<u>Query Processing</u>

Recollect that user submits DML statements to interact with database. This is non procedural i.e. we specify what is required, not how to get it. One important functional unit is query processor that takes the responsibility to make the detailed plan of execution.

User given query can be executed in number of ways to obtain the result. It becomes the responsibility of the system to transform the query given by user into an equivalent and efficient form. The improvement of strategy for processing a query is called query optimization.

User given query undergoes **translation** and intermediate representation is generated.. Thereafter, **optimization** is done on that intermediate form. Relational algebra is useful for intermediate representation.

Translation:

- Translation takes user query as input and generates intermediate representation
- During translation parser breaks the query into tokens (like, relation name, attributes, expression, key words etc.), checks the syntax, verifies relation name, attribute names etc. appearing in the query, replaces view name with the corresponding expression
- Once translation is over, optimization is done

Optimization:

- First phase of optimization is done at relational algebra level. Goal is to obtain equivalent and efficient expression.
- Second phase decides about the detailed strategy like, index to be used or not, order of processing the tuples, join strategies
- At the end of optimization detailed execution plan is obtained

On getting the execution plan, code is generated for execution by the runtime environment of DBMS.

First phase of optimization / Expression level Optimization

a) **Select operation**

Consider the relations DEPT(<u>DCODE</u>, DNAME, TYPE) and EMP(<u>ECOD</u>E, ENAME, ADDRESS, BASIC, DCODE). Suppose we want to find out the details of the employees with BASIC more than 50,000 and working in the departments of emergency type. The expression for this may be:

$\sigma_{\text{BASIC>50000 AND TYPE='EMERGENCY'}}(\text{EMP} * \text{DEPT})$

If executed according to this expression then first of all natural join of EMP and DEPT is to be done that involves costly Cartesian product. On the output the filtering will be done. Now see, in the predicate we have two conditions that are connected by AND. Each condition is applicable on only relation. Say, BASIC>50000 can be checked with EMP and TYPE='EMERGENCY' can be applied on DEPT. If the selections are done on the applicable relations then the outputs will have relatively less number of tuples. On those join operation can be done. So a better form will be:

$\sigma_{BASIC>50000}$ (EMP)* $\sigma_{TYPE='EMERGENCY'}$(DEPT)

If it is so that memory is not large enough to hold all the tuples of EMP and DEPT then EMP*DEPT will be costly. But reduction of tuples by early selection can help to address the issue. So, **perform select operation as early as possible.**

Note that, in place of AND if the logical operator is OR then it cannot be done.

b) Projection operation

Consider the relations DEPT(DCODE, DNAME, TYPE), EMP(ECODE, ENAME, BASIC, ADDRESS, PHONE, EMAIL, DOB, DCODE), NOMINEE(ECODE, NAME, RELATION, AGE, PHONE)

NOMINEE holds information of the nominees. An employee can have multiple nominees. Suppose, we want to display ECODE, ENAME of an employee and also corresponding nominee information. The query can be written as

$\Pi_{EMP.ECODE,\ ENAME,\ NAME,\ RELATION}$ (EMP * NOMINEE)

Schema of EMP being large, size of tuple obtained after Cartesian product of EMP and NOMINEE (while doing natural join) will also be large. So, memory requirement increases. But, we need a small part of EMP schema. Thus, the query can be modified as

$\Pi_{ECODE,\ ENAME}$(EMP) * $\Pi_{ECODE,\ NAME,\ RELATION}$ (NOMINEE)

We have applied projection operation as early as possible. Thus, in the join operation size of the schema gets reduced. While applying the projection on the relations, **attributes required in the output and those required for subsequent processing are chosen**. Please note, along with NAME and RELATION, ECODE is also chosen from NOMINEE as it will be required for join operation. So, **perform projection operation as early as possible.**

