ISSN 2349-6193
Impact Factor (PIF): 2.243

IJESMR

International Journal OF Engineering Sciences & Management Research

A REVIEW OF QUERY OPTIMIZATION IN DISTRIBUTED OBJECT ORIENTED RELATIONAL DATABASE MANAGEMENT SYSTEM

Sudarshan Goswami *, Dr. Ashok Kr. Vasishtha
[*]Computer Science, Mewar University, Rajsthan, India.
Computer Science, Mewar University, Rajsthan, India.

Keywords: SQL, OQL, Distributed Query Optimization, Randomized strategies.

ABSTRACT

Execution of Structured Query Language (SQL) queries in optimized way in the distributed database is a hitch that most of the database programmer faces since the inception of database technology. Query optimization in network is one of the hardest problems in the database area. The commercialization and success of database systems is primarily due to the development of complicated query optimization techniques. Database users post their queries in a declarative mode by by means of SQL or Object Query Language (OQL) and the Query Optimizer of the related database system find a best plan to execute the same.

The optimizer determines the best indices to be used to execute a query and the order in which the operations of a query should be executed. To achieve this, the optimizer estimate alternative plans, and also estimate the cost of query plan by means of a cost model, and then selects the plan with lowest cost. There has been much research into this field. In this paper, we will review the difficulty of distributed query optimization; and will emphasis on the various components of the query optimizer required in distributed environment, i.e. cost model, search space and search strategy. A review of the existing work in this field is shown and future work is highlighted based on recent work that utilizes mobile agent technologies.

INTRODUCTION

The future of large database systems lie into the territory of distributed computing. The main reason for this is that distributed computing can be constructed at a little cost without the need for any specialized technology, by means of existing sequential computer and comparatively cheap computer networks. A database is physically distributed across different sites by fragmenting and replicating the data.

Fragmentation subdivides each relation into horizontal fragments by means of select operation or vertical fragments by means of project operation. Fragmentation is desirable since it enables the placement of data in close proximity to its place of use. Every fragment may also be replicated to a number of sites. This is preferable when the same data is accessed from applications that run at a number of sites.

The great business success of database systems is partly due to the development of sophisticated query optimization technology, where users post queries in a declarative way by means of SQL or OQL and the optimizer of the database system finds best plan to execute these queries. The optimizer determines selects indices to be used to execute a query and in which sequence the operations of a should be executed e.g. Join Query. At the end, the optimizer enumerates alternative plans, estimates the cost of every plan by means of a cost model, and chooses the plan with lowest cost [9].

Selecting the best possible execution scheme for a query is NP-hard in the number of relations [5]. For complex queries with lots of relations, this incur a prohibitive optimization cost. Thus, the key objective of the optimizer is to discover a strategy near to optimal and to stay away from bad strategies. The selection of the optimal strategy generally requires the prediction of execution cost of the alternative candidate ordering prior to actually running the query. The execution cost is spoken as a weighted combination of communication costs. I/O and CPU.

In this research paper, we will observe few ways in which queries may be optimized for distributed environments.

Later, the problem of query processing and optimization is discussed. Different kinds of search spaces, the different types of search strategies showing the advantages and disadvantages of each strategy, cost models and research concludes with the results and highlights some future directions.

ISSN 2349-6193
Impact Factor (PIF): 2.243

IJESMR
International Journal OF Engineering Sciences & Management Research

QUERY PROCESSING AND OPTIMIZATION

Query processing is the procedure of translating a query expressed in a high-level language such as SQL into low-level data manipulation operations. Query Optimization refers to the process by finding the best execution strategy for a given SQL query from a set of alternatives. Normally query processing involves many steps. The initial step is query decomposition in which an SQL query is first scanned, parsed and validated. The scanner identifies the language tokens such as SQL keywords, attribute names, and table names in the text of the query, whereas the parser checks the query syntax to find whether it is formulated accordingly or not.

The query need to be validated, by checking that all attribute and names are valid and semantically meaningful in the schema of the picked database being queried. An internal representation of the query is then created. A query articulated in relational algebra is usually called initial algebraic query and can be represented as a tree data structure called query tree. It represents the input relations of the query as leaf nodes of the tree, and represents the relational algebra operations as internal nodes.

For a given SQL query, there is more than one possible algebraic query plans. Some of these algebraic queries are better than others. The quality of an algebraic query is defined in terms of expected performance. A Query Execution Plan (QP) is then found which is represented as a query tree and includes information regarding the access method available for each relation as well as the algorithms used in computing the relational operations in the tree. Fig.1 shows the different steps of Query Processing.

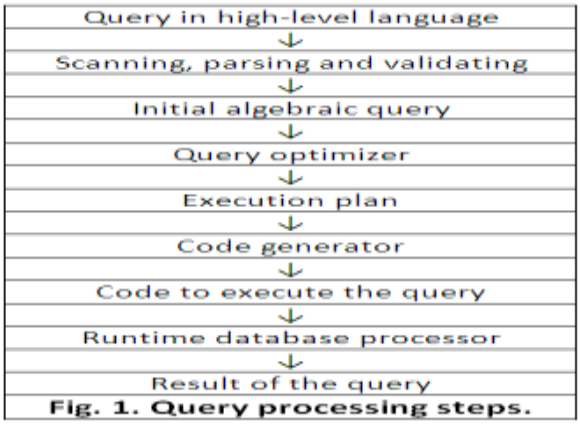


Fig. 1. Query processing steps.

