of the output buffer, i.e., measured in blocks.

Merge Join Algorithm

The merge joins are used for performing natural joins and equi-joins for given relations r and s. We use an algorithm for performing the merge join, known as **the Merge Join algorithm**. It is also known as a **sort-merge-join algorithm.**

Merge Join Algorithm

The merge join algorithm is given below:

JoinAttrs: It denotes the attributes in the intersection of **r ∩ s**.

r ∩ s: The r ∩ s refers to those attributes which are common in relations r and s.

$t_s \bowtie t_r$: A concatenated expression of the attributes of t_s and t_r tuples. It is further followed by projecting out repeated attributes.

t_s and t_r: These are two tuples having the same value of JoinAttrs.

S_s: It reads those join attributes of a group of tuples of a relation which are having the same values.

In the merge join algorithm, it associates each relation with a pointer. Initially, the pointer points to the first tuple of the relation and then moves towards the next one as soon the algorithm proceeds. Also, the algorithm needs that each set of tuples S_s fits in the memory even if the size of the relation s is large. However, if for some attribute values, S_s seem larger than the available memory size, we can perform block nested-loop join for it. Somehow if the given input relations r and s are not sorted on the join attributes or anyone is unsorted, we need to sort them before applying the merge join algorithm.

Cost Analysis of Merge Join Algorithm

If the relations are sorted and tuples having the same value on the join attributes are placed consecutively. Then we need to read each tuple only once, and thus the block will also be read for once. Thus,

Number of block transfers = $b_r + b_s$

Also, in both files, the number of block transfers is equal.

Hybrid Merge Join Algorithm

The Hybrid merge join is different from the merge join. In merge join operation, we saw that it is a must to sort the given relations before applying the merge join technique. However, both join attributes consist of secondary indices, then also we can perform a variation of the merge join operations on unsorted tuples too. For doing so, the applied merge join algorithm will scan the records through the indices, which will enable to retrieve the records in a sorted manner. Thus, such variation of the merge join operations leads to a significant drawback, i.e.:

- It is possible that the records might be placed in different file blocks. It means they might be scattered In several blocks of files. So, for accessing each tuple, we also need to access the particular file block, and it is a costly step.

For preventing ourselves from such expensive access, we use a new technique which is known as 'Hybrid Merge Join' technique. The hybrid merge join operation combines the indices with merge join.

Nested-Loop Join Algorithm

In our previous section, we learned about joins and various types of joins. In this section, we will know about the nested loop join algorithm.

A nested loop join is a join that contains a pair of nested for loops. To perform the nested loop join i.e., θ on two relations **r** and **s**, we use an algorithm known as the **Nested loop join algorithm**. The computation takes place as:

$$\mathbf{r} \bowtie_{\theta} \mathbf{s}$$

where r is known as the **outer relation** and s is the **inner relation** of the join. It is because the for loop of r encloses the for loop of s.

Block Nested-Loop Join

Block Nested-Loop Join is a variant of nested-loop join in which each block of the inner relation is paired with each block of the outer relation. The block nested-loop join saves major block access in a situation where the buffer size is small enough to hold the entire relation into the memory. It does so by processing the relations on the basis of per block rather on the basis of per tuple. Within each pair of blocks, the block nested-loop join pairs each tuple of one block with each tuple in the other block to produce all pairs of tuples. It pairs only those tuples that satisfy the given join condition and them to the result.

Block Nested-Loop Join Algorithm

The algorithm that is used to perform the block nested-loop join is known as the **Block Nested-Loop Join algorithm**. We will use the same relations r and s in this algorithm.

```
for each block b_r of r do begin
  for each block b_s of s do begin
    for each tuple t_r in b_r do begin
      for each tuple t_s in b_s do begin
        test pair (t_r, t_s) to determine if they pass the given join condition
         if test passed
        add t_r . t_s to the result;
  end
end
```

```
        end
    end
```

Cost Analysis of Block Nested-Loop Join Algorithm

There is a major difference between the cost of block nested-loop join and nested loop-join algorithm. In the worst case of block nested-loop join, each block in the inner relation s is read only for one time for each block in the outer relation r. On the other hand, the nested-loop join reads each tuple in the inner relation s for one time for each tuple in the outer relation r. Thus in block nested-loop join,

Total number of block transfers in worst case = $b_r * b_s + b_r$

Total number of seeks required = $2 * b_r$

Here, b_r and b_s are the number of blocks holding records of the given relation r and s, respectively. Also, each scan of s (inner relation) needs only one seek, and r (outer relation) requires one seek per block. In the best case, the inner relation fits entirely into memory. Thus,

Total number of block transfers in best case = $b_r + b_s$

Total number of seeks required = $2(n_r + b_r)$

In the case where none of the given relations r and s fits entirely into the memory, it is efficient to use the inner relation i.e., s as the outer relation.

