

Query Processing & Optimization

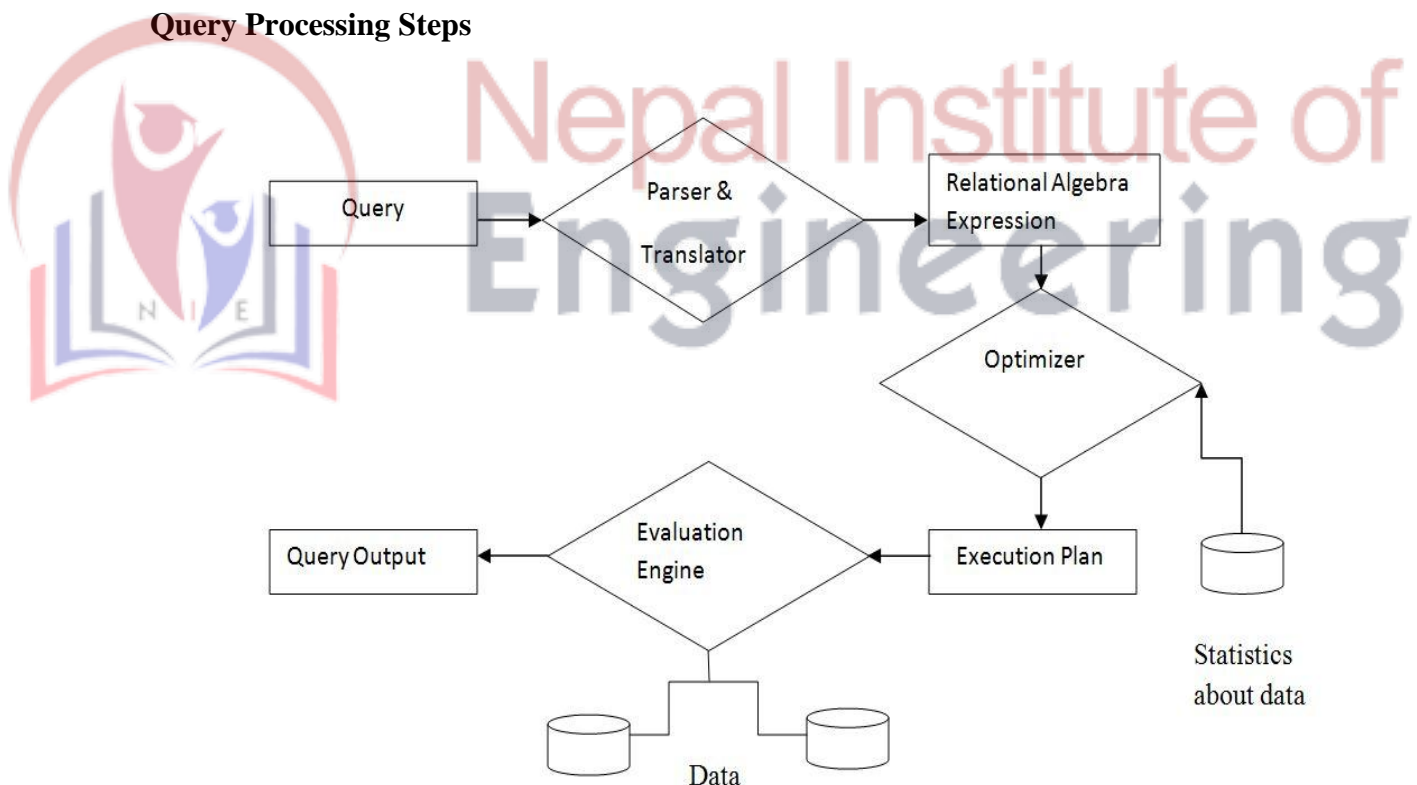
Query Processing Overview

The main goal of creating a database is to store the related data at one place, access and manipulate them as and when it is required by the user. Accessing and manipulating the data should be done efficiently i.e.; it should be accessed easily and quickly. Query Processing would mean the entire process or activity which involves query translation into low level instructions, query optimization to save resources, cost estimation or evaluation of query, and extraction of data from the database. It includes translation of high-level queries into low-level expression that can be used at the physical level of the file system, query optimization and actual execution of the query to get result.

Query Optimization

It is the process in which multiple query execution plans for satisfying a query are examined and a most efficient query plan is identified for execution.

Query Processing Steps



The steps involved in processing a query are :

1. Parsing & translation
2. Optimization
3. Evaluation

1. Parsing & Translation

In this step, the parser of the query processor checks and verifies the syntax of the query, the user's privilege to execute the query, the table names and attributes name etc. The correct table names, attribute names and the privilege of the users can be taken from the system catalog (data dictionary).

Parser performs the following checks as (refer detailed diagram):

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

Translation involves conversion of high level query to low level instruction in relational algebra.

Example : Select book_title, price From Book

Where price > 400

This query can be translated into either of the following relational-algebra expressions:

- $\pi_{\text{book_title, price}} (\sigma_{\text{price} > 400} (\text{Book}))$
- $\sigma_{\text{price} > 400} (\pi_{\text{book_title, price}} (\text{Book}))$

2. 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. Optimizer uses the statistical data stored as part of data dictionary. The statistical data are information about the size of the table, the length of records, the indexes created on the table, etc. Optimizer also checks for the conditions and conditional attributes which are parts of the query.

Execution Plan : The query processor module, at this stage, using the information collected in query optimization to find different relational algebra expressions that are equivalent and return the result of the one which we have written already. For our example, the query written in Relational algebra can also be written as the one given below;

$$\sigma_{\text{price} > 400} (\pi_{\text{book_title, price}} (\text{Book}))$$

3. Evaluation

It takes a physical query plan i.e. evaluation plan, executes that plan and return the result. It is used to fully specify how to evaluate a query, each operation in the query tree is annotated with instruction which specify the algorithm or the index to be used to evaluate that operation.

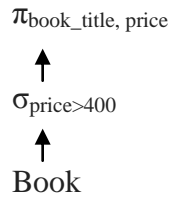


Fig : Query Tree

(Note : Different evaluation plans for a given query can have different cost. It is the responsibility of query optimizer to generate a least costly plan.)

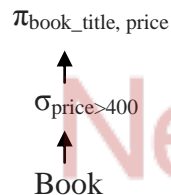
(Best query evaluation plan is finally submitted to the query evaluation engine for actual execution.)

Query Tree : It is used to represent relational algebra expression.