There may be combination of selection and projection. Suppose we want to find out the nominee information of employees in the department with DCODE = D1. Thus the query is $\Pi_{EMP.ECODE,\ ENAME,\ NAME,\ RELATION}$ ($\sigma_{DCODE='D1'}$ (EMP * NOMINEE))

Better form may be $\Pi_{ECODE,\ ENAME}$($\sigma_{DCODE='D1'}$ (EMP)) * $\Pi_{ECODE,\ NAME,\ RELATION}$ (NOMINEE)

c) Join operation

Suppose, R1, R2 and R3 are three relations and we want to perform R1*R2*R3. Join operation being associative (R1*R2)*R3 is same as R1*(R2*R3). Assume it is possible to

join R1 and R2 and same is true for R2 and R3. If it is so that R1*R2 results into less number of tuples than that of R2*R3 then R1*R2*R3 should be evaluated as (R1*R2)*R3. It helps to reduce the size of intermediate result. Suppose we want find out the employee details along with nominee details for the department with name XYZ. The query is of the form $\sigma_{DNAME='XYZ'}$ (EMP * DEPT * NOMINEE). For simplicity projection is ignored. Applying early selection, the query can be transformed as
EMP* $\sigma_{DNAME='XYZ'}$ (DEPT)*NOMINEE
Now, one can do EMP*NOMINEE first and there after apply join on the output and $\sigma_{DNAME='XYZ'}$ (DEPT). Here, the intermediate result may have number of tuples more than that of number of employees as an employee can have multiple nominees. On the other hand EMP* $\sigma_{DNAME='XYZ'}$ (DEPT) reduces number of tuples significantly. So the efficient form is (EMP* $\sigma_{DNAME='XYZ'}$ (DEPT))*NOMINEE

Second Phase of Optimization / Detailed Strategy

While forming the detailed strategy the goal is to reduce the query processing cost. Cost may be in the form number of disk access required (it is linked with time requirement. Disk access is slow process. More the number of access more is time requirement), size of the intermediate result (as memory size is a constraint) etc. Cost estimation requires different statistics like number of tuples in the relations, size of the tuples, and distribution of the attribute values (number of tuples for a particular value of an attribute and it is very difficult to estimate). Based on the estimates strategy is formulated. As all the statistics can be estimated in a definitive manner the strategy formed always may not be the best. But it is an efficient one. We will not focus much on cost estimation. Mostly we will try to reduce number of disk access.

a) **Strategy for select operation**
   i) **For Simple criteria i.e. search condition involves a single attribute**
      For different scenario some of the possible strategies may be as follows.
      - In any case **linear search** may be applied. Retrieve every record from the file and test whether attribute value satisfies the criteria or not. It can be used in all cases. But depending on the situation, subsequent strategies are more efficient.
      - Suppose, search criteria is for a particular value of a key attribute (equality comparison) and the file is ordered on the attribute. Then apply **binary search**.
      - Suppose the search is based on the equality comparison of key-attribute and primary index exists. Then use **primary index**. If hash organization on the key exists then hash based access can be followed.

- If search criteria involves >, >=,<, <= on key attribute with primary index then use **primary index to retrieve multiple records**. Suppose, records with key-attribute>=v are to be retrieved. Use primary index to find the record with key-attribute=v. Then, all subsequent records are with value > v. All preceding records are with value <v. So, retrieve as per the requirement.
- If the search criteria involves equality comparison on a non-key attribute and there exists clustering index then use **clustering index to retrieve all the records**. Find the corresponding entry into the index to obtain the starting data block for the value and access subsequent blocks till a different value is obtained.
- If a **secondary index** (may be B/B+ tree based) exists for the attribute then use it. If the attribute is a key then single record is retrieved else multiple records are retrieved. Can also be used for comparison involving <, <=,>,>=.

