ITERATIVE DYNAMIC PROGRAMMING SOLUTION FOR MULTIPLE QUERY OPTIMIZATIONS IN HOMOGENEOUS DISTRIBUTED DATABASES

Article · December 2012

CITATION
1

READS
92

2 authors, including:

Anju Mishra
GLB-ITM Greater Noida
5 PUBLICATIONS **1** CITATION

SEE PROFILE

International Journal of Mathematics and
Computer Applications Research (IJMCAR)
ISSN 2249-6955
Vol. 2 Issue 4 Dec - 2012 33-46

ITERATIVE DYNAMIC PROGRAMMING SOLUTION FOR MULTIPLE QUERY OPTIMIZATIONS IN HOMOGENEOUS DISTRIBUTED DATABASES

[1]ANJU MISHRA & [2]ASHISH PANDEY

[1]Department of Computer Application, IEC-CET, Greater Noida, India

[2]Sapient Consulting, Gurgaon, India

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 day's distributed database applications are applied on Heterogeneous distributed database systems and developing Homogeneous distributed databases. We will apply a query optimizer for query optimizing join queries in a relational database system by iterative application of dynamic programming (DP) to select optimal subgraph join execution plans. 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. We assume that at the beginning of the optimization, all promising alternative plans have been generated and shared tasks are identified. Therefore, our algorithm finds an optimal solution to the MQO problem an optimal iterative dynamic programming method for such high dimensional queries will derive.

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

KEYWORDS: Dynamic Programming Algorithm (DP), Multiple Query Optimization (MQO)

INTRODUCTION

The "Iterative dynamic programming for query optimization" has been studied in 1997 and "Multiple Query Optimization" in 1980. Iterative dynamic programming query optimizer optimizes the join queries by selecting the optimal subgraph join execution plan. 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 [1]. The relational database query optimizer systems specifically to an iterative dynamic programming systems for successive optimization of query plans after parsing into subgraphs will be used to generate promising alternative plans, which will make use of already generated results during the query execution. This algorithm finds an optimal solution to the MQO problem to generate (not necessarily optimal) plans that will provide more shared tasks among queries.

The following terminology is used in the paper:

Definition (*Task*). A task is a database operation, which may partially (sometimes fully) solve a database query, and it has an associated cost. For convenience we are going to represent the costs with integers [1].

Definition (*Plan*). A plan is a set of tasks, which can solve a database query [1].

Definition (*MQO*). Given P plans to solve Q queries such that the first P_1 plans can solve the first query,

plans from $P_1 + 1$ to $P_1 + P_2$ can solve the second query, . . . and plans from $(\Sigma\{P_i \mid 1 \le i \le Q-1\})+1$ to P $(=\Sigma\{P_i \mid 1 \le i \le$

Q}) can solve the Qth query, determine a set of tasks, with minimum total cost, that contains all the tasks of at least one plan of each query (or alternatively determine a set of plans, containing one plan for each query, with the minimum total cost, in which shared tasks costs are considered only once during the computation) [1].

Definition (Query Graph model QGM), The Query Graph Model is an example of preferred internal representation of user query [7].

Definition (Join graph), 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. 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) [7].

Heuristic Search Method, [7] The practitioner may enumerates all possible "feasible plans" for join execution, through "heuristically limited" search method to reduce the number of alternative plans in the search space considered by the optimizer. 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. Heuristic search method for query optimization is limiting the time and space complexity by truncating the enumeration of feasible query execution plans.

Dynamic Programming Approach, The practitioner may then either exhaustively enumerates all possible "feasible plans" for join execution, using "dynamic programming" 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. Dynamic programming search space denominates a set of executable query plans selected on the basis of some criteria related to primitive database operators. 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].

STRUCTURE OF PROPOSED WORK

In this proposed work we first select the least costly plan from a small search space defined by disabling all "optional" classes of plan alternatives. This first "optimal" execution plan is then evaluated to obtain an initial estimate of query execution time, which is then evaluated to determine whether the time needed for additional query optimization in a larger search space is warranted. The global optimization problem is generally considered to be the problem of locating a global optimum in a large space having many local optimal. Each solution to a combinatorial optimization problem can be viewed as "state" in a "state space".

