A Survey of Issues of Query Optimization in Parallel Databases

Article *in* International Journal of Computer Applications · December 2010

DOI: 10.5120/1560-2082 · Source: DOAJ

2 authors:

Sunita Mahajan
Mumbai Educational Trust
20 PUBLICATIONS **71** CITATIONS
SEE PROFILE

Vaishali Jadhav
Philadelphia College of Osteopathic Medicine
3 PUBLICATIONS **3** CITATIONS
SEE PROFILE

International Journal of Computer Applications (0975 – 8887)
Volume 11– No.3, December 2010

A Survey of Issues of Query Optimization in Parallel Databases

Dr. Sunita Mahajan
Principal, Institute of Computer Science, Mumbai Education Trust, Bandra, Reclaimation, Mumbai, India

Vaishali P. Jadhav
Research scholar, NMIMS University
Vile Parle, Mumbai, India

ABSTRACT

Parallel database systems are being used nowadays in a wide variety of systems, right from database applications to decision support systems. These implementations involve database processing and querying over parallel systems. For the parallel databases to be effective and efficient, various optimizing solutions need to be implemented. These solutions deal with various issues associated with such database systems. In this paper, we focus on several techniques for query optimization in shared-nothing parallel database systems. Paper will discuss some of the ways in which queries can be optimized for parallel execution. It will look at various cost models, search algorithms and methods of generating query execution plans (QEPs), resource allocation techniques. It will also focus on some of the major issues associated with parallel databases and how well these algorithms address them. A survey of the proposed methods reveals their pros and cons. Taking advantage of strengths and eliminating weaknesses is the goal in implementing an algorithm. Paper focuses on a design method which fits each algorithm to the environment it is best suited for.

General Terms

Algorithms, Query Execution Plan, Cost Model.

Keywords

Parallel Database Systems, Query Optimization, Resource Allocation, System Architecture

1. INTRODUCTION

The development of database management systems has coincided with significant developments in distributed computing and processing technologies. The merging of these two resources has resulted in the emergence of parallel database management systems. These systems have become the dominant data management tools for highly data-intensive applications. The future of large database machines lie in the realm of highly parallel computers. The main reason for this is that parallel machines can be constructed at a low cost without the need for any specialized technology, using existing sequential computers and relatively cheap interconnection networks. Parallelism demonstrates two properties that make it desirable. The first is possible linear *speedup* where adding n times as many processors reduces response time by a factor of n. The second is linear *scaleup* where adding n times as many processors allows the system to perform a task n times the size in the same amount of time. Three problems exist which hinder parallelism in obtaining linear speedup and scaleup:

Startup: Starting a parallel system may require the creation and coordination of thousands of processes. If this task dominates the actual query processing time then an inefficient system results.
Contention: As more processes are created, contention for shared resources increases. This slows down the system.
Skew: Skewed data can lead to unbalanced load so that only a few processors are working while the rest are idle.

A perfect algorithm on a parallel machine would scale easily at a low-cost, and would demonstrate linear speedup and scaleup even for hundreds or even thousands processors. Three architectures have emerged from the quest for perfect parallel machine; they are shared-memory (SM), shared-disk (SD) and shared-nothing (SN).

In the shared memory architecture, each processor has access to all disks in the system and to a global shared memory. The main advantage of a shared-memory system is that it has zero communication cost; processors exchange messages and data through the shared memory. This also makes it easier to synchronize processes. Another advantage of such a system is that load imbalance can be corrected with minimal overhead. The drawback of such a system is the problem of contention. Shared-memory architecture can only be scaled to few processors (30-40) before interference begins to affect the rate at which the memory can be accessed. Paper further discussed why shared-nothing is preferred.

Shared-nothing is the extreme opposite. Every processor has its own memory and disks. Nodes communicate by passing messages to one another through an interconnection network. The main advantage of shared-nothing is its ability to scale to hundreds and potentially thousands of processors. The drawbacks of such a system are complications created by load-imbalance and its dependence on a high bandwidth interconnection network for effective message exchange. Another disadvantage to shared-nothing architecture is the problem of node failure. If a processor fails, then all access to the data owned by that processor is lost. A solution to the node-failure problem is a shared-disk architecture where every processor can read and write to any of the disks in the system but manages its own memory. We further discuss why SN is preferred.

Shared-nothing has emerged as the design of choice, mainly due to its high reliability and ease of scalability to hundreds of processors. Of course one of the main issues is cost, and since shared-nothing architectures can be constructed from existing sequential machines using a simple interconnection network, it is the most cost effective means of parallelizing a database system. Most of the current research in the field of parallel databases concentrates on shared nothing architectures. This fact is reflected in this paper where most of the algorithms and solutions discussed apply mainly to shared nothing systems.In examining the way in which queries are executed on parallel systems we observe two forms of parallelism:

Intra-operator parallelism: one operation is parallelized over several processors. On a shared-nothing system this is achieved by partitioning or de-clustering the data among the processors. When an operation such as a scan is performed each processor operates on its local data cluster, the results of each processor are then combined for the final result (which can also be partitioned among processors). There exist several methods for de-clustering data; these include round-robin distribution, range partitioning and hash partitioning.

