Dynamic Programming Solution for Query Optimization in Homogeneous Distributed Databases

Ms. Anju Mishra
Department of Computer Application, IEC-CET, Greater Noida

Ms. Gunjan Nehru
Department of Computer Application, IEC-CET, Greater Noida

Mr. Ashish Pandey
Sapient Consulting, Gurgaon

Abstract

Due to new distributed database applications such as huge deductive database systems, the search complexity is constantly increasing and we need better algorithms to speedup traditional relational database queries. Now a days distributed database applications are applied on Heterogeneous distributed database systems and developing Homogeneous distributed databases. The “multiple query optimization” (MQO) tries to reduce the execution cost of a group of queries by performing common tasks only once, whereas traditional query optimization considers a single query at a time. An optimal dynamic programming method for such high dimensional queries has the big disadvantage of its exponential order and thus we are interested in semi-optimal but faster approaches.

In this work we present a multiple query optimization on homogeneous distributed database application through dynamic programming for semi optimal solution.

Keywords: Distributed information retrieval (DIR), Multiple query optimization” (MQO)

1. Introduction

Distributed Systems is described as a partnership among independent cooperating centralized systems. Based on this concept the number of large scale applications has been investigated during past decades among which distributed information retrieval (DIR) systems has developed to provide a single search interface that provides access to the available databases involving resource descriptions building for each database, choosing which databases to search for particular information, and merging retrieved results into a single result list [2]-[3].

In this work we proposed a dynamic programming approach to get the solution for query optimization in distributed database.
2. Distributed Database

A distributed database (DDB) is a collection of multiple, logically interrelated databases distributed over a computer network. This resource distribution improves performance, reliability, availability and modularity that are inherent in distributed systems. As with traditional centralized databases, distributed database systems (DDBS) must provide an efficient user interface that hides all of the underlying data distribution details of the DDB from the users.

The use of a relational query allows the user to specify a description of the data that is required without having to know where the data is physically located [4]. Data retrieval from different sites in a DDB is known as distributed query processing (DQP).

Distributed Databases supports two types of distributed databases: homogenous and heterogeneous. In a homogenous distributed database system, each database is of same type. In a heterogeneous, distributed database system, at least one of the databases is of different type. The figure 1 below shows the illustration of homogeneous and heterogeneous distributed databases.
A homogenous distributed database system is a network of two or more Databases of same type that reside on one or more systems. Figure 2. illustrates a distributed system that connects three databases: HQ (Headquarter), MFG (Manufacturing), and SAL (Sales). An application can simultaneously access or modify the data in several databases in a single distributed environment. For example, a single query from a Manufacturing client on local database MFG can retrieve joined data from the table1 on the local database and the table2 on the remote database.

For a client application, the location and platform of the databases are transparent by distribution transparency. You can also create synonyms for remote objects in the distributed system so that users can access them with the same syntax as local objects. For example, if you are connected to database MFG but want to access data on database HQ, creating a synonym on MFG for the remote T2 table enables you to issue this query:

```
SELECT * FROM T2;
```
In this way, a distributed system gives the appearance of native data access. Users on MFG do not have to know that the data they access resides on remote databases.

3. Query Processing

The path that a query traverses through a DBMS until its answer is generated is shown in Figure 3. The system modules through which it moves have the following functionality:

- The Query Parser checks the validity of the query and then translates it into an internal form, usually a relational calculus expression or something equivalent.
- The Query Optimizer examines all algebraic expressions that are equivalent to the given query and chooses the one that is estimated to be the cheapest.
- The Code Generator or the Interpreter transforms the access plan generated by the optimizer into calls to the query processor.
- The Query Processor actually executes the query.

Queries are posed to a DBMS by interactive users or by programs written in general-purpose Programming languages (e.g., C/C++, FORTRAN, PL-1) that have queries embedded in them. An Interactive (ad hoc) query goes through the entire path [8].