It has - Tree data structure

- I/P relations are represented as leaf node
- Relational algebra operations as internal node

Example :

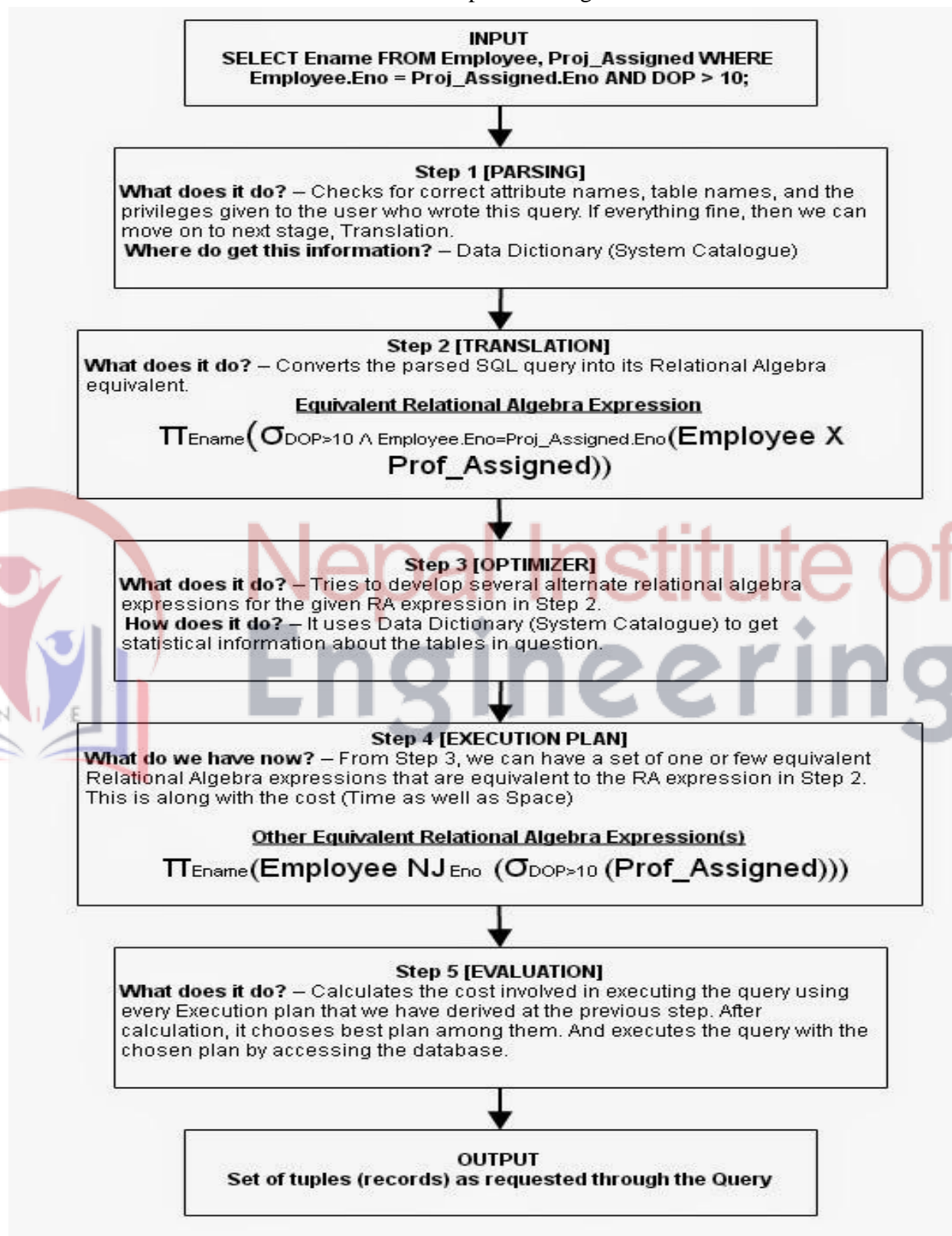


Query Blocks : Query submitted to the database system is first decomposed into query blocks. A query block from a basic unit that can be translated into relational algebra expression and optimized.

Example : **Select book_title From Book** ← Query Blok One

Where price > (Select Min(price) From Book ← Query Block Two
Where category = 'nobel')

The overall information discussed above are depicted in Figure 2 below :



5.1 Query Cost Estimation

Cost is generally measured as total time elapsed for answering query. To convert high level query to desired query we need some measurements.

Basic measure for query cost are

- disk access
- CPU cycle
- Transit time in network

Here CPU cost is difficult to calculate.

For disk access, how many disk access required to convert the high level query to desired query.

For cpu cycle, how many cpu cycle are consumed to evaluate desired query. CPU speed increases at faster rate than disk speed. Due to which cpu cost is relatively lower than disk cost.

Transit time in network

It is primarily considered with parallel/distributed system.

Disk Access Cost

Cost is measured by taking :

- no of seeks (no of attempts)
- no of block read
- no of block write

Normally, $\text{cost}(w) > \text{cost}(r)$

$\text{cost}(w)$: writing means continuously checking before writing or executing the program.

$\text{cost}(r)$: only once you will read.

To calculate disk access cost

No of seeks(N) ; $\text{Cost} = N * \text{Average seek time}$

No of block read ; $\text{Cost} = N * \text{Average block read cost}$

No of block write ; $\text{Cost} = N * \text{Average block write cost}$

For simplicity, we use

No of blocks transfer from disk and no of seeks as measure of cost.

t_T = time to transfer 1 block

t_S = time for 1 seek

So, cost for b block transfer plus s seeks

$$= b * t_T + s * t_S$$

5.2 Query Operation

Implementation of select operation in query processing

In query processing, the file scan is the lowest level operator to access data. File scans are search algorithm that locate and retrieve records that fulfill a selection condition.

A. Simple select operation (No logical operators such as AND, OR, NOT)

A1. Linear search

All records are scanned to see whether they satisfy the selection condition. This algorithm is simple and slower than other algorithm for implementing selection. It can be applied to any file regardless of the ordering of file, availability of indices or the nature of selection operation.

There can be 2 cases :

Whether the record is at 1st position or at the last position in relation.

If there are b_r number of blocks , then

Avg. Cost = $(b_r/2)$

Worst case

Cost = b_r

A2 Binary search

It is used if file is ordered on some attributes and select condition involves an equality comparison on that attributes.

Book

BID	BPrice
A01	100
A02	200
A03	300

$\sigma_{BPrice = 200}(\text{Book})$

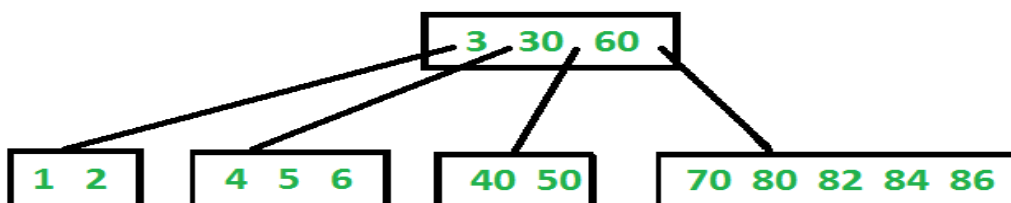
Cost = $\log_2(b_r)$

A3 Use of primary index, Equality on key attribute

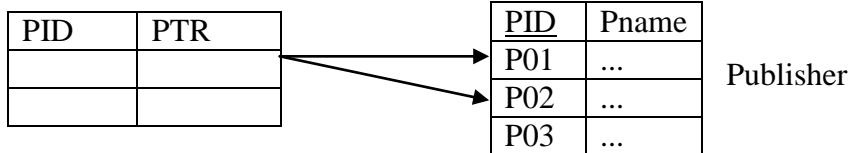
Select condition involves an equality comparison on a key attribute with primary index. Here we use B+ tree for maintaining the index.

Primary index is defined on an ordered data file. The data file is ordered on a key field.

(B+ Tree is a file organization method, similar to binary search tree but it can have more than two leaf node. It stores all the records only at the leaf node. Intermediary nodes will have pointer to the leaf node. They do not contain any data record.)



$[\sigma_{PID = 'P01'} (\text{Publisher})]$



PTR points to PID

Index is on key attribute.

Cost to find the PID in index is height of the tree.