Distributed Query Optimization

In DQO two more steps are involved between query decomposition and query optimization:
1. Data localization and 2. Global query optimization.
The input to data localization is the first algebraic query generated by the query decomposition. The algebraic query is specified on global relations irrespective of their fragmentation. The main role of data localization is to localize the query by means of data distributed information.

In this, the fragments that are concerned in the query are determined and the query is transformed into one that operates on fragments rather than global relations. Thus, during the data localization step, every global relation is first replaced by its localization program and then the resultant fragment query is simplified and rationalized to produce another good query. The final fragment query is generally far from optimal and this process can only eliminates bad queries. The input to the third step is a fragment query, which is an algebraic query on fragments. By permuting the ordering of operations within one fragment query, many equivalent query execution plans can be found.

The objective of query optimization is to discover an execution strategy for the query that is close to optimal. An execution strategy for a distributed query can be described with relational algebra operations and communication primitives (send/ receive operations) for transferring data between sites.

The query optimizer that follows the approach is seen as three components: A search space, a search strategy and a cost model.

ISSN 2349-6193
Impact Factor (PIF): 2.243

IJESMR
International Journal OF Engineering Sciences & Management Research

The search space is the set of alternative execution to represent the input query. These strategies are equivalent, in the sense that they yield the same result but they differ on the execution order of operations and the way these operations are implemented. The search strategy explores the search space and selects the best plan. It defines which plans are examined and in which order. The cost model predicts the cost of a given execution plan which may consist of the following components [5]:

1. **Secondary storage cost:** This is the cost of searching for reading and writing data blocks on secondary storage.
2. **Memory storage cost:** This is the cost pertaining to the number of memory buffers needed during query execution.
3. **Computation cost:** This the cost of performing in memory operations on the data buffers during query optimization.
4. **Communication cost:** This is the cost of shipping the query and its results from the database site to the site or terminal where the query originated.

SEARCH SPACE

The optimizer considers each algebraic operation independently since the query decomposition step has already taken global reorganization decisions. Optimization of all operations but the n-way Select Project Join (SPJ) is quit straight forward. It consists of choosing the algorithm and the home of operation. The crucial issue in terms of the search strategy is the join ordering problem, which is NP-hard on the number of relations [4]. The search space, or solution space, is the set of all QPs that compute the same result. A point in the solution space is one particular plan, i. e. solution for the problem. A solution is described by the query tree for executing the join expression. Every point of the search space has a cost associated with it; a cost function maps query trees to their respective costs. The query tree itself is a binary tree that consists of base relations as its leaves and joins operations as its inner nodes; edges denote the flow of data that takes place from the leaves of the tree to the root.

As the Join operation is commutative and associative, the number of possible query trees increases quickly with increasing the number of relations involved in the query [17]. Thus, we reduce the goal of the query optimizer to find the best join order, together with the best join method for each join [6]. For a complex query, involving many relations and many operators, the number of equivalent operator trees can be very high. Investigating a large search space may make optimization time prohibitive, sometime much more expensive than the execution time. Therefore, query optimizers typically restrict the size of the search space they consider. The first restriction is to use heuristics.

Another important restriction is the shape of the join tree. The following sections discuss the characteristics of the most interesting kinds of search spaces depending on the shape of the join tree they include.

Left-Deep And Right-Deep Trees

The search space restricted to only left-deep trees consists of all query trees where the inner relation of each join is a base relation as in Fig.2-a. For a fixed number of base relations, left-deep tree does not leave any degree of freedom concerning the shape of the tree, but there are n! Ways to allocate n base relations to the tree leaves.

In most sequential optimizers like the optimizer of IBM System R [15], the search space is restricted to left-deep trees. The reason for this heuristic restriction is that the space including all trees is much larger, while an optimal or nearly optimal plan can often be found in the much smaller space of left-deep trees. When join algorithms are not symmetric, which is the case for hash-join, it is useful to distinguish between left-deep and right-deep trees, see Fig.2-b.Hash-based join algorithm have been shown to be the most efficient for equi-joins [4] and particularly interesting in a parallel environment because they naturally induce intra-operation parallelism. Hash-join usually consists of two consecutive phases: Build and probe. In the build phase, a hash table is constructed on the join attribute of the smallest relation. In the probe phase, the largest relation is sequentially scanned and the hash table is consulted for each of its tuples.