![Query Flow Diagram]

Figure 3: Query flow through a DBMS.
4. Query Optimization in Distributed Database

Global query management provides the ability to combine data from different local databases in a single retrieval operation. The necessity for global query management arises in an open, heterogeneous multidatabase system, since autonomy and heterogeneity of component databases have given rise to a number of new major issues regarding the global query optimization strategy and context mediation including data conversion and query translation. For instance, some local DBMSs never support semi join operator which has been proposed in order that data transmission between sites could be reduced. In this regard, the global query optimization strategies developed for homogeneous distributed database systems make extensive use of semi joins which are not attractive in the multidatabase context since this may increase the local processing time. Moreover, these do not consider the cost incurred as a result of data conversion and query translation.[5]. A major cost in executing queries in a distributed database system is the data transfer cost incurred in transferring relations (fragments) accessed by a query from different sites to the site where the query is initiated. The objective of a data allocation algorithm is to determine an assignment of fragments at different sites so as to minimize the total data transfer cost incurred in executing a set of queries. This is equivalent to minimizing the average query execution time, which is of primary importance in a wide class of distributed conventional as well as multimedia database systems. Our basic principle to get a high performance is that we decompose a global query to the finest level of subqueries in order to explore all possible execution plans.

5. Components and Problems of Distributed Query Optimization

There are three components of distributed query optimization [10][11]:

5.1 Access Method: In most RDBMS products, tables can be accessed in one of two ways: by completely scanning the entire table or by using an index. The best access method to use will always depend upon the circumstances. For example, if 90 percent of the rows in the table are going to be accessed, you would not want to use an index. Scanning all of the rows would actually reduce I/O and overall cost. Whereas, when scanning 10 percent of the total rows, an index will usually provide more efficient access. Of course, some products provide additional access methods, such as hashing. Table scans and indexed access, however, can be found in all of the "Big Six" RDBMS products (i.e., DB2, Sybase, Oracle, Informix, Ingres, and Microsoft).

5.2 Join Criteria: If more than one table is accessed, the manner in which they are to be joined together must be determined. Usually the DBMS will provide several different methods of joining tables. For example, DB2 provides three different join methods: merge scan join, nested loop join, and hybrid join. The optimizer must consider factors such as the order in which to join the tables and the number of qualifying rows for each join when calculating an optimal access path. In a distributed environment, which site to begin with in joining the table is also a consideration.

5.3 Transmission Costs: If data from multiple sites must be joined to satisfy a single query, then the cost of transmitting the results from intermediate steps needs to be factored into the equation. At times, it may be more cost effective simply to ship entire tables across the network to enable processing...
6. Dynamic Programming Solution

Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems which are only slightly smaller and optimal substructure (described below) when applicable; the method takes far less time than naive methods.

The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), and then combine the solutions of the subproblems to reach an overall solution. Often, many of these subproblems are really the same. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations: once the solution to a given subproblem has been computed, it is stored or "memo-ized": the next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating subproblems grows exponentially as a function of the size of the input.

6.1 Elements of Dynamic Programming:

Optimal Substructure: A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems. It is often easy to show the optimal sub problem property as follows:-

1. Split problem into subproblems.
2. Sub problems must be optimal, otherwise the optimal splitting would not have been optimal.

Overlapping Subproblem: In overlapping subproblems the space of subproblems must be small in the sense that a recursive algorithm for the problem solves the same subproblems over and over. But in dynamic programming same subproblem resolved once and result used in repeated subproblem.

Memoization: Dynamic programming algorithm typically take advantage of overlapping subproblem once and then storing the solution in a table where it can be looked up when needed.

Based on the study of the previous work, we proposed a scheme to reduce the search space of Dynamic Programming based on reuse of query plans among similar subqueries. The method generates the cover set of similar subgraphs present in the query graph and allows their corresponding subqueries to share query plans among themselves in the search space. Numerous variants to this scheme have been developed for enhanced memory efficiency and one of them has been found better suited to improve the performance of Iterative Dynamic Programming.