For query optimization , the state space is the search space wherein each state represents a complete ordered "join sequence" describing the feasible join execution plan for a query. Each state in

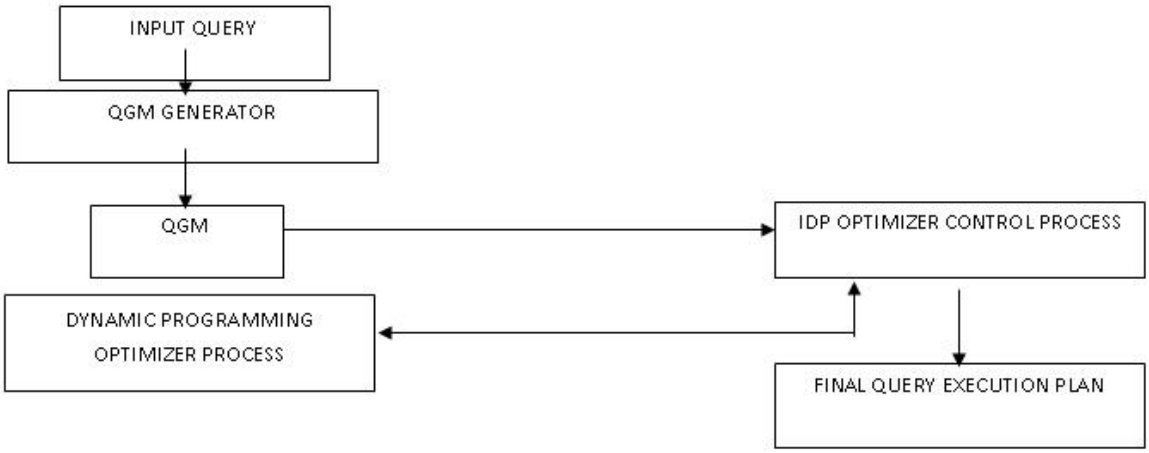

Figure 1. Procedure for Query Execution Plan

the search space has a "cost" associated with it as defined by some "cost function". A"move" is a perturbation applied to a solution ,that is, one "moves" from the state represented by a former solution to the state represented by a new solution. A "move set" is the set of moves available to proceed from one state to another, where each state in the search space is associated with a "move set". A move is chosen from the move set according to a probability distribution specified as part of the move set. Two states are said to be "adjacent" states if one move suffices to go from one state to the other. The "local minimum" in the state space is a state such that its cost is lower than that associated with all adjacent states. There may be many such local minima. A "global minimum" is a state that has the lowest cost among all local minima in the state space.The usual practical solution to the global optimization problem is to search through the space while indexing across several of the local optimal until such time as the search should be terminated [7].

A heuristic approach for optimizing cyclic(recursive) join queries that combines randomizing and local improvements with a previously known polynomial time query optimization procedure use both nested-loop and merge join methods to optimize cyclic joins. Other practitioners consider query optimization in distributed databases, employing the "semi-join" operator and observe that the class of user queries represented by "acyclic" (nonrecursive) join trees can be executed with a sequence of semi-joins. The acyclic class of join graphs is known to include all join graphs wherein no predicate edge closes an uninterrupted circuit formed by an edge plurality.

The primary objective for any query optimizer system is to ensure that the solution space of feasible plans contains the most efficient plans without making the space too large for practical generation and search. The DP optimizer produces exponentially complex solution spaces as the price for ensuring identification of the global optimum execution plan

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 3.

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 an iterative dynamic programming algorithm for Multiple Query optimization in homogeneous distributed database systems [2].