ISSN 2349-6193
Impact Factor (PIF): 2.243

IJESMR
International Journal OF Engineering Sciences & Management Research

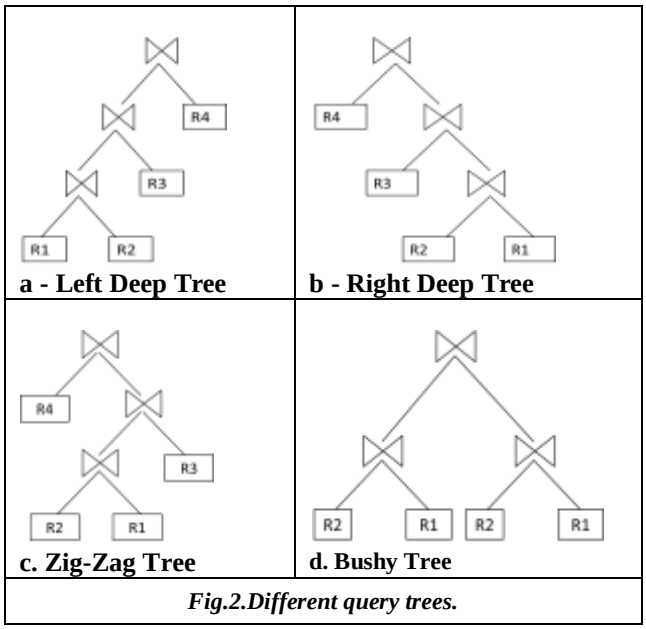


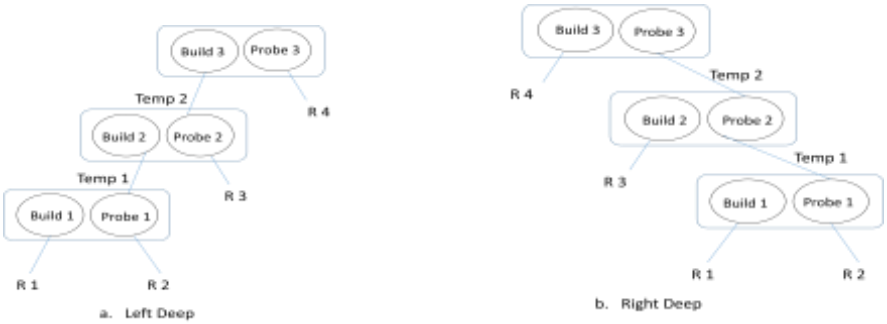
Fig.2.Different query trees.

Consider the example presented in [13, 14] as shown in Fig.3. In the left-deep tree of Fig.3-a, the temporary relationTemp1 must be completely produced and the hash table in Build2 must be finished before Probe2 can start consuming R3. The same is true for Temp2, Build3 and Probe3. Thus this tree is executed in four consecutive phases: Build R1's hash table, then probe it with R2 and build Temp1's hash table, then probe it with R3 and build Temp2's hash table, and finally probe it with R4 and produce the result.

In the right-deep tree of Fig.3-b, the arrows indicate that the temporary results are pipeline to the next operation. This tree can be executed in two phases if enough memory is available to build the hash tables: Build the hash tables for R1, R3, and R4, and then execute probe1, probe2 and probe3 in pipeline.

Zigzag Trees

Zigzag trees are mainly of interest in distributed and parallel database. Ziane, Zait, and Borle - Salanet proposed zigzag trees as an intermediate formats between left-deep trees and right-deep trees, see Figure2-c. Zigzag trees can le ad to better response time than right-deep trees in case of limited memory, especially when temporary relations are not staged to disk. The rationale is to "turn right" instead of simply slicing the right-deep trees. Turning right means that the temporary relation produced by the right-deep sub tree will be used as a building relation in the following hash-join operation, it assumed that all joins in the tree are hash join, instead of a probing relation if the right-deep had simply been used. Note that choosing a temporary relation as a building one is particularly useful when the build phase can be avoided, that is when it is already hashed on the attribute of the next join. A reasonable heuristic to favour right-deep trees or zigzag trees are when the relations are partially fragmented on disjoint homes and intermediate relations are rather larger. Consider the sample query of Fig.3-c, less memory than the right-deep tree of Fig.3-b is needed for storing the hash tables. This tree is executed in three phases: Build the hash tables of R1 and R4, execute probe1 and build2, and then execute probe2and probe3 is in pipeline.



ISSN 2349-6193
Impact Factor (PIF): 2.243

IJESMR
International Journal OF Engineering Sciences & Management Research

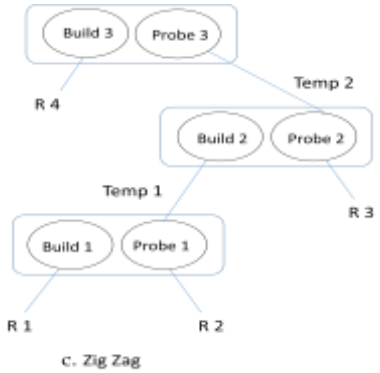


Fig. 3. Scheduling hash-join with different query trees

Bushy Trees

This search space permits join nodes where both operands are composites, i.e. no base relations as inFig2-d. Thus, the solutions in this space are not restricted. Consequently, this search space includes left-deep as well as other special tree shapes as subsets. Since the shape of possible query trees can be arbitrary, the cardinality (no. of solutions) of this space is much higher than the cardinality of the left-deep space. Bushy trees offer the best opportunity for exploiting independent parallelism and it is the most suitable for parallel machines. Independent parallelism is useful when the relations are partitioned on disjoint homes. The set of nodes where a relation is stored is called its home [3, 10]. Suppose the sample query in Fig.2-d is executed on shared-nothing machine, and that the relations (R1 and R2, R3 and R4) are partitioned on disjoint homes (respectively h1 and h2).Then R1? R2 and R3 R4 could be independently? Executed in parallel by the set of nodes that constitutesh1 and h2.