7. Need of Cost Factors

In a centralized DBMS, query execution “cost” is a single dimensional factor measured in conceptual units. In a distributed database, costs must be dividing into multiple dimensions under the control of single logical database. One proposal for a universal cost metric is hard currency, but typically there are
other costs that are valuable to expose orthogonally, including response time, data freshness, and accuracy of computations [12].

8. Need of Cost Estimation

A centralized optimizer cannot accurately estimate the costs of operations at many autonomous sites. Z. G. Ives and A. Tomasic proposed middleware systems in [13, 14] address this problem by involving site specific wrappers in the optimization process, but they do not consider the cost of communicating with these wrappers. This cost is not significant in these systems because the wrappers typically reside in the same address space as the optimizer. But in general, the execution costs may also depend on transient system issues including inter communication cost between two sites. Therefore cost estimation process must be distributed in a manner reflective of the query processing, with cost estimates being provided by the sites that would be doing the work. However, to the best of our knowledge, complete cost estimation, which requires the optimizer to communicate with the sites merely to find the cost of an operation, has not been studied before. In such a scenario, communication may become the dominant cost in the query optimization process. The high cost of costing raises a number of new design challenges, and adds additional factors to the complexity of distributed query optimization.

9. Query Graph Model

The Query Graph Model is an example of preferred internal representation of user query, and join graph is an example of preferred canonical form. A join graph denominates a user query representation having nodes connected by edge, where each node represents relation and edge represents a join predicate. A relation denominates a database table having tuples and column. A join predicates relates columns of two relations to be joined by specifying conditions on column values. The Cardinality of a relation denominates the number of tuples embraced by the relation and the Selectivity of a join predicate denominates the expected fraction of tuples for which the join column value in the relation satisfies the predicate. Query cardinality is the product of cardinalities of every relation in the query times the product of selectivity factor of the query predicate.

A hybrid technique for joining tables that selects a join execution plan from among the well-known “nested-loop” and “sort-merge” methods. A method for optimizing query execution that relies on measuring the degree of “clustering” (sortedness) in the stored relations. The testing of each join column to calculate the degree of clustering of the column values in their storage order to estimate the number of page accesses required for a partial index scan. These estimates are then used to calculate the cost of each proposed join execution plan.

Any user query can be recast as a join graph made up of some combination of “linear” and “star” subgraphs.”Linear queries” can be describe as a series of relation nodes each connected by predicate edges to no more than two other relation nodes. “Star queries” describes as a group of relation nodes with a single central relation node connected by predicate edges to each of the other relation nodes. A join graph representing such a series of two way join as a canonical form of the query graph model(QGM). The practitioner may then either exhaustively enumerates all possible “feasible plans” for
join execution, using “dynamic programming” or may employ some “heuristically limited” search method to reduce the number of alternative plans in the search space considered by the optimizer. It is easily proven that dynamic programming never eliminates optimal plan, because all possible plans enumerated and evaluated. However any” heuristic” search method presents some non-zero probability of excluding superior execution plan without notice. There are usually many feasible plans for any given query and many practitioners use the exponential worst-case complexity argument to justify a priori search space truncation through heuristic search methods. Dynamic programming search space grows as N!, Here in, a search space denominates a set of executable query plans selected on the basis of some criteria related to primitive database operators.

Heuristic search method for query optimization is limiting the time and space complexity by truncating the enumeration of feasible query execution plans. Dynamic programming (DP) is the time honored method for optimizing join queries in relational database management systems and virtually all commercial optimizer rely on some abbreviated form of DP for this purpose. DP uses exhaustive enumeration with pruning to produce “optimal” execution plans without missing the best of these plans [7].