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.

2. **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).

3. **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.

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.

2. **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).

3. **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.

4. **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.

5. **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.

6. **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.

7. **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.

8. **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.

9. **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.

Estimating Statistics of Expression Results

The cost of an operation depends on the size and other statistics of its inputs. Given an expression such as *a* (*b c*) to estimate the cost of joining *a* with (*b c*), we need to have estimates of statistics

such as the size of *b c*.

In this section we first list some statistics about database relations that are stored in database system catalogs, and then show how to use the statistics to estimate statistics on the results of various elational operations.

One thing that will become clear later in this section is that the estimates are not very accurate, since they are based on assumptions that may not hold exactly. A query evaluation plan that has the lowest estimated execution cost may therefore not actually have the lowest actual execution cost. However, real-world experience has shown that even if estimates are not precise, the plans with the lowest estimated costs usually have actual execution costs that are either the lowest actual execution costs, or are close to the lowest actual execution costs.

Catalog Information

The DBMS catalog stores the following statistical information about database relations:

• *nr* , the number of tuples in the relation *r*.

• *br* , the number of blocks containing tuples of relation *r*.

• *lr* , the size of a tuple of relation *r* in bytes.

• *fr* , the blocking factor of relation *r* — that is, the number of tuples of relation *r* that fit into one block.

• *V* (*A*, *r*), the number of distinct values that appear in the relation *r* for attribute *A*. This value is the same as the size of Π*A*(*r*). If *A* is a key for relation *r*, *V* (*A*, *r*) is *nr* .

The last statistic, *V* (*A*, *r*), can also be maintained for sets of attributes, if desired, instead of just for individual attributes. Thus, given a set of attributes, *A*, *V* (*A*, *r*) is the size of Π*A*(*r*).

If we assume that the tuples of relation *r* are stored together physically in a file, the following equation holds:

Statistics about indices, such as the heights of B+-tree indices and number of leaf pages in the indices, are also maintained in the catalog.

If we wish to maintain accurate statistics, then, every time a relation is modified, we must also update the statistics. This update incurs a substantial amount of over- head. Therefore, most systems do not update the statistics on every modification. In- stead, they update the statistics during periods of light system load. As a result, the statistics used for choosing a query-processing strategy may not be completely accurate. However, if not too many updates occur in the intervals between the updates of the statistics, the statistics will be sufficiently accurate to provide a good estimation of the relative costs of the different plans.

The statistical information noted here is simplified. Real-world optimizers often maintain further statistical information to improve the accuracy of their cost estimates of evaluation plans. For instance, some databases store the distribution of values for each attribute as a **histogram**: in a histogram the values for the attribute are divided into a number of ranges, and with each range the histogram associates the number of tuples whose attribute value lies in that range. As an example of a histogram, the range of values for an attribute *age* of a relation *person* could be divided into 0 – 9, 10 – 19, …, 90 – 99 (assuming a maximum age of 99). With each range we store a count of

the number of *person* tuples whose *age* values lie in that range. With- out such histogram information, an optimizer would have to assume that the distribution of values is uniform; that is, each range has the same count.

Selection Size Estimation

The size estimate of the result of a selection operation depends on the selection predicate. We first consider a single equality predicate, then a single comparison predicate, and finally combinations of predicates.

• $\sigma A = a(r)$: If we assume uniform distribution of values (that is, each value appears with equal probability), the selection result can be estimated to have $nr /V (A, r)$ tuples, assuming that the value a appears in attribute A of some record of r. The assumption that the value a in the selection appears in some record is generally true, and cost estimates often make it implicitly. However, it is often not realistic to assume that each value appears with equal probability. The *branch-name* attribute in the *account* relation is an example where the assumption is not valid. There is one tuple in the *account* relation for each account. It is reasonable to expect that the large branches have more ac- counts than smaller branches. Therefore, certain *branch-name* values appear with greater probability than do others. Despite the fact that the uniform distribution assumption is often not correct, it is a reasonable approximation of reality in many cases, and it helps us to keep our presentation relatively simple.

• $\sigma A \leq v (r)$: Consider a selection of the form $\sigma A \leq v (r)$. If the actual value used in the comparison (v) is available at the time of cost estimation, a more ac- curate estimate can be made. The lowest and highest values ($\min(A, r)$ and $\max(A, r)$) for the attribute can be stored in the catalog. Assuming that values are uniformly distributed, we can estimate the number of records that will

satisfy the condition $A \leq v$ as 0 if $v < \min(A, r)$, as nr if $v \geq \max(A, r)$, and

$$n_r \cdot \frac{v - \min(A, r)}{\max(A, r) - \min(A, r)}$$

otherwise.

In some cases, such as when the query is part of a stored procedure, the value v may not be available when the query is optimized. In such cases, we will assume that approximately one-half the records will satisfy the comparison condition. That is, we assume the result has $nr /2$ tuples; the estimate maybe very inaccurate, but is the best we can do without any further information.