SEARCH STRATEGIES

One central component of a query optimizer is its search strategy or enumeration algorithm. The enumeration algorithm of the optimizer determines which plans to enumerate, and classically is based on dynamic programming. There are basically two approaches to solve this problem. The first approach is the deterministic strategies that are preceded by building plans, starting form base relations, joining one more relation at each step till the complete plans are obtained. When constructing QPs through dynamic programming, equivalent partial plans are constructed and compared on some cost model. To reduce the optimization cost, partial plans that are not likely to lead to the optimal plan are pruned (discarded) as soon as possible, and the cheaper partial plans are retained and used to construct the full plan. A greedy strategy builds only one such plan by means of depth-first search, while dynamic programming builds all possible plans breadth-first. The other approach is the randomized strategies that concentrate on searching the optimal solution around some particular points. They do not guarantee that the optimal plan is obtained, but avoid the high cost of optimization, in terms of memory and time consumption [13]. First, a start plan is built by a greedy strategy. Then, the algorithm tries to improve the start plan by visiting its neighbours. A neighbour is obtained by applying a random transformation to a plan. An example of a typical transformation consists in changing two randomly chosen operand relations of the plan. Iterative improvement and simulated annealing differ on the criteria for replacing the current plan by the transformed one and on the stopping criteria.

Deterministic Strategies

Dynamic Programming

This algorithm is pioneered in IBM's SystemR project [16] and it is used in almost all commercial database products [18]. The basic dynamic programming for query optimization as presented in [8] is shown in Fig. 4. It works in bottom-up way by building more complex sub-plans from simpler sub-plans until the complete plan is constructed. In the first phase, the algorithm builds access plan for every table in the query (Lines 1 to 4 of Fig.4). Typically, there are several different access plans for a relation (table). If relation A, for instance, is replicated at sites S1 and S2, the algorithm would enumerate table-scan (A, S1) and table-scan (A, S2) as alternative access plans for table A. In the second phase, the algorithm enumerates all two-way join plans by means of the access plans as building blocks (Lines 5 to 13 of Fig.4). Again, the algorithm would enumerate

ISSN 2349-6193
Impact Factor (PIF): 2.243

IJESMR
International Journal OF Engineering Sciences & Management Research

alternative join plans for all relevant sites; i.e. consider carrying out joins with A at S1 and S2. Next, the algorithm builds three-way join plans, by means of access-plans and two-way joinPlans as building blocks. The algorithm continues in this way until it has enumerated all n-way joinPlans. In the third phase, the n-way join plans are massaged by the finalizePlans function so that they become complete plans for the query; e. g. project, sort or group-by operators are attached, if necessary (Line 14 of Figure4). Note that dynamic programming uses in every step of the second phase the same join Plans function to produce more and more complex plans by means of the simpler plans as building blocks. Just as there are usually several access plans, there are usually several different ways to join two tables (e. g. nested loop join, hash join, sort-merge join, etc) and the joinPlans function will return a plan for every alternative join method.

The beauty of the dynamic programming is that inferior plans are discarded as early as possible. This approach is called pruning and is carried out by the prunePlan function. A plan can be pruned if an alternative plan exists thatdoes the same or more work at lower cost. While enumerating 2-way join plans, for example, dynamic programming would enumerate A? B and B? A as two alternative plans to execute this join, but only the cheaper of the two plans would be kept in the optPlan({A, B}) structure after pruning, and will be used as building block for 3-way, 4-way, … join plans involving A and B. Pruning significantly reduces the complexity of query optimization.

It should be noted that there are situations in which two plans, join plans of A and B, are incomparable and must both be retained in optPlans({A, B}) structure, even though one plan is more expensive than the other plan. For example, A sort-merge-join B and A hash-join B are incomparable if the sort-merge-join is more expensive than the hash-join, when the ordering of the result by the sort-merge-join is interesting. In this case, the ordering of the result might help to reduce the cost of later operations.