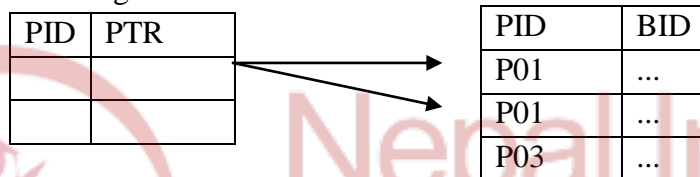
Cost = (height of tree + 1)

There will be no data block transfer, where relevant record will exist.

A4 Use of clustering index, equality on non-key attributes

Select condition involves an equality comparisons on the non-key attributes with clustering index.

Clustering index



There can be one publisher that publish more than one book.

Cost = height of tree + B

Where B - no of blocks satisfying the condition.

A5 Use of secondary index, equality on key or non-key attributes

In this case, when search key is candidate key, single record can be retrieved.

Here,

$$\text{Cost} = (H+1) * (t_T + t_S)$$

And when search key is not candidate key, multiple records can be retrieved.

Here,

$$\text{Cost} = (H+B) * (t_T + t_S)$$

A6 Use of primary index

If selection condition involves comparison like $A > V$, $A \geq V$, $A < V$ or $A \leq V$.

A is the attribute with primary index.

V is the attribute value.

$$\text{Avg. Cost} = (b_R/2) + H$$

Where H is height of tree.

Example : $\sigma_{PID \geq P02} (\text{Publisher})$

Then it will search index (PID=P02)

PID	Pname
P01	...
P02	...
P03	...
....

If we take $[\sigma_{A \leq v}(\text{Relation})]$, then we don't search index. We just start from top and up to the record that satisfies the condition we take that all record.

A7 Use of secondary index

It can also be used for comparison.

Cost = disk access, since records may be on different blocks.

-Much expensive

Implementing Complex Select Operations

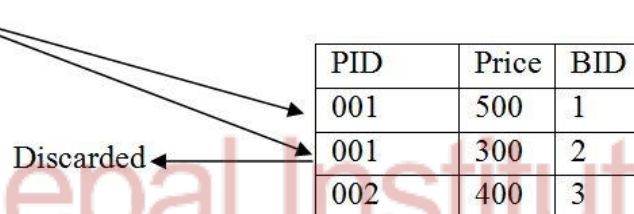
Use of logical operation AND(conjunction) or OR(disjunction)

A8 Conjunctive selection using one index

If an attribute involved in conjunctive condition has index, any algorithm from A2 to A7 can be used.

Example : $\sigma_{PID = '001' \wedge Price > 400}(\text{Book})$

(PID=001)Clustering index



PID	Price	BID
001	500	1
001	300	2
002	400	3

A9 Conjunctive Selection Using Composite Index

If the select operation specifies an equality condition on 2 or more attributes and combination of those attributes is/are composite index, the index can be searched directly to retrieve records. The type of index determine which of algorithm A3, A4 or A5 will be used.

Example : $[\sigma_{PID = '001' \wedge BID = 'B01'}(\text{Book})]$

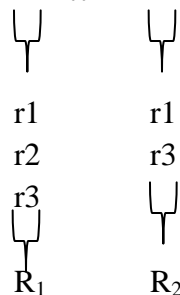
Composite index (PID , BID), We can use this composite index to directly retrieve the records.

A10 Conjunctive Selection By Intersection of Records Pointers

Each index is used to retrieve the set of record pointers that satisfy an individual condition. The intersection of all the retrieved record pointer gives the pointer to the tuples satisfy conjunctive condition.

Example :

$\sigma_{PID = '001' \wedge BID = 'B01'}(\text{Book})$



Join Operation(Optional)

Different algorithms for computing the join of relations are :

- Nested Loop Join
- Merge Join
- Hash Join

Nested Loop Join (NLJ)

In this method, when we join the tables, inner and outer loops based on the table records are created. The condition is tested in the inner most loop and if it satisfies, then it will be stored in the result set, else next record will be verified. The algorithm for this method can be written as below

For each record **t** **in** T

Loop

For each record **s** **in** S

Loop

Check θ on (t, s)

If matches **then** copy to result set

Else move to **next** record

End If

End Loop

End Loop

In above algorithm, we can see that each record of the outer table T is verified with the each record **s** of the inner table S. Hence it is very costly type of join. In the worst case, it requires $NT + BT$ seeks and $(NT * BS) + BT$ Block transfers. It is always better to put smaller tables in the inner loop, if it accommodated into its memory. Then the cost will reduce to 2 seeks and $BT + BS$ Block transfers.

Suppose we have EMPLOYEE and DEPT tables with following number of records and blocks. Let us see their costs based on these numbers and the position of the larger and smaller tables. Observe case 1 and 2, where when smaller table DEPT is set as outer table, then the number of block transfer doubles where as number of seek reduces. In the case 3 where both the tables fit into the memory block, the number of seek is reduced to 2 and block transfer also considerably reduces. But we can observe the real difference when only smaller table DEPT fits into the memory and is considered as inner table in case 4. The thumb rule for using tables in outer loop is always smaller table. But when smaller table fits into the memory, then use it in inner loop.

				Worst Case Cost Estimate			
		Number of Records (N_r)	Number of Records Blocks (B_r)	EMPLOYEE in Outer Loop		DEPT in Outer Loop	
				Number of Seeks (N_r+B_r)	Block Transfer ($N_r * B_r+B_r$)	Number of Seeks (N_r+B_r)	Block Transfer ($N_r * B_r+B_r$)
Case 1	EMPLOYEE	10000	400	10,400	1,000,400	3,100	2,000,100
	DEPT	3000	100				
Case 2	EMPLOYEE	100	4	104	104	31	201
	DEPT	30	1				
				Best Case Cost Estimate (both table Blocks fit into Memory)			
		Number of Records (N_r)	Number of Records Blocks (B_r)	EMPLOYEE in Outer Loop		DEPT in Outer Loop	
				Number of Seeks (2)	Block Transfer B_r+B_r	Number of Seeks (2)	Block Transfer B_r+B_r
Case 3	EMPLOYEE	10000	400	2	300	2	300
	DEPT	3000	100				
				Best Case Cost Estimate (Only Smaller table Blocks fit into Memory)			
		Number of Records (N_r)	Number of Records Blocks (B_r)	EMPLOYEE in Outer Loop		DEPT in Outer Loop	
				Number of Seeks (2)	Block Transfer B_r+B_r	Number of Seeks (N_r+B_r)	Block Transfer ($N_r * B_r+B_r$)
Case 4	EMPLOYEE	10000	400	2	300	3,100	2,000,100
	DEPT	3000	100				

From above examples, we can understand that cost of nested loop join is the game of number of records, blocks and positioning of the tables. However, the cost of nested loop join is very expensive compared to other joins below.

Merge Join

Tables used in the join query may be sorted or not sorted. Sorted tables give efficient costs while joining. In this method, the column used to join both the tables is used to sort the two tables. It uses merge-sort technique to sort the records in two tables. Once the sorting is done then join condition is applied to get the result. Joining is also similar to any other joins, but if two records have same column value, then care should be taken to sort records based on all the columns of the record. This method is usually used in natural joins or equijoins. Assume that all the records can be accommodated into the memory block. The each block of records is read only once to join. Then the cost of this join would be

Cost of Seeks: $BT/M + BS/M$

Cost of Block Transfer: $BT + BS$

If tables are not sorted, then cost for merge sorting is also included in above cost.