• Complex selections:

Conjunction: A *conjunctive selection* is a selection of the form

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n}(r)$$

We can estimate the result size of such a selection: For each θi, we estimate the size of the selection $\sigma\theta i (r)$, denoted by si, as described previously. Thus, the probability that a tuple in the relation satisfies selection condition θi is si/nr .

The preceding probability is called the **selectivity** of the selection $\sigma\theta i (r)$. Assuming that the conditions are *independent* of each other, the probability that a tuple satisfies all the conditions is simply the product of all these probabilities. Thus, we estimate the number of tuples in the full selection as

$$n_r * \frac{s_1 * s_2 * \cdots * s_n}{n_r^n}$$

Disjunction: A *disjunctive selection* is a selection of the form

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

A disjunctive condition is satisfied by the union of all records satisfying the individual, simple conditions θ_i.

As before, let s_i/n_r denote the probability that a tuple satisfies condition θ_i. The probability that the tuple will satisfy the disjunction is then 1 minus the probability that it will satisfy *none* of the conditions:

$$1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \cdots * (1 - \frac{s_n}{n_r})$$

Multiplying this value by *nr* gives us the estimated number of tuples that satisfy the selection.

D **Negation:** In the absence of nulls, the result of a selection $\sigma\neg\theta$ (*r*) is simply the tuples of *r* that are not in $\sigma\theta$ (*r*). We already know how to estimate the number of tuples in $\sigma\theta$ (*r*). The number of tuples in $\sigma\neg\theta$ (*r*) is therefore estimated to be *n*(*r*) minus the estimated number of tuples in $\sigma\theta$ (*r*).

We can account for nulls by estimating the number of tuples for which the condition θ would evaluate to *unknown*, and subtracting that number from the above estimate ignoring nulls. Estimating that number would require extra statistics to be maintained in the catalog.

Join Size Estimation

In this section, we see how to estimate the size of the result of a join.

The Cartesian product $r \times s$ contains $nr * ns$ tuples. Each tuple of $r \times s$ occupies $lr + ls$ bytes, from which we can calculate the size of the Cartesian product.

Estimating the size of a natural join is somewhat more complicated than estimating the size of a selection or of a Cartesian product. Let *r*(*R*) and *s*(*S*) be relations.

• If $R \cap S = \emptyset$ — that is, the relations have no attribute in common — then *r s* is the same as $r \times s$, and we can use our estimation technique for Cartesian products.

• If $R \cap S$ is a key for *R*, then we know that a tuple of *s* will join with at most one tuple from *r*. Therefore, the number of tuples in *rs* I s no greater than the number of tuples in *s*. The case where $R \cap S$ is a key for *S* is symmetric to the case just described. If $R \cap S$ forms a foreign key of *S*, referencing *R*, the number of tuples in *r s* is exactly the same as the number of tuples in *s*.
• The most difficult case is when $R \cap S$ is a key for neither *R* nor *S*. In this case, we assume, as we did for selections, that each value appears with equal probability. Consider a tuple *t* of *r*, and assume $R \cap S = \{A\}$. We estimate that tuple *t* produces

$$\frac{n_s}{V(A,s)}$$

tuples in $r \bowtie s$, since this number is the average number of tuples in s with a given value for the attributes A. Considering all the tuples in r, we estimate that there are

$$\frac{n_r * n_s}{V(A,s)}$$

tuples in $r \bowtie s$. Observe that, if we reverse the roles of r and s in the preceding estimate, we obtain an estimate of

$$\frac{n_r * n_s}{V(A,r)}$$

tuples in *r s*. These two estimates differ if $V(A, r) \neq V(A, s)$. If this situation occurs, there are likely to be dangling tuples that do not participate in the join.

Thus, the lower of the two estimates is probably the more accurate one.

The preceding estimate of join size may be too high if the $V(A, r)$ values for attribute A in r have few values in common with the $V(A, s)$ values for attribute A in s. However, this situation is unlikely to happen in the real world, since dangling tuples either do not exist, or constitute only a small fraction of the tuples, in most real-world relations. More important, the preceding estimate depends on the assumption that each value appears with equal probability. More sophisticated techniques for size estimation have to be used if this assumption does not hold.

We can estimate the size of a theta join $r\ s$ by rewriting the join as $\sigma\theta\ (r \times s)$, and using the size estimates for Cartesian products along with the size estimates for selections, which we saw in Section 14.2.2.

To illustrate all these ways of estimating join sizes, consider the expression

$$depositor \bowtie customer$$

Assume the following catalog information about the two relations:

- $n_{customer} = 10000$.
- $f_{customer} = 25$, which implies that $b_{customer} = 10000/25 = 400$.
- $n_{depositor} = 5000$.
- $f_{depositor} = 50$, which implies that $b_{depositor} = 5000/50 = 100$.
- V(customer-name, depositor) = 2500, which implies that, on average, each customer has two accounts.