All final plans are comparable so that only one plan will be retained as the final plan. In a distributed DBMS, neither table-scan (A, S1)nor table-scan (A, S2) may be immediately pruned in order to guarantee that the optimizer finds a good plan. Both plans do the same work, but they produce their result at different sites. Even if table-scan (A, S1) is cheaper than table-scan (A, S2), it must be kept because it might be a building block of the overall optimal plan if, for instance, the query results are to be presented at S2. Only if the cost of table-scan (A, S1)plus the cost of shipping A from S1 to S2 is lower than the cost of table-scan (A, S2), table-scan (A, S2) is pruned.

```
Input: SPJ query q on relations R1, R2, …, Rn
Output: A query plan for q
1 for I = 1 to n do
2       optPlan ({Ri})=accessPlans (Ri)
3       prunePlans (optPlan ({Ri}))
4       End for
5       for I = 2 to n do
6       for all S ⊂{ R1, R2, …, Rn} such that |S| = I do
7               optPlan (S) = null
8                       for all O ⊂ S do
9       optPlan (S) = opPlan (S) ∪joinPlans
                (optPlan (O), optPlan (S-O))
10              prunePlans (optPlan (S))
11                      End for
12              End for
13      End for
14      finalizePlans (optPlan ({R1, R2, …, Rn}))
15      prunePlans (optPlan ({R1, R2, …, Rn}))
16      return optPlan ({R1, R2, …, Rn})
```

Fig.4.Dynamic programming algorithm.

Lanzelotte et al. [13] addressed an important issue, that is which plans are equivalent in order to prune the expensive ones? At first glance, equivalent partial plans are those that produce the same result (tuples). In fact, the order of resulting tuples is important equivalence criterion. The reason is that in the presence of sort-merge join; a partial with a high cost could lead to a better plan, if a sort operation could be avoided. The researchers

ISSN 2349-6193
Impact Factor (PIF): 2.243

IJESMR
International Journal OF Engineering Sciences & Management Research

made some experiments showing that dynamic programming performs better than a randomize strategy for queries with small number of relations, but this situation is inverted when the query has 7 relations or more.

Greedy Algorithm

As an alternative to dynamic programming, greedy algorithms have been proposed. These greedy algorithms run much faster than dynamic programming, but they typically produce worse plans.

Fig.5 shows the basic greedy algorithm for query optimization. Just like dynamic programming, this greedy algorithm has three phases and constructs plans in a bottom-up way. It makes use of the same accessPlans, joinPlans, and finalizePlans functions in order to generate plans. However, in the second phase this greedy algorithm carries out a very simple and rigorous selection of the join order. With every iteration of the greedy loop (Lines 5 to 11 of Fig.5), this algorithm applies a plan evaluation function, in order to select the next best join. As an example, for a five-way join query with tables A, B, C, D, and E, the plan evaluation function could determine that A and D should be joined first; the result of A?D should be joined with C next; B and E should be joined next; finally, the results of C ? (A ? D) and B?E should be joined. Obviously, the quality of the plans produced by this algorithm strongly depends on the plan evaluation function [9].

Input: SPJ query q on relations R1, R2, …, Rn
Output: A query plan for q
1 for I = 1 to n do
2 optPlan ({Ri}) = accessPlans (Ri)
3 prunePlans (optPlan ({Ri}))
4 End for
5 toDo = { R1, R2, …, Rn}
6 for I = 1 to n-1 do
7 find O, I ∈toDo, P ∈ joinPlans (optPlan (O), optPlan (I)) such that eval (P) = min {eval(P')| P' | joinPlans (optPlan(O'), optPlan (I')); O,I | toDo}
8 generate new symbol: T
9 optPlan ({T}) = {P}
10 toDo = toDo - {O, I} ∪{T}
11 delete (optPlan (O), delete (optPlan(I))
12 End for
13 finalizePlans (optPlan ({R1, R2, …, Rn}))
14 prunePlans (optPlan ({R1, R2, …, Rn}))
15 return optPlan ({R1, R2, …, Rn})

Fig. 5. Greedy algorithm.

ITERATIVE DYNAMIC PROGRAMMING

a new class of enumeration algorithms was presented and evaluated by Kossmann and Stocker [9], which was based on the early work in called Iterative Dynamic Programming (IDP). The main idea of IDP is to concern dynamic programming several times in the process of optimizing a query and can be seen as a combination of dynamic programming and greedy algorithm. IDP has reasonable (i.e. polynomial) complexity and produces in most situations very good plans. Researchers made some experiments to show that IDP produce better plans than any other algorithm in situations in which dynamic programming is not viable because of its high (exponential) complexity. One particular advantage is that certain IDP-variants adapt to optimization problem. If the query is simple, these IDP-variants produce an optimal plan and in the same time as dynamic programming. If the query is too complex for dynamic programming, these IDP-variants produce best plan, close to optimal, but this plan is significantly better than the plans produced by other algorithms that are applicable in those situations (e. g. randomized algorithms). IDP can be classified as a generalization of dynamic programming and the greedy algorithm with the goal to combine the advantage of both. They presented eight variants of IDP that differ in three ways: The time the iteration takes place, the size of the building blocks generated in each algorithm, and the number of building blocks produced in every iteration.