Hash Join

This method is also useful in case of natural and equijoins. We use hash function h on the joining column, to divide the records of each tables into different blocks. Assume each of these hash divided block of records fit the memory block and we have NH number of hash divided blocks. DT_0, DT_1, DT_2, DT_N : are the hash divided blocks of table T , where $DT_i = h(RT(\text{joinCol}))$ and RT is in any one of the record in table T and is in partition DT_i .

DS_0, DS_1, DS_2, DSN : are the hash divided blocks of table S , where $DS_i = h(RS(\text{joinCol}))$ and RS is in any one of the record in table S and is in partition DS_i .

When the tables are partitioned, one memory block is reserved for the input and one memory block is reserved for the output buffer of each partition. Rest of the memory blocks are allocated for the input buffers and outputs equally. Then the hash algorithm can be written as below

For each partition i Loop

Load DS_i into memory and build a hash index on the joining column

Retrieve each record DT_i from memory, for each record in DS_i build a hash index on joining column and find the matching record in DS_i

If matching records are found, then join the records RT and RS into output

End loop

End Loop

For example, take EMP and DEPT tables, with joining column as DEPT_ID. A hash function on DEPT_ID is used to generate the hash divided block (hash partition) for each of the table. Suppose it generates 5 partitions for each table. According to the algorithm,

If total number of memory block is M , and the number of hash partition is i , then

- $i < M$: Above hash algorithm is used to join the tables.
- $i \geq M$: Recursive partitioning method is used to join the tables. That is, both tables are partitioned into $M-1$ blocks, and proceed with joining and repeat the partition again for rest of the blocks.

There are some cases where some bad hash function or because of join condition, we will get multiple same records. In such case, it is difficult to accommodate all the records into one partition block. This is called hash table overflow. This can be avoided by carefully partitioning the tables. This can be resolved while creating the partition itself by creating sub partitions using another hash function. But both the tables should be partitioned in same way. But we will still have overflow, if tables have lot many duplicate values and in such case we have to use BNLJ method.

There is another issue called skewed partition where some partition will have relatively large number of records compared to other partitions. This is also not good for joining the tables. Let us see the cost of hash joins.

Let us consider an example. Assume

Total number of blocks in memory, $M = 20$

Number of blocks in EMP, $BT = 100$

Number of blocks in DEPT, $BS = 400$

Total number of Partition, $I = 5$

Then, each partition of EMP will have 20 blocks each and DEPT will have 80 each. That means in the case of EMP $i = M$.

Hence, total Seeks = $2((BT/B) + (BS/B)) = 2((100/5) + (400/5)) = 336$

Total block transfer = $3(BT+BS) + 4 \cdot N_i = 3(100 + 400) = 1500$, where $4N_i$ is ignored because we did not consider partially filled partitions.

Evaluation of Expressions

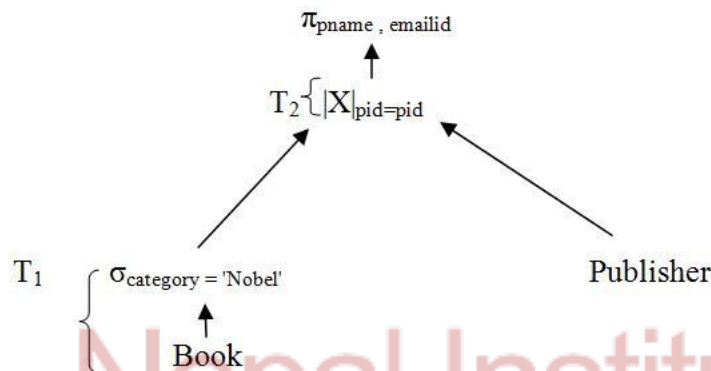
The obvious way to evaluate an expression is simply to evaluate one operation at a time, in an appropriate order. There are 2 general approaches while evaluating an expression.

- **Materialization**

Each operation in the expression is evaluated one by one in appropriate order and result of each operation is materialized (created) in a temporary relation which becomes input for subsequent operations.

Example : $\pi_{pname, emailid} (\sigma_{category = 'Nobel'} (Book) \bowtie Publisher)$

It's expression tree is



From above example, the relation created by the selection on Book relation will be temporary relation and then join will be evaluated between temporary relation and Publisher relation, which further gives another temporary relation.

By repeating the process, we evaluate the operation at root of the tree that gives the final result.

A disadvantages to this approaches is the need to construct the temporary relations, which (unless they are small) must be written to disk.

Cost : Generally the cost of evaluating an expression is the addition of cost of all operation and cost of writing intermediate result to disk.

We assume that records of the result accumulate in a buffer and when the buffer is full, they are written to disk. The number of block written out 'b_r' can be estimated as n_r/f_r, where n_r is the estimated number of tuples in result relation r and f_r is the blocking factor i.e., number of records r that fit in a block. In addition to transfer time, some disk seeks may be required, since the disk head may have moved between successive writes. The number of seeks can be estimated as (b_r/b_b) where b_b is the size of output buffer (measured in blocks).

Double Buffering : (Using two buffers, with one continuing execution of the algorithm while the other is being written out) allows the algorithm to execute more quickly by performing CPU activity in parallel with I/O activity. The number of seeks can be reduced by allocating extra blocks to the output buffer and writing out multiple blocks at once.

- **Pipelining**

This is an alternative approaches used to evaluate several operations simultaneously with the results of operation passed on to the next without the need to store a temporary relation.

Example : Consider the expression $\pi_{a_1, a_2}(r \bowtie s)$. In the case of materialization, evaluation would involve creating temporary relation to hold result of join and then read back in the result to perform projection. But in the case of pipelining, when the join operation generates a tuple of its result, it passes that tuple immediately to project operation for processing. By combining join and projection, we avoid creating intermediate result.

Advantages

- Reduces the cost of query evaluation by eliminating the cost of reading and writing temporary relations.
- It can start generating query result quickly, if a root operator of a query evaluation plan is combined in a pipeline with its inputs. This can be quite useful if the results are displayed to a user as they are generated since otherwise there may be a long delay before the user sees any query result.

Pipelines can be executed in 2 ways :

Demand driven or Lazy Evaluation

- The system keeps requests for tuples from the operation at the top of the pipeline.
- Each operation requests next tuple from children operations as required in order to output its next tuple.
- If the inputs of the operation are not pipelined, the next tuple to be returned can be computed from the input relations, while the system keeps track of what has been returned so far. If it has some pipelined inputs, the operation also makes requests for tuple from its pipelined inputs, the operation computes for its output and pass then to its parents.

Implementation

Each operation in demand driven pipeline can be implemented as an iterator that provides the following :

- **Open ()**
Ex1 :- file scan : initializes file scan , store pointer to beginning of the file as state
Ex2 :- merge join : sort relation and store pointers to beginning of sorted relation as state.
- **Next()**
Ex1 :- file scan : output next tuple and advance and store file pointer.
Ex2 :- merge join : continue with merge from earlier state till next output tuple is found, save pointer as iterator state.
- **Close()**
The function close() tells iterator that no more tuples are required.

Producer driven or Eager Pipelining

Here, operators do not wait for request to produce tuples but instead generate the tuples eagerly. Each operation is modeled as separate process or thread within the system that takes a stream of tuples from its pipelined inputs and generates a stream of tuples for its output.