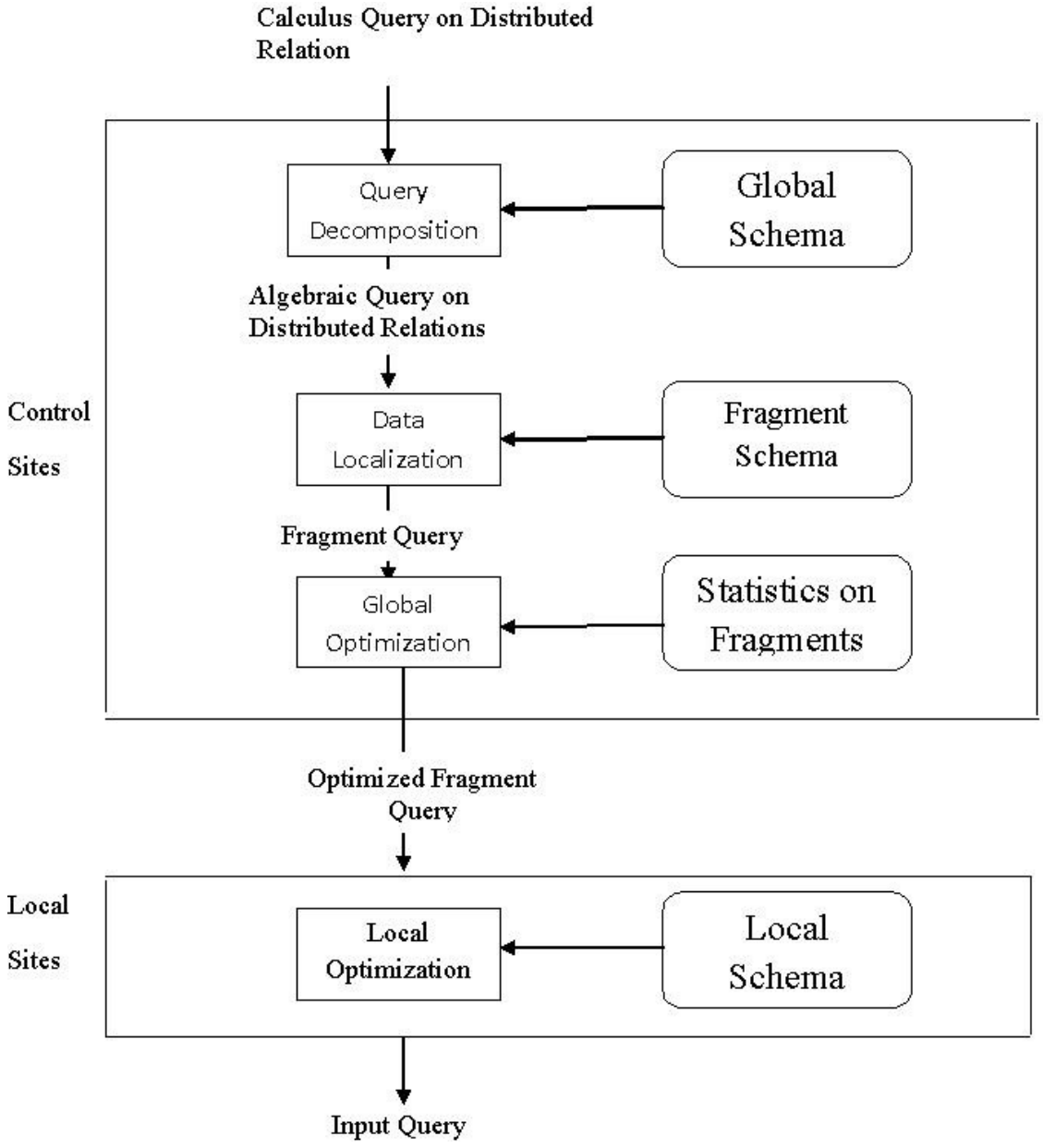

Figure 2: Distributed Query Optimization

COMPONENTS OF DISTRIBUTED QUERY OPTIMIZATION

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

1.1 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.ome products provide additional access methods, such as hashing, table scans and indexed access.

1.2 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, 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 tables is also a consideration?

1.3 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 to occur at a single site, thereby reducing overall transmission costs. This component of query optimization is an issue only in a distributed environment.

PROPOSED WORK

In this paper we need to know the properties of optimal semi-join programs for processing distributed query graphs and introduction of "execution graph" to represent semi-join programs. According to first phase we need to select exactly one plan with least cost from the joingraph of each query till the single relation is left in the query in the query set. To select the plan with least cost we can follow these steps, Figure.3. With this process we can find the least cost plan from a query and, then this process iteratively use for Multiple Queries for query optimization.

We are going to use the following convention to represent the input parameters of MQO:

• ***Q* queries:** $q_1, q_2, \ldots, q_Q$ (queries are identified as $1, 2, \ldots, Q$).

• ***T* tasks:** $t_1, t_2, \ldots, t_T$ with associated costs $\text{cost}(t_i) = c_i$ and $C = \Sigma c_i$ (tasks are identified as $1, 2, \ldots, T$).