Randomized Strategies

Randomized Strategies typically perform random walks in the solution space via sequence of moves. The kinds of moves that are considered depend on the solution space. If left-deep processing trees are desired, each

ISSN 2349-6193
Impact Factor (PIF): 2.243

IJESMR
International Journal OF Engineering Sciences & Management Research

solution can be represented uniquely by an ordered list of relation participating in the join. There are different moves for modifying these relations, Swap and 3Cycle.
Swap exchanges the position of two arbitrary relations in the list, and 3Cycle performs a cyclic rotation of three arbitrary relations in the list. For instance, if A1 A2 A3 A4 A5 was a point in the solution space, application of Swap might lead to A1 A4 A3 A2 A5, whereas 3Cycle could yield A5 A2 A1 A4 A3 [4] If bushy processing trees are considered, the following moves, introduced by Ioannidis and Kang [6] are used for traversal of the solution space:

- Commutativity: A?B ? B?A
- Associativity: (A?B)?C?A?(B?C)
- Left Join Exchange: (A?B)?C? (A?C)?B
- Right Join Exchange: (A?B)?C? B? (A?C)

The points that can be reached in one move, which form a point P are called the neighbors of P. A move is called uphill (downhill) if the cost of the source point is lower (higher) than the cost of the destination point. A point is a local minimum if in all paths starting at that state; any downhill move comes after at least one uphill move. A point is a global minimum if it has lowest cost among all points.

Iterative Improvement
The iterative improvement algorithm is shown in Fig. 6. The inner loop of the algorithm is called a local optimization that starts at random point and improves the solution by repeatedly accepting random downhill moves until it reaches local minimum. Iterative improvement repeats these local optimizations until a stopping condition (a predetermined number of starting points are processed or a time limit is exceeded) is met, at such point it returns the local minimum with lowest cost [6].

Simulated Annealing
Iterative improvement suffers from a major drawback. Because local optimization performs only downhill moves, it is possible that even with a high number of starting points the final result is still unacceptable. This is the case especially when the solution space contains a large number of high cost local minima. In this case, the algorithm gets easily trapped in one of the high cost local minima. Simulated annealing is a refinement of iterative improvement that removes this restriction. It accepts uphill moves with some probability, trying to avoid being caught in a high cost local minimum. The algorithm is shown in Fig.7. It was originally derived by analogy to process the annealing of crystals. The inner loop of the algorithm is called stage. Each stage is performed under a fixed value of a parameter T, called temperature, which controls the probability of accepting uphill moves. Each stage ends when the algorithm is considered to have reached equilibrium then the temperature is reduced according to some function and another stage begins. The algorithm stops when it is considered to be frozen, i.e. when the temperature is equal zero [6].

Two Phase Optimization
The basic idea for this variant is the combination of iterative improvement and simulated annealing in order to combine the advantage of both [6]. Iterative improvement, if applied repeatedly, is capable of covering a large part of the search space and descends rapidly into a local minimum, whereas simulated annealing is very well suited for thoroughly covering the neighbourhood of a given point in the search space. Thus, Two Phase Optimization works as follows:

1. For a number of randomly selected starting points, local minima are sought by way of iterative improvement.
2. From the lowest of these local minima, the simulated annealing algorithm is started in order to search the neighbourhood for better solutions. Because only the close proximity of the local minimum needs to be covered, the initial temperature for simulated annealing pass is set lower than it would be for the simulated annealing run by itself.

/* Function Iterative Improvement */
Output: MinState = "Optimized processing tree"
MinCost = 8
    Do
    State = "Random starting point"
    Cost = cost(State)
Do
NewState = "State after random move"

ISSN 2349-6193
Impact Factor (PIF): 2.243

IJESMR
International Journal OF Engineering Sciences & Management Research

```
NewCost = cost(NewState)
If NewCost< Cost then
        Cost = Newcost
End if
While "Local minimum not reached"
If Cost <MinCost then
MinState = State
MinCost = Cost
End if
While "Time limit not exceeded"
Return MinState
End Iterative Improvement
```

Fig.6.Iterative improvement.

```
/* Function Simulated Annealing */
Input: State "Random starting point"
Output: MinState "optimized processing tree"
MinState = State; Cost = cost(State); MinCost
        = Cost;
        T = "Starting temperature"
        Do
Do
NewState = "State after random move"
NewCost = cost(NewState)
If NewCost<= Cost then
                State = NewState
        Cost = NewCost
        Else "with probability e (NewCost-Cost)/T"
                State = NewState
        Cost = NewCost
        End if
        If Cost <MinCost then
MinStae = State
MinCost = Cost
        End if
        While "Equilibrium not reached"
While "Not frozen"
Return MinState
End Simulated Annealing
```

Fig.7. Simulated Annealing.

COST MODEL

Lanzelotte, Valduriez, and Zait [12] introduced a cost model that captures all aspects of parallelism and scheduling. They define the cost estimate of a QP containing only join nodes. All formulas given below compute response time and they simply refer to it by cost. In addition to the traditional assumptions, uniform distribution of values and independence of attributes, they also assume that tuples of a relation are uniformly partitioned among nodes of different homes, and there is no overlap between nodes of different homes, although several relations may share the same home.

In the following, R refers to a base relation of the physical schema, and N to the operation captured by QP node. P denotes, in the same time, a QP and the transient relation produced by that QP. The parameters, database schema or system parameters, used in the cost model are shown in Table 1.