Implementation

For each pair of adjacent operations in a producer driven pipeline, the system creates a buffer to hold tuples being passed from one operation to the next. The operations at bottom continually generates output tuples and put them in its output buffer until it is full. An operation at any other level of a pipeline generates output tuples when it gets input tuples from lower down in pipeline until its output buffer is full. Once the operation uses a tuple from a pipelined input, it removes the tuple from its input buffer. In either case, once the output buffer is full, the operation waits until its parent operation removes tuples from the buffer, so that the buffer has space for more tuples. At this point, the operation generates more tuples, until the buffer is full again. The operation repeats this process.

Query Optimization

The process of selecting the most efficient query evaluation plan from among the many strategies usually possible for processing a given query especially if the query is complex.

Example : Relation schema

instructor (ID , name , dept_name , salary)

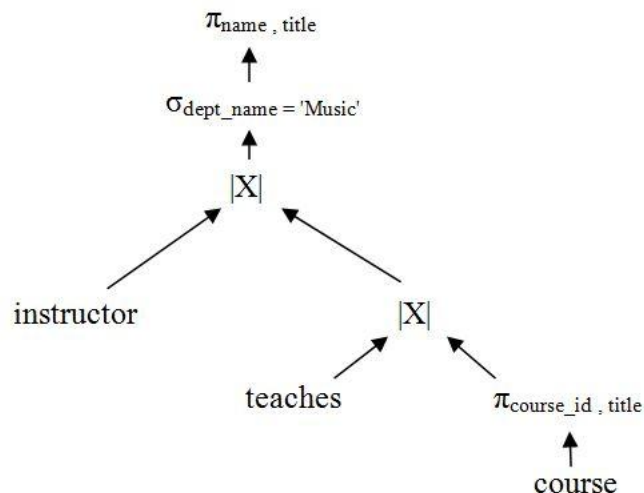
teaches (ID , course_id , sec_id , semester , year)

course (course_id , title , dept_name , credits)

Q. Find the names of all instructors in the music department together with the course title of all the courses that the instructor teaches?

$\pi_{name, title} (\sigma_{dept_name = 'Music'} (instructor \bowtie (teaches \bowtie \pi_{course_id, title} (course))))$

It's expression tree is :



Transformed relation is :

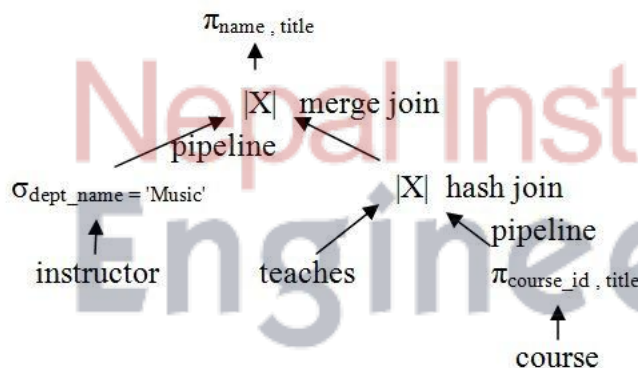
$\pi_{name, title} ((\sigma_{dept_name = 'Music'}(instructor)) \bowtie [X] (teaches \bowtie [X] \pi_{course_id, title} (course)))$ [Using equivalence rule 8(a)]

which is equivalent to our original algebra expression but it generates smaller intermediate relations.

Here an evaluation plan defines exactly what algorithm should be used for each operation and how the execution of the operation should be coordinated.

For a given relational algebra expression, it is the job for query optimizer to come up with a query optimizer evaluation plan that computes the same result as the given expression and is the least costly way of generating the result. Generation of query evaluation plans involve three steps:

1. Generating expression that are logically equivalent to the given expression.
2. Annotating the resultant expression to generate query evaluation plan.
3. Estimating the cost of each evaluation plan and choosing the one whose estimated cost is the least.



Implement First Step, the query optimizer must generate expressions equivalent to a given expression. It does so by means of equivalence rules that specify how to transform an expression into a logically equivalent one.

Transformation of relational expression

A query can be expressed in several different ways with different cost of evaluations. Here, two relational algebra expressions are said to be equivalent if, on every legal database instance, the two expressions generate the same set of tuples.

Equivalence Rules

It says that expressions of two forms are equivalent.

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections. It is referred to as cascade of σ .

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

2. Selection operations are commutative.

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

3. Only the final operations in a sequence of projection operations are needed; the others can be omitted. This transformation can be referred to as a cascade of π .

$$\pi_{L1}(\pi_{L2}(\dots(\pi_{Ln}(E))\dots)) = \pi_{L1}(E)$$

4. Selections can be combined with cartesian product and theta join.

a) $\sigma_{\Theta}(E_1 \bowtie E_2) = E_1 \bowtie \sigma_{\Theta} E_2$

b) $\sigma_{\Theta_1}(E_1 \bowtie_{\Theta_2} E_2) = E_1 \bowtie_{\Theta_1 \wedge \Theta_2} E_2$

5. Theta join operations are commutative.

$$E_1 \bowtie_{\Theta} E_2 = E_2 \bowtie_{\Theta} E_1$$

6. a) Natural join operations are associative.

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

- b) Theta joins are associative in the following manner.

$$(E_1 \bowtie_{\Theta_1} E_2) \bowtie_{\Theta_2 \wedge \Theta_3} E_3 = E_1 \bowtie_{\Theta_1 \wedge \Theta_3} (E_2 \bowtie_{\Theta_2} E_3)$$

Here, Θ_2 involves attributes from E_2 and E_3 only.

7. The selection operation distributes over the theta join operation under the following conditions :

- a. It distributes when all the attributes in selection condition Θ_0 involve only the attributes of one of the expressions (say E_1) being joined.

$$\sigma_{\Theta_0}(E_1 \bowtie_{\Theta} E_2) = (\sigma_{\Theta_0}(E_1)) \bowtie_{\Theta} E_2$$

- b. It distributes when selection condition Θ_1 involves only the attributes of E_1 and Θ_2 involves only the attributes of E_2 .

$$\sigma_{\Theta_1 \wedge \Theta_2}(E_1 \bowtie_{\Theta} E_2) = (\sigma_{\Theta_1}(E_1)) \bowtie_{\Theta} (\sigma_{\Theta_2}(E_2))$$

8. The projection operation distributes over the theta join operation under the following condition :

- a) Let L_1 and L_2 be set of attributes of E_1 and E_2 respectively. Suppose that the join condition Θ involves only attributes in $L_1 \cup L_2$. Then

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

- b) Consider a join $E_1 \bowtie_{\Theta} E_2$. Let L_1 and L_2 be sets of attributes from E_1 and E_2 , respectively. Let L_3 be the attributes of E_1 that are involved in join condition Θ , but are not in $L_1 \cup L_2$ and let L_4 be attributes E_2 that are involved in join condition Θ but are not in $L_1 \cup L_2$. Then ,

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

9. The set operations union and intersection are commutative.

$$E_1 \cup E_2 = E_2 \cup E_1$$

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

Set difference is not commutative.

10. Set union and intersections are associative.

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

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

11. The selection operation distributes over the union, intersection and set difference operation.

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

Similarly, the preceding equivalence with - replaced with either \cup or \cap , also holds. Further,

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

The preceding equivalence, with - replaced by \cap also holds, but does not hold if - replaced by \cup .