• ***P* plans:** $P = \Sigma P_i$ and each query q_i has P_i plans, $p_{i1}, p_{i2}, \ldots, p_{iPi}$ such that each plan is a subset of tasks with costs $C_j = \Sigma\{c_k \mid t_k \text{ in } p_j\}$. For simplifying the descriptions of the algorithms, plans are also identified as $1, 2, \ldots, P_1, P_1 + 1, \ldots, P_1 + P_2, \ldots, P$, and alternatively represented as $p_1, p_2, \ldots, p_P$. In order to relate the plans with their associated queries, we also use pq_i to represent the query number of plan i .

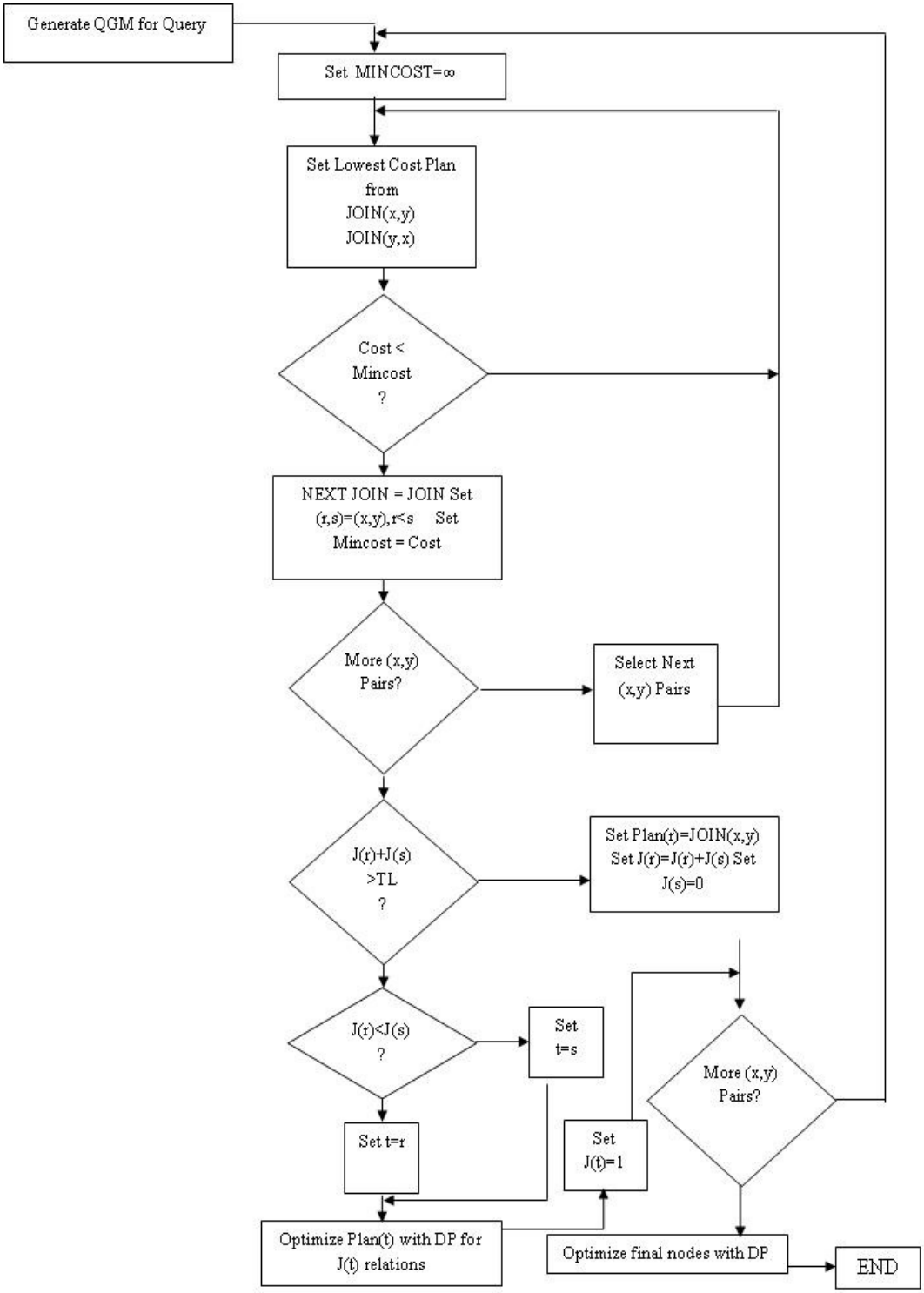

Figure 3: Selecting Least Cost Execution Plan from each Query

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 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 home site: in this case, the operations are not concurrently executed just because the homes 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