An optimal execution of the join operation requires each operand to be partitioned the same way. For example, if p and q are both partitioned on n nodes by means of the same function on the join attribute, the operation join(p, q) is equivalent to the union of n parallel operations join(pi, qi), with I = 1,…, n. If the above mentioned condition is not satisfied, parallel join algorithm attempt to make such condition available by recognizing the

ISSN 2349-6193
Impact Factor (PIF): 2.243

IJESMR
International Journal OF Engineering Sciences & Management Research

relations, i. e. dynamically repartitioning the tuples of the operand relations on the nodes by means of the same function on the join attribute..

card(R)	Number of tuples in relation
width(R)	Size of one tuple of relation R
cpu	CPU speed
network	Network speed
packet	The size of packet
send	The time for a send operation
receive	The time for a receive operation

Table 1: Cost model parameters

First, estimate the cost of partitioning an operand relation *R*. Obviously, if the relation is appropriately partitioned, this cost is 0. Let *# source* be the number of nodes over which *R* is partitioned, and *# dest* be the number of nodes of the destination home. Each source node contains *card(R)/ # source* tuples. Thus it will send *card(R) * width(R)/ (n * packet)* packets. If we assume that tuples will be uniformly distributed on destination nodes, then each node will receive *card(R)/ # dest* tuples, and thus will process *card (R) * width (R)/ (m * packet)* incoming packets. Since a destination node starts processing only when first packet arrives, the cost of repartitioning *R* on *# dest* nodes is:

*cost (part(R)) = max((card(R) * width(R)/ (# source * packet)) * send, (card(R)* width(R)/ (# dest * packet)) * receive + send + packet/ network)*

The cost of joining tuples of *p* and *q*, where *p* and *q* are, respectively, the pipelined and stored operands of the *join* operation, is:

cost (join(p, q)) = max(costalg (join(p, q)),cost(part(p))) + cost(part(q))

where *costalg(join(p, q))* is the cost to process the join at one node. It depends on the join algorithm used. The partitioning of *p* is performed simultaneously to the join processing, after the repartitioning of q has completed.

Given a QP *p* scheduled in phases, each denoted by *ph*, the cost of *p* is computed as follows:

$$\text{cost (P)} = \sum_{ph \in P} \left(\max_{o \in ph} \begin{pmatrix} respTime(N) \\ +pipe\_delay(N \end{pmatrix} + store\_delay(ph)\right)$$

Where *pipe_delay* (*N*) is waiting period of node *N*, necessary for the producer to deliver the first result tuples. It is equal to 0 if the input relations of *N* are stored. *Store_delay* (*ph*) is the time necessary to store the output results of phase *ph*. It is equal to 0 if *ph* is the last phase, assuming that the results are delivered as soon as they are produced.

Based on the above model, and taking into account the aspect of parallel execution, the same researchers define the cost of the plan as three components: Total Work (*TW*), Response Time (*RT*), and Memory Consumption (MC). TW and RT are expressed in seconds, and MC in Kbytes [13].

The first two components are used to express a trade-off between response time and throughput. The third component represents the size of memory needed to execute the plan. The cost function is a combination of the first two components, and plans that need more memory than available are discarded. Given a plan p, its cost is computed by a parameterized function packet The size of packet send The time for a send operation receive The time for a receive operation $\cos_{(\text{t(WRT,WTW)}}$p, defined as follows:

$$cos_{(W_{RT},w_{TW})} = \{_{\infty}^{W_{RT}* RT+W_{TE}* TW} \quad \begin{pmatrix} if\ MC\ of\ plan\ does\ not \\ exceed\ available\ memory\ otherwise \end{pmatrix}$$

Where *WRT* and *WTW* are weight factors between 0 and 1, such that *WRT* + *WTW* = 1.

A major difficulty in evaluating the cost is in assigning values to the weight of the first two components. These factors depend on the system state (e. g. load of the system and number of the queries submitted to the system), and are ideally determined at run time. Lanzelotte et al. [13] suppose that only one query is submitted to the system at a time, therefore the cost function become:

$$\cos t(P) = \{_{\infty}^{RT\ (\ if\ MC\ of\ plan\ doen\ not\ exceed\ available\ meory\ otherwise)}$$

The response time of P, scheduled in phases (each denoted by ph), is computed as follows:

$$\text{respTime (P)} = \sum_{ph \in P} \left(\max_{o \in ph} \begin{pmatrix} respTime(O) \\ +pipe\_delay(O \end{pmatrix} + store\_delay(ph)\right)$$

ISSN 2349-6193
Impact Factor (PIF): 2.243

IJESMR
International Journal OF Engineering Sciences & Management Research

Where *O* denotes an operation and *respTime (O)* is the response time of *O*. *pipe_delay (O)* is the waiting period of *O*, necessary for the producer to deliver the first result tuples. It is equal to 0, if the input relations of *O* are stored. *Store_delay (ph)* is the time necessary to store the output results of phase *ph*. It is equal to 0, if *ph* is the last phase, assuming that the results are delivered as soon as they are produced. The cost model, in a parallel environment depends on dynamic parameters. For example, the amount of available memory may have an impact on the choice of a scheduling strategy. Insufficient memory is a reason that forces the execution plan to be split into more phases. Memory size is usually unknown to the optimizer that operates at compile time. Parallelism introduces another crucial dynamic parameter, which is the way the load is balanced among the processors. In some cases, the impact of dynamic parameters is limited. For example, in shared-memory architecture, if we do not suppose inter-operation, knowing the amount of available memory is not relevant, because only one operation is executed at a time. However, in general case, some optimization decisions should be made at run time. One solution to this problem is to build several execution plans, put together by means of choosing operators.