It may be noted that using primary index, clustering index and secondary index it is possible to perform **range queries** of the form v1<=attribute<=v2. V1 and V2 are the value of attribute.

ii) **For Complex criteria i.e. search conditions involve multiple simple criteria**
Suppose, search criteria is a conjunctive condition i.e. number of simple conditions are connected by the logical operator AND. Depending on the situation following strategies can be adopted for efficient use.

- **Using single index**: Suppose index exists for an attribute in any simple condition. Use corresponding index (strategies discussed earlier) to retrieve the records satisfying the simple condition. On the retrieve records verify other conditions.
- **Using composite index**: If multiple attributes are involved in conjunctive conditions and an index exists with composite indexing field (those multiple attributes or its subset) then use it. If the indexing field is subset of the attributes in conjunctive condition, then check the remaining criteria on retrieved records.
- **Using multiple indices**: If number of secondary indexes are present on more than one of the attributes involved in the simple conditions of conjunctive conditions then retrieve set of record pointers using individual index. Finally take the intersection of record pointers and retrieve the records corresponding to those pointers. For the additional

attributes in the conjunctive condition, if any, for which there was no index check the criteria on the retrieved records.

b) Strategy for join operation

In general, join requires a Cartesian product and subsequent filtering. Evaluation of Cartesian product is costly. The target is to reduce the number of disk block access. Remember with disk the basic unit of transfer is block. Depending on the situation suitable one from the following strategies may be adopted.

i) Simple iteration

Suppose we want to perform join operation on the relations R_1 and R_2. Simplest algorithm will be as follows.

```
for every tuple $t_1$ of $R_1$
 {
   For every tuple $t_2$ of $R_2$
    {
      Check $t_1$ and $t_2$ and decide whether it will be in the result or not
     }
   }
```

As it does not make any assumption regarding the storage of the tuples of the relation, for every tuple it needs to access a disk block. Let n_1 and n_2 are the number of tuples in R_1 and R_2 respectively. All the tuples of R_1 in the outer loop will be read only once. Thus, it requires n_1 disk block access. Inner loop is executed n_1 times. In each iteration of outer loop, all the tuples of R_2 are read. Reading all the tuples of R_2 once requires n_2 disk block access. It is to be done n_1 times and leads to n_1*n_2 block accesses. So, total number of block accesses is $n_1+n_1*n_2$.

Choice of relations in inner and outer loop is important. Suppose in the outer loop instead of R_1, R_2 is used and in the inner loop R_1 is considered. Then total number of block accesses is $n_2+n_2*n_1$. If $n_2<n_1$ then, this ordering requires less block access. Thus, it seems that using the relations with less number of tuples in the outer loop is better.

But, if one relation (say R_2) is so small that all the tuples can be read and stored into memory. Then the algorithm may be as follows.

```
read all the tuples of $R_2$ in the memory
for every tuple $t_1$ in $R_1$
 {
   For every tuple $t_2$ in $R_2$ that are already in memory    //$R_2$ is smaller
    {
      Check $t_1$ and $t_2$ and decide whether it will be in the result or not
    }
   }
```

Here, all the tuples of R_1 and R_2 are read only once requiring n_1+n_2 block accesses. Note, in the inner loop the smaller relation has been used (contrary to previous findings. Mind it, it can be done only if all the tuples of smaller relation fit in to memory) and its tuples are read from the memory (brought into memory at the very beginning). Hence, does not require further disk access.

ii) Block oriented iteration

It can be applied provided the tuples of R_1 are physically stored together and those of R_2 are also stored together physically. By physically stored together it is meant that a block contains tuples of uniform type. Block does not hold tuples of different relations. Thus, reading a block provides number of tuples of a relation in the memory. Exploiting this simple iteration that works on per tuple basis can be modified to process on per block basis. The algorithm is as follows.

```
for every block $b_1$ of $R_1$                 // here, disk access occurs
 {
   For every block $b_2$ of $R_2$          //here also, disk access occurs
    {
      for every tuple $t_1$ of $b_1$      // No disk access as $b_1$ is in memory
       {
         for every tuple $t_2$ of $b_2$    // No disk access as $b_2$ is in memory
          {
            Check $t_1$ and $t_2$ and decide whether it will be in the result or not
          }
       }
    }
 }
```