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

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 bellow:

$$RT = T_{CPU} * seq\_\#insts + T_{I/O} * seq\_\#_{I/Os} + T_{MSG} * seq\_\#msgs + T_{TR} * seq\_\#_{bytes} + T_{DELAY} * seq\_\#_{insts}$$

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:

$$TT = 2 * T_{MSG} + 2 * T_{DELAY} + T_{TR} * (x + y)$$

$$RT = max\ \{T_{MSG} + T_{DELAY} + T_{TR} * x,\ T_{MSG} + T_{DELAY} + T_{TR} * y\ \}$$

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]

Database Statistics

The main factor affecting the performance is the size of the intermediate relations that are produced during the execution. When a subsequent operation is located at a different site, the intermediate relation must be transmitted over the network. It is of prime interest to estimate the size of the intermediate results in order to minimize the size of data transfers. The estimation is based on statistical information about the base relations and formulas to predict the cardinalities of the results of the relational operations. Let R and S are relations stored at different sites [2].

Cartesian Product: The cardinality of the Cartesian product of R and S is

$$card(R \times S) = card(R) \times card(S)$$

Join: There is no general way to estimate the cardinality of a join without additional information. The upper bound of the join cardinality is the cardinality of the Cartesian product. Some systems, [12], use this upper bound, which is quite pessimistic. R* [13] uses this upper bound divided by a constant to reflect the fact that the join result is smaller than the Cartesian product. However, there is a case, which occurs frequently, where the estimation is simple. If relation R is equi-join with S over attribute A from R, and B from S, where A is a key of relation R, and B is a foreign key of relation S, the cardinality of the result can be approximated as $card(R \bowtie_{A=B} S) = card(S)$

In other words, the Cartesian product $R \times S$ contains $n_r * n_s$ tuples; each tuple occupies $s_r + s_s$ bytes. If $R \cap S = \Phi$, then $R \bowtie S$ is the same as $R \times S$. If $R \cap S$ is a key for R, then a tuple of s will join with at most one tuple from R, therefore, the number of tuples in $R \bowtie S$ is no greater than the number of tuples in S. If $R \cap S$ *in* S is a foreign key in S referencing R, then the number of tuples in $R \bowtie S$ is exactly the same as the number of tuples in s. The case for $R \cap S$ being a foreign key referencing S is symmetric [2].

As discussed earlier, ordering joins is an important aspect of centralized query optimization. This matter in a distributed context is even more important since joins between fragments may increase the communication time. Two main approaches exist to order joins in fragment queries:

1) Direct optimization of the ordering of joins.

2) Replacement of joins by combination of *semi-joins* in order to minimize communication costs.

Let R and S are relations stored at different sites and query $R \bowtie S$ be the join operator. The obvious choice is to send the smaller relation to the site of the larger one.

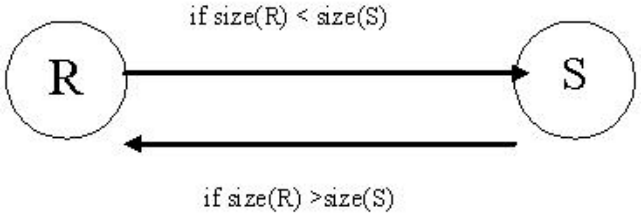

Figure 4: Join of Two Relations

More interesting is the case where there are more than two relations to join. The objective of the join ordering algorithm is to transmit smaller operands. Since the join operations may reduce or increase the size of intermediate results, estimating the size of joint results is mandatory, but difficult. Consider the following query expressed in relation.

$R \bowtie_m S \bowtie_n T$ whose join graph is below:

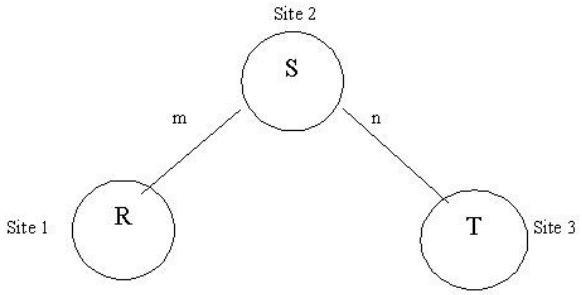

Figure 5: Join Graph

This query can be executed in at least 5 different ways.