For a given query, the search space can be defined as the set of equivalent operator trees that can be produced using transformation rules. The example below illustrates 3 equivalent join trees, which are obtained by exploiting the associative property of binary operators. Join tree (c) which starts with a Cartesian product may have a much higher cost than other join trees [2].

```
SELECT ENAME, RESP
FROM EMP, ASG, PROJ
WHERE EMP.ENO=ASG.ENO
AND ASG.PNO=PROJ.PNO
```

![Query equivalent trees](image)

**Figure 4. Query equivalent trees**

Regarding different search spaces, there would be different shape of the join tree. In a linear tree, at least one operand of each operand node is a base relation. However, a bushy tree might have operators whose both operands are intermediate operators. In a distributed environment, bushy trees are useful in exhibiting parallelism [18].

www.ijert.org
In DP optimizer the cardinality of the join of N relations is the same regardless of join order, the “cost” of joining in different orders may vary substantially. Accordingly, for an N-way join query, there are N! Permutations of relation join orders embraced by the “search space” generated by DP [7].

10. Related Works

Three most common types of algorithms for join-ordering optimization are deterministic, Genetic and randomized algorithms [15].

Deterministic algorithm, also known as exhaustive search dynamic programming algorithm, produces optimal left-deep processing trees with the big disadvantage of having an exponential running time. This means that for queries with more than 10-15 joins, the running time and space complexity explodes [15]. Genetic and randomized algorithms [16]-[17] on the other hand do not generally produce an optimal access plan. But in exchange they are superior to dynamic programming in terms of running time. Experiments have shown that it is possible to reach very similar results with both genetic and randomized algorithms depending on the chosen parameters. Still, the genetic algorithm has in some cases proved to be slightly superior to randomized algorithms. Layers of distributed query optimization have been depicted in Figure 6.
There are number of Query Execution Plan for DDB such as: row blocking, multi-cast optimization, multi-threaded execution, joins with horizontal partitioning, Semi Joins, and Top n queries. In this paper we propose a dynamic programming algorithm for Query optimization in homogeneous distributed database systems [2].

11. COST MODEL

11.1 Cost Model in DBMS

The cost model assigns an estimated cost to any partial or complete plan in the searchspace. It also determines the estimated size of the data stream for output of every operator in the plan. It relies on:
a) A set of statistics maintained on relations and indexes, e.g., number of data pages in a relation, number of pages in an index, number of distinct values in a column

b) Formulas to estimate selectivity of predicates and to project the size of the output datastream for every operator node. For example, the size of the output of a join is estimated by taking the product of the sizes of the two relations and then applying the joint selectivity of all applicable predicates.

c) Formulas to estimate the CPU and I/O costs of query execution for every operator. These formulas take into account the statistical properties of its input data streams, existing access methods over the input data streams, and any available order on the data stream (e.g., if a data stream is ordered, then the cost of a sort-merge join on that stream may be significantly reduced). In addition, it is also checked if the output data stream will have any order.

The cost model gives a cost estimate for each operator tree. The cost refers to resource consumption, either space or time. Typically, query optimizers estimate the cost as time consumption [6].

10.2 Cost Model in Distributed DBMS

In a distributed execution environment, there are two different time consumption estimates to be considered: total time or response time. The former is the sum of the time consumed by each processor, regardless of concurrency, while the latter considers that operations may be carried out concurrently. Thus, response time is a more appropriate estimate, since it corresponds to the time the user has to wait for an answer to the query. In a distributed environment, the execution of an operator tree $S$ is split into several phases. Pipelined operations are executed in the same phase, whereas a storing indication establishes the boundary between one phase and the subsequent one. Resource contention is also another reason for splitting an operator tree into different phases. For instance, if a sequence of operations that could be concurrently executed require more memory than available (e.g., if the memory is not sufficient to store the entire hash tables for pipelined operations in the hash join algorithm), then it is split into two or more phases. An operator tree is also split into different phases if independent operations (which, in principle, could remain in the same phase) should be executed at the same host site: in this case, the operations are not concurrently executed just because the hosts are the same and, accordingly, they are scheduled at different phases [6].