Disk access is required when blocks are read into memory. When tuples are taken from memory blocks, no disk access is involved. Suppose, there are N_1 and N_2 blocks for R_1 and R_2. N_i can be computed as ceiling of (n_i/bfr_i). n_i is the number of tuples in R_i and corresponding blocking factor is bfr_i. All the blocks of the relation in outer loop (R_1 in our case) are read once and will require N_1 disk block

accesses. To read all the blocks of R_2 once, N_2 disk block access is required. In the inner loop, all the blocks of R_2 are to be read N_1 times leading to N_1*N_2 disk block accesses. So, total number of disk block access is $N_1+ N_1*N_2$.
Note that, instead of reading a tuple in memory, reading a block reduces the number of disk access. Moreover, inner loop (that reads blocks of other relation) is repeated for number of blocks (instead of number of tuples) in outer relation. As in simple iteration, if the order of R1 and R2 are changed then number of disk block access will be $N_2+ N_1*N_2$. If $N_2<N_1$ then it is better. Using the smaller relation in the outer loop is better. But, if the smaller relation can be put into memory temporarily then it is better to use the same in inner loop. As in case of simple iteration, the algorithm for this scenario may be as follows. Assume R_2 be the relation of very small size.

```
Read all the disk blocks of $R_2$ in memory   // disk access occurs
 for every block $b_1$ of $R_1$                // here, disk access occurs
  {
    For every in-memory block $b_2$ of $R_2$          //here no disk access occurs
     {
       for every tuple $t_1$ of $b_1$      // No disk access as $b_1$ is in memory
        {
          for every tuple $t_2$ of $b_2$   // No disk access as $b_2$ is in memory
           {
             Check $t_1$ and $t_2$ and decide whether it will be in the result or not
           }
        }
     }
  }
```

For both the relations, all the blocks are read once. So number of disk block access gets further reduced and it s N_1+N_2.

iii) Merge join

It can be applied only if both the relations are physically ordered in the same manner (ascending/descending) on the joining attribute and tuples of individual relations are physically stored together (a disk block holds tuples of one relation). Suppose R_1 and R_2 are to be joined. Let A be the joining attribute and tuples in both the relations are sorted in ascending order of A. The algorithm is as follows.

```
Let p1 and p2 points to first tuple of R_1 and R_2 respectively
While (p1!=null && p2!=null)
  {
    t1=*p1
    s1={t1}   // s1 is to hold a set of tuples from R_1 for a particular value of A
   p1++   // p1 points to next tuple of R_1
   // form the set of tuples of R_1 for a particular value of A
   while (p1!=null && t1[A]==(*p1)[A])
    {
     s1=s1 U {*p1}
    p1++
    }

  // skip the tuples of R_2 as long as value of A is smaller than the value we are
looking for i.e. t1[A]
   while(p2!=null && (*p2)[A]<t1[A])
    {
     p2++
    }
// as long as (*p2)[A] is same as t1[A] join t2 with every tuple in s1
   while(p2!=null && (*p2)[A]==t1[A])
    {
     for every tuple t in s1
      {
        Consider t and *p2 for output
      }
    p2++
  }

}
```

For both the relations all the tuples are read once. So, number of disk block access is $N_1 + N_2$ i.e. sum of the number of blocks in the relations.

iv) Index based

Block iteration can be applied if the tuples of individual relations are stored physically together. Merge join further requires the sorted order of tuples on the joining attribute. Moreover, joining criteria may be different and it is not possible that simultaneously tuples will maintain sorted order on multiple criteria. Prior to join, sorting the tuples according to suitable criteria is very costly. Only simple iteration does not make any assumption. But it requires large number of disk access. If an index (may be secondary) exists on joining attribute then it can be utilized. Very often the joining is based on primary key – foreign key of the participating relation. Suppose, R_1 and R_2 are to be joined and joining attribute is the key of R_1. An index on the key of R_1 exists. Then the strategy may be as follows.