Inter-operator parallelism: several operations are executed concurrently. Each of these operations is assigned to one or more processor. When more than one processor is used for a given operation, then we are also exploiting intra-operator parallelism. There are two forms of inter-operator parallelism: independent and pipelined. Independent means that no dependencies exist between operations, pipelined refers to a producer-consumer relationship between operators. The result of the producer operator is pipelined to the input of the consumer operator instead of being materialized. This has the obvious advantage of not having to wait for the input data to be completed before starting a new operation on the data.

2. SEARCH STRATEGIES

Optimization strategies for parallel execution come in two flavors: two-phase optimization and one-phase optimization. The first phase in two-phase optimization generates a query execution plan (without scheduling information). A resource allocation algorithm is then applied in the second phase to distribute the query into schedulable components. Modeling a QEP is the first step in searching for an optimal one. The method used in most approaches is to model QEPs as annotated join trees. The level of granularity is thus reflected in the nodes of the join tree, which are base relations at the leaves and components of join operations in internal nodes.

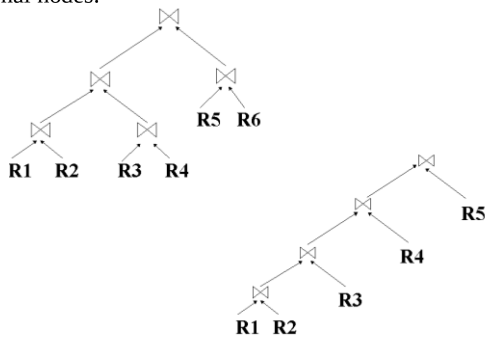


Fig.1 A bushy tree joining 6 relations and a linear left-deep tree joining 5 relations.

The problem of determining the optimal join ordering is NP-hard on the number of base relations. So an exhaustive search would be far too time consuming for larger number of joins. The number of possible parallel execution plans greatly exceeds the number of possible sequential plans due to the extra possibilities for exploiting parallelism such as partitioning and pipelining. Single processor optimizers limit the search space by considering only certain tree configurations. Search strategies can be organized as depicted in Fig.2.

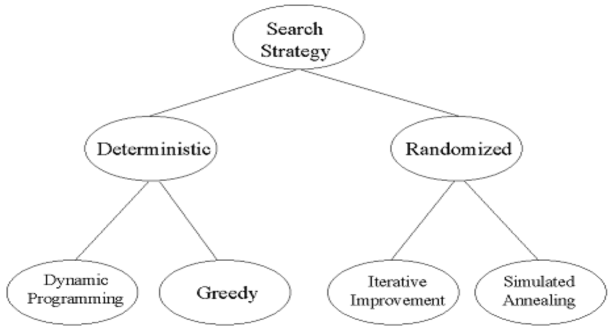


Fig.2 Organization of Search Strategies

Deterministic strategies proceed by building plans, starting from base relations and joining another relation at each step until the complete query is formed. When constructing QEPs through dynamic programming (DP), equivalent partial plans are constructed and compared based on some cost model. Partial plans that are cheaper are retained and used to construct the full plan.

Greedy algorithms are another deterministic strategy. They work by making the choice that seems best at the current iteration. The parallel multi-way join optimization algorithm GP, considered in [2], uses a greedy approach. The algorithm considers bushy QEPs as well as linear ones. To limit the search space QEPs are divided into synchronized and unsynchronized. Synchronized QEPs are those where the whole multi way join process is divided into synchronized steps. At each step a number of joins are executed concurrently and the joins at the next step will not begin executing until all joins in the previous step are complete. The greedy multi-way join algorithms works by selecting as many relations as possible to be joined concurrently at each synchronized step. In the beginning, all relations make up the working set T. A set of relation pairs, R, is selected for the first step by calling a function to select *k* pairs of relations to be joined at the current step. The function to select pairs of relations uses a minimum cost function that takes as its input the working set and the number of relation pairs *k* to be joined concurrently and determines the minimum cost plan that joins those *k* relation pairs first and determines the join method for each pair of relations. Unfortunately the GP algorithm only makes use of independent parallelism.