CONCLUSION

We have studied the problem of distributed query optimization. We focus on the major optimization issues being addressed in distributed databases. We have seen that a query optimizer is mainly consists of three components: The search space, the search strategy, and the cost model. Different kinds of search spaces are discussed with different schedules. Search strategies, the central part of the optimizer, can be seen as two classes. We have shown that all published algorithms of the first class; i.e. deterministic strategies have exponential time and space complexity and are guaranteed to find the optimal plan. Whereas, the big advantage of the algorithms of the second class, i.e. randomized algorithms, is that they have constant space overhead.
Typically, randomized algorithms are slower than heuristics and dynamic programming for simpler queries but this is inverted for large queries. Randomized strategies do not guarantee to find the optimal plan. Some cost models are discussed and the basic parameters for parallel environment are shown. Finally, a promising future direction is the use of mobile-agent technologies [11], which provide the combination of flexibility and precision, to optimize the execution of distributed queries. The proposed mobile agent system is a rule -based and should contain the necessary information, i.e. rules, used in the optimization process such as selectivity and cardinality.

REFERENCES

1. Galindo-Legaria C., Pellenkoft A., and Kersten,M., “Fast, Randomized Join-Order Selection –Why Use Transformation?,” in Proceedings of the Conference on Very Large Databases(VLDB), Santiago, Chile, pp. 85-95, 2004.
2. Ibraraki T. and Kameda T. “Optimal Nesting for Computing N-Relational Joins,” ACM Transactions on Database Systems, vol. 9, no. 3,pp. 482-502, 2004.
3. Chen M. S., Yu P. S., and Wu K. L., “Optimization of Parallel Execution for Multi-Join Queries,” IEEE Transaction on Knowledge and Data Engineering, vol. 6, no. 3, 2006.
4. DeWitt D. J. and Gray J., “Parallel Database Systems: The Future of High Performance Database Systems,” Communication of ACM, vol.35, no. 6, pp. 85-98, 2002.
5. Elmasri R. and Navathe S. B., Fundamentals of Database Systems, Reading, MA, Addison-Wesley, 2000.
6. Ioannidis Y. E. and Kang Y. C., “Randomized Algorithms for Optimizing Large Join Queries,” in Proceedings of the ACM SIGMOD Conference on Management of Data, Atlantic City, USA, pp.312-321, 2010.
7. Ioannidis Y. E., “Query Optimization,” in Trucker A. (Ed), The Computer Science and Engineering Handbook, CRC press, pp. 1038-1054, 2006.
8. Kossmann D., “The State of Art in Distributed Query Optimization,” ACM Computing Surveys, September 2009.
9. Kossmann D. and Stocker K., “Iterative Dynamic Programming: A New Class of Query Optimization Algorithm,” ACM TODS, March2010.
10. Kremer M. and Gryz J., “A Survey of Query Optimization in Parallel Databases,” Technical Report, CS-04-1999, York University, Canada,2009.
11. Lange D. B., “Mobile Objects and Mobile Agents: The Future of Distributed Computing,” in Jul E. (Ed), ECOOP’98, LNCS 1445, pp.1-12,2008.
12. Lanzelotte R. S. G., Valduriez P., and Zait M., “On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces,” in Proceedings of the Conference on Very Large Databases (VLDB), Dublin, Ireland, pp. 493-504,2003.

ISSN 2349-6193
Impact Factor (PIF): 2.243

IJESMR

International Journal OF Engineering Sciences & Management Research

13. Lanzelotte R. S. G., Valduriez P., Zait M., and Ziane M., "Industrial-Strength Parallel Query Optimization: Issues and Lessons," Information Systems, vol. 19, no. 4, pp. 311-330, 2004.
14. Niccum T. M., Srivastava J., Himatstingka B., and LI J., A Tree-Decomposition Approach to Parallel Query Optimization, Technical ReportTR 93-016, University of Minnesota, 2006.
15. Pham V. A. and Karmouch A. "Mobile Software Agents: An Overview," IEEE Communication Magazine, pp. 26-37, 2008.
16. Selinger P. G., Astrahan M. M., Chamberlin D.D., Lorie R. A., and Price T. G., "Access Path election in A Relational Database Management System," in Proceedings of the ACM SIGMOD Conference on Management of Data, Boston,USA, pp. 23-34, 2009.
17. Shekita E. and Young H., Iterative Dynamic Programming, IBM Technical Report, 2008.
18. Steinbrunn M., Moerkotte G., and Kemper A., "Heuristic and Randomized Optimization for the Join-Ordering Problem," VLDB Journal, vol. 6,no. 3, pp. 191-20, 2007..