If tuples of R_2 are not stored together:

```
for each tuple $t_2$ in $R_2$
 {
   find the corresponding tuple $t_1$ from $R_1$ using the index
   consider $t_1$, $t_2$ for the result
 }
```

Number of disk block access is n_2+n_2*k where, n_2 is the number of tuples in R_2 and k is the number of disk block access required for index based searching of a tuple in R_1.

If tuples of R_2 are stored together:

```
for each block $b_2$ in $R_2$
 {
  for each tuple $t_2$ in $b_2$  // b2 is in memory and does not require disk access
  {
    find the corresponding tuple $t_1$ from $R_1$ using the index
    consider $t_1$, $t_2$ for the result
  }
}
```

Number of disk block access is N_2+n_2*k where, n_2 is the number of tuples in R_2, N_2 is number of blocks for R_2, and k is the number of disk block access required for index based searching of a tuple in R_1.

For a B+ tree based indexing, k is depth of the tree +1.

v) Hash join

It is observed that suitable index can be created and used for efficient join. But instead of having the overhead of maintaining the index all throughout one can go for creating the hash on the fly as and when required. There is no need to retain it. The algorithm is as follows.

R_1 and R_2 are the relations to be joined and A is the joining attribute. Let h be the hash function that maps A to 1,2, … K. $H1_1$, $H1_2$, … $H1_k$ are the buckets of pointers to tuples in R_1. $H2_1$, $H2_2$, … $H2_k$ are the buckets of pointers to tuples in R_2.

```
//initially all the buckets are empty
for i=1 to k
  {
    $H1_i$={}
    $H2_i$={}
  }
// make the assignment to the buckets and it requires disk accessing to read all
tuples of the relations once
for each tuple t1 in $R_1$
 {
    i=h(t1[A])
    $H1_i$= $H1_i$ U {pointer to disk record corresponding to t1}
  }
for each tuple t2 in $R_2$
 {
    i=h(t2[A])
    $H2_i$= $H2_i$ U {pointer to disk record corresponding to t2}
  }
// perform join operation considering corresponding buckets $H1_i$ and $H2_i$
for i=1 to k
 {
  S1={}
  S2={}
  // prepare the set of tuples S1 and S2 corresponding to pointers in $H1_i$ and $H2_i$
  for each pointer p1 in $H1_i$
   {
    S1=S1 U {*p1}
   }
for each pointer p2 in $H2_i$
   {
    S2=S2 U {*p1}
   }
for each tuple t1 in S1
  {
   for each tuple t2 in S2
    {
     Check t1 and t2 whether they will appear in result or not
    }
```

```
        }
    }
```

Note that, all the tuples of the individual relations are read from the disk twice. It is read first time to prepare the buckets. If the tuples of individual relations are stored physically together then the block can be read and thereafter tuples in the block can be accessed without further disk access. As the buckets store the pointers to the tuples (not the tuples), space requirement is less and can be placed into memory. Finally, considering corresponding buckets of the relations join is done. Each tuple of a relation is mapped to one bucket. Tuples of the two relations that can qualify to get joined are bound to be in corresponding buckets. Taking the pointers from such buckets tuples from the relations are accessed to form S1 and S2. This process is repeated for all mapped values (i.e. i =1 to k). During this stage all the tuples are read again. As tuples at this point are accessed using the pointer, block wise reading may not be useful. Each tuple is to be accessed individually as per the described process. In general, all the tuples are read twice. Note that, tuples t1 and (taken from S1 and S2) are further checked for inclusion in the output. It is so because multiple values of A may be mapped to same value.