Also assume that *customer-name* in *depositor* is a foreign key on *customer*.

In our example of *depositor customer*, *customer-name* in *depositor* is a foreign key referencing *customer*; hence, the size of the result is exactly *ndepositor* , which is 5000.

Let us now compute the size estimates for *depositor customer* without using information about foreign keys. Since V (*customer-name, depositor*) = 2500 and V (*customer- name, customer*) = 10000, the two estimates we get are $5000 * 10000/2500 = 20,000$ and $5000 * 10000/10000 = 5000$, and we choose the lower one. In this case, the lower of these estimates is the same as that which we computed earlier from information about foreign keys.

Size Estimation for Other Operations

We outline below how to estimate the sizes of the results of other relational algebra operations.

Projection: The estimated size (number of records or number of tuples) of a projection of the form $\Pi A(r)$ is $V(A, r)$, since projection eliminates duplicates. **Aggregation:** The size of $AGF(r)$ is simply $V(A, r)$, since there is one tuple in $AGF(r)$ for each distinct value of A.

Set operations: If the two inputs to a set operation are selections on the same relation, we can rewrite the set operation as disjunctions, conjunctions, or negations. For example, $\sigma\theta 1\ (r) \cup \sigma\theta 2\ (r)$ can be rewritten as $\sigma\theta 1 \vee \theta 2\ (r)$. Similarly, we can rewrite intersections as conjunctions, and we can rewrite set difference by using negation, so long as the two relations participating in the set operations are selections on the same relation. We can then use the estimates for selections involving conjunctions, disjunctions, and negation in Section 14.2.2.

If the inputs are not selections on the same relation, we estimate the sizes this way: The estimated size of $r \cup s$ is the sum of the sizes of r and s. The estimated size of $r \cap s$ is the minimum of the sizes of r and s. The estimated size of $r - s$ is the same size as r. All three estimates may be

inaccurate, but provide upper bounds on the sizes.

Outer join: The estimated size of $r ⟕ s$ is the size of $r \bowtie s$ plus the size of r; that of $r ⟖ s$ is symmetric, while that of $r ⟗ s$ is the size of $r \bowtie s$ plus the sizes of r and s. All three estimates may be inaccurate, but provide upper bounds on the sizes.

Estimation of Number of Distinct Values

For selections, the number of distinct values of an attribute (or set of attributes) A in the result of a selection, $V(A, \sigma\theta(r))$, can be estimated in these ways:

- If the selection condition θ forces A to take on a specified value (e.g., $A = 3$), $V(A, \sigma_\theta(r)) = 1$.
- If θ forces A to take on one of a specified set of values (e.g., $(A = 1 \vee A = 3 \vee A = 4)$), then $V(A, \sigma_\theta(r))$ is set to the number of specified values.
- If the selection condition θ is of the form A *op* v, where *op* is a comparison operator, $V(A, \sigma_\theta(r))$ is estimated to be $V(A, r) * s$, where s is the selectivity of the selection.
- In all other cases of selections, we assume that the distribution of A values is independent of the distribution of the values on which selection conditions are specified, and use an approximate estimate of $\min(V(A, r), n_{\sigma_\theta(r)})$. A more accurate estimate can be derived for this case using probability theory, but the above approximation works fairly well.

For joins, the number of distinct values of an attribute (or set of attributes) A in the result of a join, $V(A, r \bowtie s)$, can be estimated in these ways:

- If all attributes in A are from r, $V(A, r \bowtie s)$ is estimated as $\min(V(A, r), n_{r \bowtie s})$, and similarly if all attributes in A are from s, $V(A, r \bowtie s)$ is estimated to be $\min(V(A, s), n_{r \bowtie s})$.
- If A contains attributes $A1$ from r and $A2$ from s, then $V(A, r \bowtie s)$ is estimated as

  $$\min(V(A1, r) * V(A2 - A1, s), V(A1 - A2, r) * V(A2, s), n_{r \bowtie s})$$

  Note that some attributes may be in $A1$ as well as in $A2$, and $A1 - A2$ and $A2 - A1$ denote, respectively, attributes in A that are only from r and attributes in A that are only from s. Again, more accurate estimates can be derived by using probability theory, but the above approximations work fairly well.

The estimates of distinct values are straightforward for projections: They are the same in $\Pi A(r)$ as in r. The same holds for grouping attributes of aggregation. For results of **sum**, **count**, and **average**, we can assume, for simplicity, that all aggregate values are distinct. For **min**(A) and **max**(A), the number of distinct values can be estimated as $\min(V(A, r), V(G, r))$, where G denotes the grouping attributes. We omit details of estimating distinct values for other operations.