1. R→ S; R'=R⋈S→T; R'⋈T
2. S→R; R'=R⋈S→T; R'⋈T
3. S→T; S'=S⋈T→R; S'⋈R
4. T→S; T'=T⋈S→R; T'⋈R
5. R→S; T→S; R⋈T⋈S

To select one of these programs, the following sizes must be known or predicted: size(R), size(S), size(T), size(R⋈S), size(S⋈T). Furthermore, if it is the response time that is being considered, the optimization must take into account the fact that transfers can be done in parallel with strategy 5. An alternative to enumerating all the solutions is to use heuristics that consider only the size of the operand relations by assuming, for example, that the cardinality of the resulting join is the product of cardinalities. In this case, relations are ordered by increasing sizes and the order of execution is given by this ordering and the join graph For instance, the order (R, S, T) could use strategy 1, while the order (T, S, R) could use strategy 4 [2].

ITERATIVE DYNAMIC PROGRAMMING APPROACH

Memoization, which is a technique in dynamic programming, works in the following way. We start with a recursive function and add a table that maps the function's parameter values to the results computed by the function. Then if this function is ever called twice with the same parameters, we simply look up the answer in the table. Iterative dynamic programming (DP) goes the other way. We start with the table and instead of having a recursive function; we simply fill in the table directly. At the end, we have the answer to our original problem sitting in some cell in the table.

We propose 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.

The Iterative 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, . . ., 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_i: starting plan number for the query q_i(that is, plans from PS_i to $PS_i + P_i$ belongs 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:

$S[i, j] = \cup \{s_c \cup \{p_i\} \mid \forall sc$ such that cost $(s_c \cup \{p_i\})$

$= j$, and query number of plan i 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, $c + 1$, . . . 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].