12. The projection operation distributes over union operation.

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

Example : Relation schema

instructor (ID , name , dept_name , salary)

teaches (ID , course_id , sec_id , semester , year)

course (course_id , title , dept_name , credits)

Q. Find the names of all instructors in the music department together with the course title of all the courses that the instructor teaches in 2009?

$$\pi_{name, title} (\sigma_{dept_name = 'Music' \wedge year=2009} (instructor \bowtie (teaches \bowtie \pi_{course_id, title} (course))))$$

Now we can't apply the selection predicate directly to instructor relation, since the predicate involves attributes of both the instructor and teaches relation.

By applying rule 6 (a) to transform the join

instructor \bowtie (teaches \bowtie $\pi_{course_id, title} (course)$) in to

(instructor \bowtie teaches) \bowtie $\pi_{course_id, title} (course)$

We get

$$\pi_{name, title} (\sigma_{dept_name = 'Music' \wedge year=2009} ((instructor \bowtie teaches) \bowtie \pi_{course_id, title} (course)))$$

Using rule 7 (a)

$$\pi_{name, title} ((\sigma_{dept_name = 'Music' \wedge year=2009} (instructor \bowtie teaches)) \bowtie \pi_{course_id, title} (course))$$

Using rule 1, we get

$$\pi_{name, title} ((\sigma_{dept_name = 'Music'} (\sigma_{year=2009} (instructor \bowtie teaches)) \bowtie \pi_{course_id, title} (course)))$$

Using rule 7(b), we get

$$\pi_{name, title} ((\sigma_{dept_name = 'Music'} (instructor) \bowtie \sigma_{year=2009} (teaches)) \bowtie \pi_{course_id, title} (course))$$

Choice of Evaluation Plan

1) Cost Based Optimization

This is based on the cost of the query. The query can use different paths based on indexes, constraints, sorting methods etc. This method mainly uses the statistics like record size, number of records, number of records per block, number of blocks, table size, whether whole table fits in a block, organization of tables, uniqueness of column values, size of columns etc.

Suppose, we have series of table joined in a query.

$$T1 \propto T2 \propto T3 \propto T4 \propto T5 \propto T6$$

For above query we can have any order of evaluation. We can start taking any two tables in any order and start evaluating the query. Ideally, we can have join combinations in $(2(n-1))! / (n-1)!$ ways. For example, suppose we have 5 tables involved in join, then we can have $8! / 4! = 1680$ combinations. But when query optimizer runs, it does not evaluate in all these ways always. It uses dynamic programming where it generates the costs for join orders of any combination of tables. It is calculated and generated only once. This least cost for all the table combination is then stored in the database and is used for future use. i.e.; say we have a set of tables, $T = \{ T1, T2, T3 \dots Tn \}$, then it generates least cost combination for all the tables and stores it.

Dynamic Programming

As we learnt above, the least cost for the joins of any combination of table is generated here. These values are stored in the database and when those tables are used in the query, this combination is selected for evaluating the query.

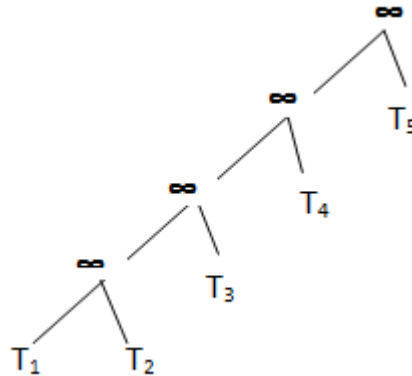
While generating the cost, it follows below steps :

Suppose we have set of tables, $T = \{T1, T2, T3 \dots Tn\}$, in a DB. It picks the first table, and computes cost for joining with rest of the tables in set T. It calculates cost for each of the tables and then chooses the best cost. It continues doing the same with rest of the tables in set T. It will generate $2n - 1$ cases and it selects the lowest cost and stores it. When a query uses those tables, it checks for the costs here and that combination is used to evaluate the query. This is called dynamic programming.

In this method, time required to find optimized query is in the order of 3^n , where n is the number of tables. Suppose we have 5 tables, then time required in $3^5 = 243$, which is lesser than finding all the combination of tables and then deciding the best combination (1680). Also, the space required for computing and storing the cost is also less and is in the order of 2^n . In above example, it is $2^5 = 32$.

Left Deep Trees

This is another method of determining the cost of the joins. Here, the tables and joins are represented in the form of trees. The joins always form the root of the tree and table is kept at the right side of the root. LHS of the root always point to the next join. Hence it gets deeper and deeper on LHS. Hence it is called as left deep tree.



Here instead of calculating the best join cost for set of tables, best join cost for joining with each table is calculated. In this method, time required to find optimized query is in the order of $n2^n$, where n is the number of tables. Suppose we have 5 tables, then time required is $5 \times 2^5 = 160$, which is lesser than dynamic programming. Also, the space required for computing storing the cost is also less and is in the order of 2^n . In above example, it is $2^5 = 32$, same as dynamic programming.

Interesting Sort Orders

This method is an enhancement to dynamic programming. Here, while calculating the best join order costs, it also considers the sorted tables. It assumes, calculating the join orders on sorted tables would be efficient. i.e.; suppose we have unsorted tables $T_1, T_2, T_3 \dots T_n$ and we have join on these tables.

$$(T_1 \bowtie T_2) \bowtie T_3 \bowtie \dots \bowtie T_n$$

This method uses hash join or merge join method to calculate the cost. Hash Join will simply join the tables. We get sorted output in merge join method, but it is costlier than hash join. Even though merge join is costlier at this stage, when it moves to join with third table, the join will have less effort to sort the tables. This is because first table is the sorted result of first two tables. Hence it will reduce the total cost of the query.

But the number of tables involved in the join would be relatively less and this cost/space difference will be hardly noticeable.

All these cost based optimizations are expensive and are suitable for large number of data. There is another method of optimization called heuristic optimization, which is better compared to cost based optimization.

2) Heuristic Optimization (Logical)

This method is also known as rule based optimization. This is based on the equivalence rule on relational expressions; hence the number of combination of queries get reduces here. Hence the cost of the query too reduces.

This method creates relational tree for the given query based on the equivalence rules. These equivalence rules by providing an alternative way of writing and evaluating the query, gives the better path to evaluate the query. This rule need not be true in all cases. It needs to be examined after applying those rules. The most important set of rules followed in this method is listed below:

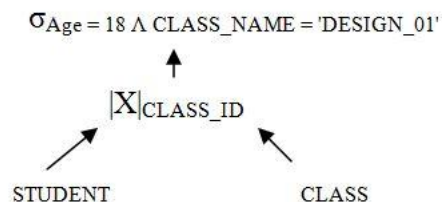
- Perform all the selection operation as early as possible in the query. This should be first and foremost set of actions on the tables in the query. By performing the selection operation, we can reduce the number of records involved in the query, rather than using the whole tables throughout the query.

Suppose we have a query to retrieve the students with age 18 and studying in class DESIGN_01. We can get all the student details from STUDENT table, and class details from CLASS table. We can write this query in two different ways.

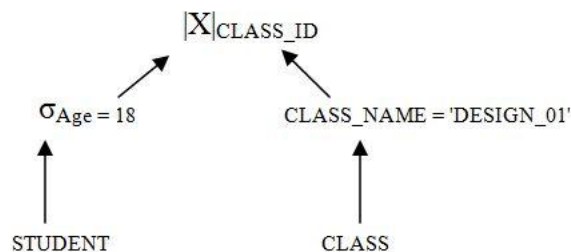
Here both the queries will return same result. But when we observe them closely we can see that first query will join the two tables first and then applies the filters. That means, it traverses whole table to join, hence the number of records involved is more. But he second query, applies the filters on each table first. This reduces the number of records on each table (in class table, the number of record reduces to one in this case!). Then it joins these intermediary tables. Hence the cost in this case is comparatively less.