An optimizer cost model includes cost functions to predict the cost of operators, and formulas to evaluate the sizes of results. Cost functions can be expressed with respect to either the total time, or the response time [18]-[19]. The total time is the sum of all times and the response time is the elapsed time from the initiation to the completion of the query. The total time ($TT$) is computed as below, where $T_{CPU}$ is the time of a CPU instruction, $T_{I/O}$ the time of a disk I/O, $T_{MSG}$ the fixed time of initiating and receiving a message, and $T_{TR}$ the time it takes to transmit a data unit from one site to another, and $T_{DELAY}$ the time of waiting for the producer to deliver the first result tuples:

$$TT = T_{CPU} * \#insts + T_{I/O} * \#io + T_{MSG} * \#msgs + T_{TR} * \#bytes + T_{DELAY} * \#insts$$

When the response time of the query is the objective function of the optimizer, parallel local processing and parallel communications must also be considered. This response time ($RT$) is calculated as below:
\[ RT = T_{CPU} \ast seq\_insts + T_{IO} \ast seq\_I/Os + T_{MSG} \ast seq\_msgs + TR \ast seq\_bytes + T_{DELAY} \ast seq\_insts \]

Most early distributed DBMSs designed for wide area networks have ignored the local processing cost and concentrate on minimizing the communication cost. Consider the following example:

\[ TT = 2 \ast T_{MSG} + 2 \ast T_{DELAY} + TR \ast (x + y) \]
\[ RT = \max \{ T_{MSG} + T_{DELAY} + TR \ast x, T_{MSG} + T_{DELAY} + TR \ast y \} \]

Where, \( x \) and \( y \) considered to be two queries processing in parallel. In parallel transferring, response time is minimized by increasing the degree of parallel execution. This does not imply that the total time is also minimized. On contrary, it can increase the total time, for example by having more parallel local processing (often includes synchronization overhead and it may increase the local processing time and comprising it will increase the total time and transmissions. Minimizing the total time implies that the utilization of the resources improves, thus increasing the system throughput. In practice, a compromise between the total and response times is desired [2]

### 12. Dynamic Programming Approach

The Dynamic Programming (DP) approach is recursively called for larger subgraph to give the solution for join graph. The Dynamic Programming (DP) process for query graph subset of \( T_L \) or fewer relations, where \( T_L \) represents the predetermined size limits that may be arbitrarily selected to limit the solution space. The join graph is divided into subgraphs not more than \( T_L \) relation nodes [7]. Through dynamic programming we will get improved query optimization technique that can be applied immediately to any database.

The following conventions are used in describing algorithm:

- \( s_c \): candidate set of plans;
- \( S[i,j] \): either empty set, representing no solution for the cost value \( j \); or set of “candidate set of plans” obtained by using plans from the set \( \{ 1, 2, \ldots, i \} \) and containing exactly one plan for each query with a total cost \( j \) (that is, if \( S[i,j] \) is not empty, its sets represent solutions with cost \( j \) for all queries from 1 up to the query of a plan \( p_i \));
- \( PS_S \): starting plan number for the query \( q_i \) (that is, plans from \( PS_S \) to \( PS_S + P_i \) belong to query \( q_i \));
- \( Cost(s_c \): candidate set of plans): summation of the costs of the tasks in the tasks set obtained from the union of the tasks of the plans (task sets) in \( s_c \).

Candidate sets are obtained by adding new plans to previously obtained candidate sets. The recurrence relation for candidate sets \( S[i,j] \) is as follows:

\[
S[i,j] = \{ s_c \cup \{ p_1 \} \mid \forall s_c \text{ such that } \text{Cost}(s_c \cup \{ p_1 \}) = j, \text{ and query number of plan } i \text{ is } q \}
\]

and, \( s_c \in S \) [plans of previous query (\( PS_{q-1} \) to \( PS_{q-1} \)), potentially useful costs \( j - C_i \) to \( j \)].