ITERATIVE DYNAMIC PROGRAMMING PSUEDOCODE

Plan-Based IDP MQO Algorithm

Input: Join Graph G and size limit T_L
Output: S_c: Solution set of plans that contain exactly one plan for each query

```
1.  // Initialization
2.  S[*, *]= { }
3.  S[0, 0] = {{ }}
4.  PS0= 0; P0= 1
5.  for i= 1 to Q
6.          PSi= PSi−1+Pi−1
7.          // main part
8.  for j = 1 to C // cost values
9.  {
10.         for I = 1 to P // plans
11.         {
12.          q = pqi   //query number of plan i
13.                 for k = PSq−1 to PSq− 1   // consider plans belonging to previous query only
14.                  //query optimization of query number k
15.                  {
16.                             While | Gk | >1 do   // stop when all relations of graph covered
17.                             {
18.                             MinCost = ∞         //use minimum cost
19.                                // examined all unjoined connected pairs
20.                                 for x,y in Gk connected by an edge do
21.                                         {
22.                                         //try both join order
23.                                           Join=MinCost(plan[x]Xplan[y],plan[y]Xplan[x])
24.                                            if JoinCost<MinCost then //remember minimum cost join
25.                                            {
```

```
                                                    NextJoin = Join
                                                r=min(x,y)
                                                 s=max(x,y)
                                                 MinCost=JoinCost
                                            }//endif
                                        }//endfor
                                    //has that size limit 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

                                         plan[t] = go to step 15(relation[t])
                                         //defer join and move to step 15 for subgraph for plan
                                         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]   //go to step 15 for last relation of subgraph
                                for m = max(j −Ci , 0) to j
                                //consider candidates for previously obtainedcost 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 ∪ {p_i}) = j then
                                                        {
                                                                S[i, j] = S[i, j] ∪ {s_c ∪ {p_i}
                                                                if i≥PS_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} toPS_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".

1. Join graph G : Query Graph Model (QGM)
2. 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.
3. Relations[x]: the base relation or relation subgraph corresponding to relation node x.
4. 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].

CONCLUSIONS

Multiple Query Optimization is much more than optimizing the single query. The infrastructure for multiple query optimizations is significant. In homogeneous distributed database the multiple query optimizations has boon proposed with Iterative dynamic programming approach. Designing efficient multiple query optimization system for homogeneous distributed database is hard, developing a robust cost metric is elusive, and building extensible enumeration architecture is a significant undertaking. However, an iterative dynamic programming approach gives us efficient solution for multiple query optimizations in homogeneous distributed database system. 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 system.

REFERENCES

1. I.H. Toroslu, A. Cosar, Dynamic programming solution for multiple query optimization problem, in: Information

Processing Letters 92 (2004) 149–155

2. Reza Ghaemi, Amin Milani Fard, Hamid Tabatabaee, and Mahdi Sadeghizadeh, Evolutionary Query Optimization for Heterogeneous Distributed Database Systems, in: World Academy of Science, Engineering and Technology 43 2008
3. J. Callan, "Distributed information retrieval". In Advances in Information Retrieval, W. B. Croft, Ed. Kluwer Academic Publishers, 2000, pp. 127–150.
4. Li, Victor O. K. "Query processing in distributed data bases", MIT. Lab. for Information and Decision Systems Series/Report no.: LIDS-P ; 1107, 1981
5. Sukhoon Kang*, Songchun Moon, Global query management in heterogeneous distributed database systems, in: Volume 38, Issues 1–5, Pages 1-861 (September 1993) Proceedings Euromicro 93 Open System Design: Hardware, Software and Applications Barcelona 6–9 September 1993
6. Dilşat ABDULLAH, Query Optimization in Distributed Databases 1302108, in: Middle East Technical University December 2003
7. Eugene Jon Shekita, Honesty Cheng Young, Iterative dynamic programming system for query optimization with bounded complexity, in: United States Patent: 5,671,403 September 23,1997
8. Yannis E. Ioannidis, Query Optimization, in: University of Wisconsin Madison, WI 53706
9. QIANG ZHU, PER-AKE LARSON,Solving Local Cost Estimation Problem for Global
10. Query Optimization in Multidatabase Systems, in: 1998 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.
11. B.M. Monjurul Alom, Frans Henskens and Michael Hannaford, Query Processing and Optimization in Distributed Database Systems, in: IJCSNS International Journal of Computer Science and Network Security, VOL.9 No.9, September 2009
12. C. S. Mullins, "Distributed Query Optimization," 1996.
13. Deepak Sukheja , Umesh Kumar Singh, A Novel Approach of Query Optimization for Distributed Database Systems, in: IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 4, No 1, July 2011 ISSN (Online): 1694-0814
14. Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S.Weld. An adaptive query execution system for data integration.In *SIGMOD*, 1999.
15. A. Tomasic, R. Amouroux, P. Bonnet, O. Kapitskaia, H. Naacke, and L. Raschid. The distributed information search component (DISCO) and the world wide web. In IEEE 1998, Volume: 10 Issue: 5 pp. 808-823.
16. Kristina Zelenay,"Query Optimization", ETH Zürich, Seminar Algorithmen für Datenbanksysteme, June 2005
17. Yannis E. Ioannidis and Youngkyung Cha Kang, "Randomized Algorithms for Optimizing Large Join Queries"
18. Michael Steinbrunn, Guido Moerkotte, Alfons Kemper, "Heuristic and Randomized Optimization for the Join Ordering Problem", The VLDB Journal - The International Journal on Very Large Data Bases, Volume 6 , Issue 3 (August 1997), Pages: 191-208, ISSN:1066-8888

19. M. Tamer Özsu, Patrick Valduriez, “Principles of Distributed Database Systems, Second Edition”, Prentice Hall, ISBN 0-13-659707-6, 1999

20. Stefano Ceri, Giuseppe Pelagatti, “Distributed Databases: Principles and Systems”, Mcgraw-Hill, ISBN-10: 0070108293, ISBN-13: 978- 0070108295, 1984

21. D. Cornell et al. ,”Integrated buffer management and query Optimization Strategy for Relational Databases” IBM Technical Disclosure Bulletin, Vol. 32, No. 12, pp. 253-257, May,1990.

22. A. Shibamiya et al. “DB2 Cost Formula”, IBM Technical Disclosure Bulletin, Vol. 34, No. 12, pp. 389-394 , May,1992.

23. Chen et al. “scheduling and processor allocation for parallel execution of Multi-Join queries”, Eighth Intl. Conf. on Data Engrg, Feb.3-7, 1992, p. 60

24. Lee et al.,”Semantic Query Reformulation in deductive Databses”, IEEE Proc. of 7th Intl. Conf. on Data Engr., Koby, Japan, 8-12 Apr. 1991, pp. 232-239.

25. U.S. Pat No. 5,067,166.