Instead of writing query the optimizer creates relational algebra and tree for above case.

$\sigma_{\text{Age} = 18 \wedge \text{CLASS_NAME} = \text{'DESIGN_01'}} (\text{STUDENT} \bowtie_{\text{CLASS_ID}} \text{CLASS})$ **Not so efficient**



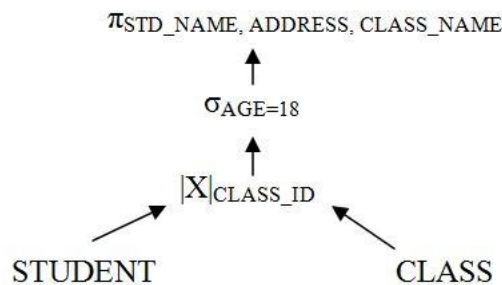
$\sigma_{\text{Age} = 18} (\text{STUDENT}) \bowtie_{\text{CLASS_ID}} (\sigma_{\text{CLASS_NAME} = \text{'DESIGN_01'}} \text{CLASS})$ **Efficient**



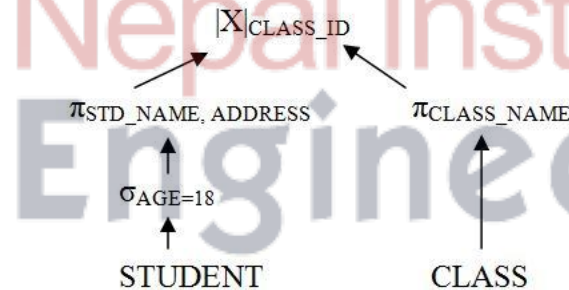
- Perform all the projection as early as possible in the query. This is similar to selection but will reduce the number of columns in the query.

Suppose for example, we have to select only student name, address and class name of students with age 18 from STUDENT and CLASS tables.

$\pi_{STD_NAME, ADDRESS, CLASS_NAME} (\sigma_{AGE=18}(STUDENT \bowtie_{CLASS_ID} CLASS))$ **Not so efficient**



$\pi_{STD_NAME, ADDRESS} ((\sigma_{AGE=18}(STUDENT)) \bowtie_{CLASS_ID} \pi_{CLASS_NAME}(CLASS))$ **Efficient**



Here again, both the queries look alike, results alike. But when we compare the number of records and attributes involved at each stage, second query uses less records and hence more efficient.

- Next step is to perform most restrictive joins and selection operations. When we say most restrictive joins and selection means, select those set of tables and views which will result in comparatively less number of records. Any query will have better performance when tables with few records are joined. Hence throughout heuristic method of optimization, the rules are formed to get less number of records at each stage, so that query performance is better. So is the case here too.

Suppose we have STUDENT, CLASS and TEACHER tables. Any student can attend only one class in an academic year and only one teacher takes a class. But a class can have more than 50 students. Now we have to retrieve STUDENT_NAME, ADDRESS, AGE, CLASS_NAME and TEACHER_NAME of each student in a school.

$\Pi_{STD_NAME, ADDRESS, AGE, CLASS_NAME, TEACHER_NAME}((STUDENT \bowtie_{CLASS_ID} CLASS) \bowtie_{TECH_ID} TEACHER)$ **Not So efficient**

$\Pi_{STD_NAME, ADDRESS, AGE, CLASS_NAME, TEACHER_NAME}(STUDENT \bowtie_{CLASS_ID} (CLASS \bowtie_{TECH_ID} TEACHER))$ **Efficient**

In the first query, it tries to select the records of students from each class. This will result in a very huge intermediary table. This table is then joined with another small table. Hence the traversing of number of records is also more. But in the second query, CLASS and TEACHER are joined first, which has one to one relation here. Hence the number of resulting record is STUDENT table give the final result. Hence this second method is more efficient.

- Sometimes we can combine above heuristic steps with cost based optimization technique to get better results.

All these methods need not be always true. It also depends on the table size, column size, type of selection, projection, join sort, constraints, indexes, statistics etc. Above optimization describes the best way of optimizing the queries.

Query Decomposition

Process of minimizing the resource usage.

Aims of query decomposition

- To transform a high level query in to a relational algebra query
- To check the query is syntactically and semantically correct.
- It is efficient way to retrieve data from database.

Example : Select * From staff as s , branch as b
 Where s.branchNo = b.branchNo And
 (s.position='Manager' And
 b.city ='London')

3 equivalent Relational Algebra

- $\sigma_{(position = 'Manager' \wedge staff.branchNo = branch.branchNo \wedge branch.city = 'London')}(staff \bowtie branch)$

Let we have 1000 records in staff & 50 records in branch. This query calculates the Cartesian product of staff and branch, (1000 + 50) disk accesses to read and creates relation with (1000 * 50) tuples. We then have to read each of these tuples again to test them against the selection predicates so it gives total cost of
 (1000 + 50) + 2 * (1000 * 50) = 101050 disk access.

- $\sigma_{(position = 'Manager') \wedge (branch.city = 'London')}(staff \bowtie_{staff.branchNo = branch.branchNo} branch)$
 2*1000 + (1000+50) = 3050 disk access

- $(\sigma_{(\text{position} = \text{'Manager'})}(\text{staff})) \bowtie_{\text{staff.branchNo} = \text{branch.branchNo}} (\sigma_{\text{city} = \text{'London'}}(\text{branch}))$
 $1000 + 2 * 50 + 5 + (50 + 5) = 1160$ disk access

Stages

1. Analysis
2. Normalization
3. Semantic Analysis
4. Simplification or redundancy eliminator
5. Query restructuring

Analysis

Query is syntactically analyzed

- verifies that the relation and attributes specified in the query are defined in the system catalog.
- verifies that any operation applied to database objects are appropriate for the object type.

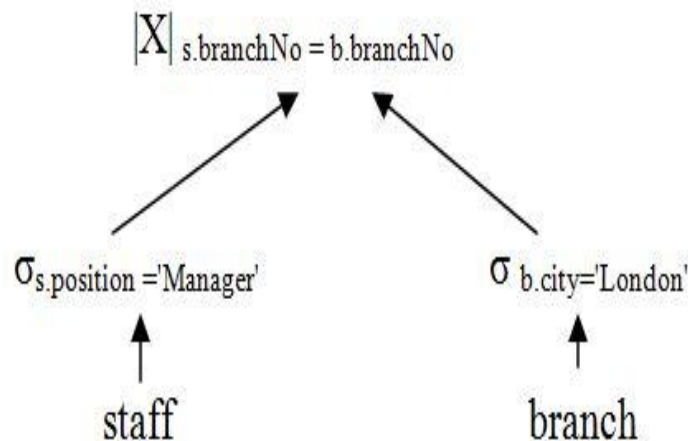
Example : Select staffNumber
 From staff
 Where position > 10 ;

This query would be rejected on 2 grounds.

- In select list the attributes staffNumber is not defined for staff relation (should be staffNo)
- In where clause comparison '>10' is incompatible with the datatype position which is a variable character string.

After completion of this stage, high level query has been transformed to relational algebra.

Query Tree



Normalization

- Normalizes query for easy manipulation.
- Arbitrarily complex predicate (in SQL, the where condition) can be converted in to one of 2 forms by applying few transformation rules.

