ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)
Volume 2, Issue 2, February 2013

Query Optimization: Finding the Optimal Execution Strategy

Vinod S.Gangwani[1], Prof.P.L.Ramteke[2]

Department of Computer Science & Engineering, H.V.P.Mandal's College of Engineering & Technology, Amravati, Maharashtra, India.

***Abstract*— The query optimizer is the component of a database management system that attempts to determine the most efficient way to execute a query. The optimizer considers the possible query plans for a given input query, and attempts to determine which of those plans will be the most efficient. Cost-based query optimizers assign an estimated "cost" to each possible query plan, and choose the plan with the smallest cost. The distributed query optimization is one of the hardest problems in the database area. The great commercial success of database systems is partly due to the development of sophisticated query optimization technology where users pose queries in a declarative way using SQL or OQL and the optimizer of the database system finds a good way (i. e. plan) to execute these queries. t. There has been much research into this field. In this paper, we study the problem of distributed query optimization; we focus on the basic components of the distributed query optimizer, i. e. search space and search strategy. A survey of the available work into this field is given.**

***Index Terms* — Deterministic strategies, query optimization, randomized strategies.**

I. INTRODUCTION

The goal of query optimization is to reduce the system resources required to fulfill a query, and ultimately provide the user with the correct result set faster. Query optimization is important for at least a few reasons. First, it provides the user with faster results, which makes the application seem faster to the user. Secondly, it allows the system to service more queries in the same amount of time, because each request takes less time than unoptimized queries. Thirdly, query optimization ultimately reduces the amount of wear on the hardware (e.g. disk drives), and allows the server to run more efficiently (e.g. lower power consumption, less memory usage) The great commercial success of database systems is partly due to the development of sophisticated query optimization technology, where users pose queries in a declarative way using SQL or OQL and the optimizer of the database system finds a good way (i. e. plan) to execute these queries. The optimizer, for example, determines which indices should be used to execute a query and in which order the operations of a query (e.g. joins, selects, and projects) should be executed. To this end, the optimizer numerates alternative plans, estimates the cost of every plan using a cost model, and chooses the plan with lowest cost [3].

II. QUERY PROCESSINS AND OPTIMIZATION

Query processing includes translation of high-level queries into low-level expressions that can be used at the physical level of the file system. Query Optimization refers to the process by which the best execution strategy for a given query is found from a set of alternatives. Typically query processing involves many steps. The first step is query decomposition in which an SQL query is first scanned, parsed and validate. The scanner identifies the language tokens – such as SQL keywords, attribute names, and relation names – in the text of the query, whereas the parser checks the query syntax to determine whether it is formulated according to the syntax rules of the query language. The query must also be validated, by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried. An internal representation of the query is then created. A query expressed in relational algebra is usually called initial algebraic query and can be represented as a tree data structure called query tree. It presents 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. Some of these algebraic queries are better than others. The quality of an algebraic query is defined in terms of expected performance. Therefore, the second step is query optimization step that transforms the initial algebraic query using relational algebra transformations into other algebraic queries until the *best* one is found. A Query Execution Plan (QEP) is then founded which represented as a query tree includes information about the access method available for each relation as well as the algorithms used in computing the relational operations in the tree. The next step is to generate the code for the selected QEP; this code is then executed in either compiled or interpreted mode to produce the query result [4, 5]. Figure 1 shows the different steps of Query Processing.

Query optimization is the refining process in database administration and it helps to bring down speed of execution. Most of the databases after are built and filled with data, and used come down on speed. The time taken to execute a query and return results exponentially grows as the amount of data increases in the database leading to more waiting times on the user, and application sides. Sometimes the wait times could range from minutes, to hours, and days as well in worst cases. Technically one of the reasons of slow speeds of executions could be excess normalization leading to multiple tables. More the number of tables, more is the complex nature of joins, and thus leading to more execution times. Sometimes, the complex nature of joins could worse the situation by bring the execution into deadlock.

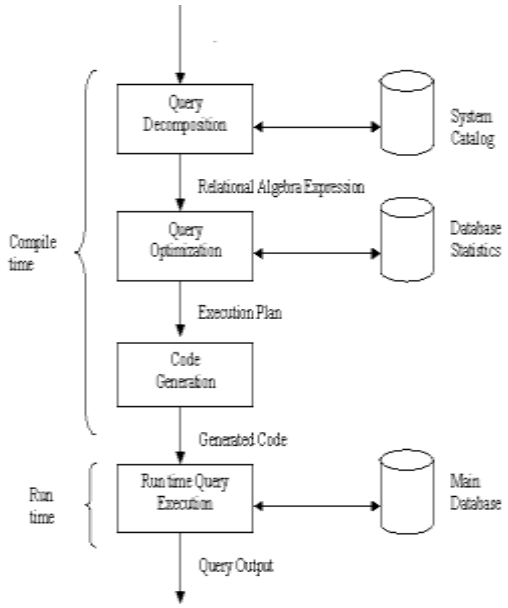

Figure1. Query Processing

III. STRATEGIES FOR SEARCHING

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 proceeded by building plans, starting form base relations, joining one more relation at each step till the complete plans are obtained. When constructing QEPs through dynamic programming, equivalent partial plans are constructed and compared on some cost model .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 [6]. First, a start plan is built by a greedy strategy. Then, the algorithm tries to improve the start plan by visiting its neighbors. A neighbor 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.