Using the inductive proof technique with an induction on cost values from 0 to \( C \) and query numbers from 0 to \( Q \), the correctness of the above recurrence relation can easily be shown. The proof is based on the following argument: if up to a certain cost value \( c \), for queries from 1 to \( q - 1 \), all plan sets including one plan for each query are known, then, these plan sets can be extended with a plan for query \( q \), producing cost values \( c, c + 1, \ldots \) such that the cost values are calculated.

The DP algorithm implementing this recurrence relation in a bottom-up manner. For the base case of the recurrence relation \( S[0,0] = \{ \{ \} \} \), represents that if there is no query, a plan containing no tasks (plan number 0) with total cost 0 is the solution. The starting indexes of the plans for each query, namely \( PS_q \)'s. The candidate sets, \( S \), are generated in a column wise manner for cost values starting from 1 and
considering plans from 1 to $P$. Since the candidates are sets of plans, and since plans may have common tasks, it is even possible to use a plan set with total cost equal to the current cost, and extend it with the current plan and still obtain the same cost. This can occur if the current plan’s tasks are common with the task set formed from the tasks of the plan set. Therefore, all the columns from “the current column minus the cost of the current plan” (in case no common task between the current plan and existing plan set’s tasks) to “the current column” must be examined [1].

**Dynamic Programming Psuedocode**

**Plan-based DP algorithm**

**Input**: Join Graph $G$ and size limit $T_L$

**Output**: $s_q$: solution set of plans that contains exactly one plan for each query

1. // initialization
2. $S[*, *] = \{\}$
3. $S[0, 0] = \{\{\}\}$
4. $PS_0 = 0; P_0 = 1$
5. for $i = 1$ to $Q$
6.   $PS_i = PS_{i-1} + P_{i-1}$
7.   // main part
8. for $j = 1$ to $C$ // cost values
9.   for $i = 1$ to $P$ // plans
10.   // query number of plan $i$
11.   $q = pq_i$
12.   for $k = PS_q - 1$ to $PS_q - 1$ // consider plans belonging to previous query only
13.   //query optimization of query number $k$
14.   While $|G_k| > 1$ do //stop when no more relations to join
15.     MinCost = $\infty$ //use minimum cost
16.     //examine all unjoined connected pairs
17.     for $x, y$ in $G_k$ connected by an edge do
18.       //try both join order
19.       Join = mincost(plan[x]×plan[y], plan[y]×plan[x])
20.       if JoinCost < MinCost then //remember minimum cost join
21.         Next Join = Join
22.         $r = \min(x, y)$
23.         $s = \max(x, y)$
24.         MinCost = JoinCost
25.     endif
26. endfor
// has that sizelimit been exceed?
if |relation[r] U relation[s]| > 𝑇_𝐿 then
    //call DP on larger subgraph
    if |relation[r] > relation[s]| then
        t = r
    else
        t = s
    endif
    // defer join and move to step 14 for subgraph for plan
    plan[t] = go to step 14(relation[t])
    relation[t] = {(relation[t])} // treat relation[t] as compound
        element
    else
        plan[r] = NextJoin  // update plan associated with r
        relation[r] = relation[r] U relation[s] // collapse s into r
        relation[s] = 0
    endif
endwhile
relation[1] // last relation for subgraph
for m = max(j – Ci , 0) to j // consider candidates for previously obtained
cost values
    if S[k,m] ≠ {} then
        for each s_c in S[k,m] // consider all the candidates in the entry
            if cost(s_c U {p_i}) = j then
                S[i, j] = S[i, j] U {s_c U {p_i}}
        } if i ≥ 𝑃_𝑆_𝑄 then return s_c
End Algorithm
On the other hand to prevent more than one plan for the same query from appearing in the candidate plan set, only the rows corresponding to the plans of previous query must be included in the search space. As a result, the set of candidates at column \( j \) and row \( i \) is determined by using the candidates at rows from \( PS_{q-1} \) to \( PS_{q-1} \) where \( PS_q \leq i < PS_{q+1} \) and columns from \( j-C_i \) to \( j \). All possible candidates obtained at each entry must be kept for further iterations. It is possible to obtain a new candidate if there is a candidate in the searched area that can be extended by the current plan. Notice that the existing candidate can be extended by a new plan if the total cost of the union of the plans is equal to the current cost value (or column number) [1].