I. Conjunctive Normal Form

A sequence of conjuncts that are connected with the AND operator. Each conjuncts contain one or more terms connected by the OR operator. Example : $(\text{Position} = \text{'Manager'} \vee \text{Salary} > 20000) \wedge \text{branchNo} = \text{'B003'}$

A conjunctive selection contains only those tuples that satisfy all conjuncts.

II. Disjunctive Normal Form

A sequence of disjuncts that are connected with the \vee (OR) operator. Each disjuncts contain one or more terms connected by the \wedge (AND) operator.

Example : Above conjunctive in to disjunctive normal form

$(\text{Position} = \text{'manager'} \wedge \text{branchNo} = \text{'B003'}) \vee (\text{Salary} > 20000 \wedge \text{branchNo} = \text{'B003'})$

Semantic Analysis

Rejects normalized query that are incorrectly formulated or contradictory.

Example : the predicate $(\text{position} = \text{'manager'} \wedge \text{position} = \text{'assistant'})$ or staff relation is contracting because a person can't be both manager and assistant.

Elimination of redundancy

Simplifying the qualification of user query to eliminate redundancy. Transform query to a semantically equivalent but more easily and efficiently computed form. Queries on relation that satisfy certain integrity and security constraints(access restriction).

○ Before elimination of redundancy

Select position From staff as s, branch as b

Where (not (b.position='manager'))

AND (b.position = 'manager' OR b.position = 'assistant') AND

NOT(b.position = 'assistant'))

OR(b.BID=s.SID AND s.Name = 'ABC')

○ After elimination of redundancy

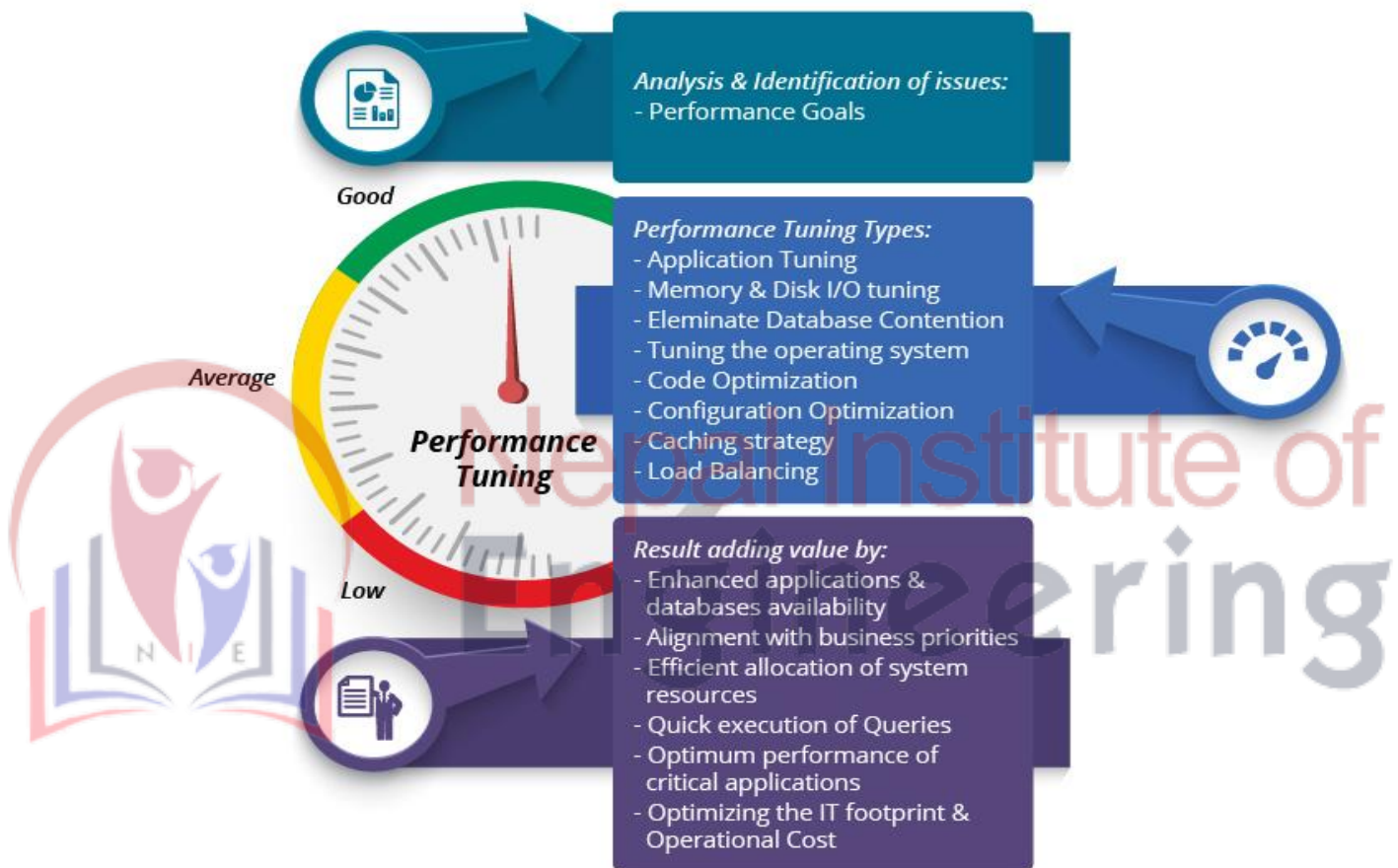
Select b.position From staff as s , branch as b Where b.BID =s.SID AND s.Name='ABC'

Query Restructuring

It is the final stage where query is restructured to provide a more efficient implementation.

Performance Tuning

Performance tuning is a process which focuses on improving the system response time without the need of changing or upgrading its configuration. Tuning becomes essential when the system indicates sluggish or becomes absolutely unresponsive. Usually, this happens due to increased load with some degree of decreasing performance. Thus, Corporations can save a lot of money using Performance tuning just by modifying a system to handle higher loads and thereby enhance the server performance without spending on new Infrastructure or applications.



Types of Performance Tuning

- **Application Tuning**

Approximately 80% of all Oracle system performance problems are resolved by coding optimal SQL.

- **Memory & Disk I/O Tuning**

Tuning to prevent frequent reloads. Properly sizing of database buffers (shared pool, buffer cache, log buffer, etc) is required. In Disk I/O tuning the files are properly sized and placed to provide maximum disk & sub disk throughput

- **Eliminate Database Contention**

Here we eliminate contentions like database locks, latches and wait events. Data access path, data partitioning, data replication, etc are also taken care

- **Tuning the Operating System**

Monitor and tune operating system CPU, I/O and memory utilization

- **Code Optimization**

It includes improving the code so that not many loops are involved and the code is executed outside the loop wherever possible

- **Configuration Optimization**

Configuration tuning includes improving the performance of the application finding the best configuration for complex applications like Big Data.

- **Caching Strategy**

It is used to remove the performance bottleneck. It improves performance by retaining frequently used information in high-speed memory, reducing access time and avoiding repeated computation.

- **Load Balancing**

This arrangement results in the utilization of systems equally for addressing those many service requests, resulting in proper utilization of resources without any system remaining idle.

Our Approach

Identify the Problem Component

This stage includes understanding the nature of the problem. In-depth analysis and system audits are conducted to identify the problem and performance obstacle. A detailed report with current system health, insights and recommendations are mentioned.

Performance Goals

Thorough System