A. Deterministic Strategies

i) *Algorithm for Dynamic Programming*

The basic dynamic programming for query optimization as presented in [8] is shown in Figure 2. 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 Figure 2). 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 using the access plans as building blocks (Lines 5 to 13 of Figure 2). Next, the algorithm builds three-way join plans, using access-plans and two-way join plans as building blocks. The algorithm continues in this way until it has enumerated all n-way join plans. 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 Figure 2). Note that dynamic programming uses in every step of the second phase the same *joinPlans* function to produce more and more complex plans using 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 (Lines 3, 10 and 15 of Figure 4). This approach is called pruning and is carried out by the *prunePlan* function. A plan can be pruned if an alternative plan exists that does 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 *hashjoin* 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 R_1, R_2, ,…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 ⊂ { R_1, R_2, …, Rn} such that |S| = I do
7 optPlan (S) = ϕ
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 finalizePlan (optPlan ({R_1, R_2, …, Rn}))

15 prunePlans (optPlan ({R_1, R_2, …, Rn}))
16 return optPlan ({R_1, R_2, …, Rn})

Figure 2. Dynamic programming algorithm.

ii) 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 [8]. Figure 3 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 Figure 3), 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 = { R_1, R_2, …, 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 ({R_1, R_2, …, Rn}))
14 prunePlans (optPlan ({R_1, R_2, …, Rn}))
15 return optPlan ({R_1, R_2, …, Rn})

Figure 3. Greedy algorithm.

A. Randomized algorithm

A randomized algorithm is an algorithm which employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits. Formally, the algorithm's performance will be a random variable determined by the random bits; thus either the running time, or the output (or both) are random variables .Randomize algorithm usually perform random walks in the solution space via series of *moves*. The kinds of moves that are considered depend on the solution space. If left-deep processing trees are desired, each 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.

i) *Iterative Improvement*

Iterative design is a design methodology based on a cyclic process of prototyping, testing, analyzing, and refining a product or process. Based on the results of testing the most recent iteration of a design, changes and refinements are made. The iterative improvement algorithm is shown in Figure 4. 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 [8,10].

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

Figure 4. Iterative improvement.

ii) *Simulated Annealing*

Simulated annealing (SA) is a generic probabilistic meta heuristic for the global optimization problem of locating a good approximation to the global optimum of a given function in a large search space. It is often used when the search space is discrete (e.g., all tours that visit a given set of cities). For certain problems, simulated annealing may be more efficient than exhaustive enumeration — provided that the goal is merely to find an acceptably good solution in a fixed amount of time, rather than the best possible solution.

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 Figure 5. 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. [8,10].

```
Function Simulated Annealing
Input: State "Random starting point"
Output: MinState "optimized proceesing 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
```

Figure 5. Simulated annealing.

iii) *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 neighborhood 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 neighborhood 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 [8].

IV. CONCLUSION

Comparing the performance of the various optimization algorithms, we can draw the following conclusions. Algorithms that perform an exhaustive or near exhaustive enumeration of the solution space, such as dynamic programming, can compute the optimal result, but the high running time makes their application only feasible for queries that are not too complex. 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.

ACKNOWLEDGMENT

My thanks to the Guide, Prof. P.L.Ramteke and Principal Dr.A.B.Marathe, who provided me constructive and positive feedback during the preparation of this paper.

REFERENCES

[1] Oszu, M. T. and Valduriez P., "Distributed and Parallel Database Systems," in Trucker A. (Ed), *The Computer Science and EngineeringHandbook*, CRC press, pp. 1093-1111, 1997.
[2] Oszu, M. T. and Valduriez P., "Distributed Database Systems: Where Are We Now," *EEE Computer*, vol. 24, no. 8, pp. 68-78, 1991.
[3] Kossmann D. and Stocker K., "Iterative Dynamic Programming: A New Class of Query Optimization Al gorithm," *ACM TODS*, March 2000.
[4] Elmasri R. and Navathe S. B., *Fundamentals of Database Systems*, Reading, MA, Addison- Wesley, 2000.
[5] Ioannidis Y. E., "Query Optimization," in Trucker A. (Ed), *The Computer Science and Engineering Handbook*, CRC press, pp. 1038- 1054, 1996.
[6] 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, 1994.
[7] Kossmann D., "The State of Art in Distributed Query Optimization," *ACM Computing Surveys*, September 2000.
[8] 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, 1997.
[9] Kossmann D. and Stocker K., "Iterative Dynamic Programming: A New Class of Query Optimization Algorithm," *ACM TODS*, March 2000.
[10] 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, 1990.
[11] 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, 1994.

Vinod S.Gangwani *is working as an Assistant Professor and currently pursuing masters degree program in Computer Science and Engineering in HVPM's College Of Engineering & Technology, Amravati, India , Mobile:+919764262528*

Prabhakar L.Ramteke *is working as Associate Professor and is Head Of Information Technology Department of HVPM's College Of Engineering & Technology, Amravati, India, Mobile:+919421818808*