For internal loop the procedure used produces a canonical Query Graph Model (QGM) herein denominated the “join graph G”.

- Join graph G : Query Graph Model (QGM)
- \( T_L \): Enumeration threshold, which is the input of predetermined limit for this process. Enumeration threshold \( T_L \) represents the maximum number of relations in any subgraph \( G_L \) referred to the dynamic programming (DP) optimization process and operates for DP search space used in optimizing graph G.
- Relations[\( x \)]: the base relation or relation subgraph corresponding to relation node \( x \).
- Plan[\( x \)]: the query execution plan selected by DP for node \( x \).

The process is initialized with relation[\( x \)]=\{\( x \)\} and plan[\( x \)]=ACCESS(\( x \)).

The mincost is first set as high as possible. In inner loop of internal loop first, connected node pair (relation[\( r \)], relation[\( s \)]) is tested for execution cost in both the directions. The optimal two-way join plan for two relations from the search space having two plans differing only in join order. The cost of optimal join order for connected node pair (relation[\( r \)], relation[\( s \)]) is then tested against mincost and, if the cost is not less than mincost, the procedure returns to select another connected node pair for evaluation. If the new two-way join plan has an execution cost that is less than the mincost saved from the previous optimal two-way plan, then reset some parameters to save the two-way join plan as the new “next join” (the new optimal two-way join plan) and tests for more untested connected node pairs.

If more connected node pairs await testing, then selects another such connected pair (arbitrarily) and returns to evaluate the next pair. If no untested connected pairs remain to be evaluated, then test the two-way join complexity by adding the node joinder numbers for the connected node pair (relation[\( r \)], relation[\( s \)]) found to have the lowest cost of all such pairs in graph G. This sum is compared to the enumeration threshold \( T_L \), if this sum is less than \( T_L \) then merges the connected node pair (relation[\( r \)], relation[\( s \)]) into a single node \( r \) having a new node joinder number, i.e. sum of node joinder number of connected node, and node relation[\( s \)] is eliminated from the join query graph G. The procedure returns to re-examine every one of the connected node pairs in the join query graph modified by the merger of nodes \( r \) and \( s \).

When a candidate is generated for a plan that belongs to the last query, the algorithm stops and returns that candidate as a solution. The verification of whether the obtained candidate set is a solution or not is trivial. If the plan belongs to the last query, and a candidate is found, then, that means this candidate contains exactly one plan for each query, thus it is a solution [7].

13. Conclusion

In Homogeneous Distributed Database the query optimization has been proposed with dynamic programming approach. Although deterministic dynamic programming algorithm produces optimal left-
deep processing trees, it has the big disadvantage of having an exponential running time. Genetic and randomized algorithms on the other hand do not generally produce an optimal access plan. But in exchange they are superior to dynamic programming in terms of running time. However, a dynamic programming approach gives us an efficient solution for Query optimization in homogeneous distributed database systems. We use “JOIN OPERATION” to estimating the cost in an intermediate stage of execution. If a new JOIN OPERATION estimates the lesser cost than we use that NextJoin and corresponding minimum cost. Hence this process used iteratively for multiple queries in homogeneous distributed database systems.

14. References


[8] Yannis E. Ioannidis, Query Optimization, in: University of Wisconsin Madison, WI 53706


[16] Yannis E. Ioannidis and Youngkyung Cha Kang, “Randomized Algorithms for Optimizing Large Join Queries”