Randomized strategies work by searching for the optimal plan around some particular points. Such strategies do not guarantee the best plan, but they avoid the high cost of optimization incurred by other methods such as DP or exhaustive search. The first step is finding a start plan, which is usually built using some kind of simple greedy strategy. Next a random transformation is applied to the plan, for example, this may consist of switching two randomly chosen relations in the join order. Two randomized strategies are simulated annealing (SA) and iterative improvement (II). They differ on criteria for replacing the current plan with a transformed one and on the stopping criteria. II works similarly to the greedy algorithm. Starting from an initial state it repeatedly checks random neighboring plans (differed by a single transformation) and accepts those that exhibit a lower cost. By using different initial states, different local minimal plans can be obtained. The best of these local minimal plans is then chosen. As time approaches infinity, the probability of finding the global minimum approaches 1[6]. While II accepts only local minimums, SA can take steps that worsen the plan, to prevent being trapped in a high-cost local minimum. Such moves are taken based on a probability that diminishes at each iteration.

3. COST MODEL AND SEARCHING

Cost models are used by the optimizer to select the best QEP from among the candidates searched, as well as to allocate resources efficiently. The information included in a cost model depends on the system architecture, the type of search algorithm, the level of complexity considered acceptable, information describing the data and any assumptions that must be made for simplifying reasons.

***System Architecture*:** In SM architecture all communication is through the shared memory, hence repartitioning of tuples in SM is simply in-memory hashing, whereas SN would involve transfer of data across the interconnection network. The locality of data in a SD system is not relevant when allocating processor tasks, hence load-balancing is a much simpler issue than it would be with SN. A major factor affecting the SD architecture under heavy load is the increase in communication due to lock request messages. This must be considered when a single process requires many locks. Memory consumption in SN and SD is complicated by interoperation parallelism. In SM all operations use the same global memory making it easier to test whether there is enough space to execute them in parallel, i.e., the sum of the memory consumption by the individual operators is less than the available memory.

***Type of Search Algorithm**:* The cost model must preserve any fundamental principles that are necessary for the correctness of an algorithm (for instance, the principle of optimality for DP algorithms).

Semantic Data Descriptions: Optimizer cost models should also reflect semantic information about the data to aid in query processing. For example, if two relation R1 and R2 are being joined on a common attribute *a*, and *a* is a (non-null able) foreign key in R2, then we know that all of R2 will join with R1.

***Simplifying Assumptions*:** When designing a cost model, certain factors must be ignored either to simplify calculations or because some things are too difficult to measure.

The following three simple algorithms taken from [7], demonstrate the use of cost models. The first algorithm, GMC greedy, generates a join sequence from a join graph based on minimum cost. Consider the following profile of relation. Six relations with their cardinalities is given in the profile.

Relation	R1	R2	R3	R4	R5	R6
Cardinality	118	102	106	100	131	120

Fig. 3 Profile of Relations

GMC: Greedy Minimum Cost

```
1. repeat until $|V| = 1$
2. begin
3. Choose the join (Ri, Rj) from $G = (V, E)$
     such that $cost(Ri, Rj) = \min \{cost(Rp, Rq)\} \ \forall \ Rp, Rq \in V$
4. Perform (Ri, Rj) and merge result into G
5. Update profile accordingly
6. End
```

The above procedure uses the cost function to perform the minimum cost join first, the resultant relation is then added to the join graph and the profile for the relations is updated. The algorithm continues until a single relation is left in the join graph.

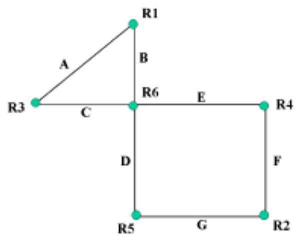


Fig. 4 An example query graph where the vertices are relations and edges are labeled with the joining attribute.

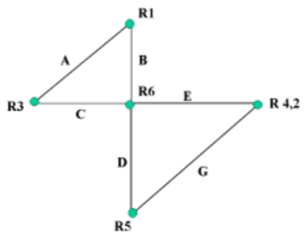


Fig. 5 The join graph after joining R4 and R2 on attribute F.

In a sequence of joins, the cardinalities of intermediate relations resulting from early joins affect the cost of joins to be performed later. The join procedure is assumed to be a Sort-Merge-Join, hence the *cost* function used by GMC is expressed by $|Ri| + |Rj| +$
$|Ri\times Rj|$. This formula is considered as a general approximation for joining large relations on a shared-nothing multi-processor system. Since the objective is to minimize the total cost required to perform a sequence of joins, we may want to execute joins that produce smaller resulting relations.

The following two algorithms are variations of GMC. The first is GMR which uses the minimal resulting relations. The second is GME which uses minimal expansion. The expansion of a join is defined as the size increase of the resulting relation over the two participating in the join.

GMR: Greedy Minimal Resulting Relation

```
1. repeat until |V| = 1
2. begin
3. Choose the join (Ri, Rj) from G = (V, E)
   such that |(Ri, Rj)| =  min {|(Rp, Rq)|} ∀ Rp, Rq ∈ V
4. Perform (Ri, Rj) and merge result into G
5. Update profile accordingly
6.End
```

GME: Greedy Minimal Expansion

```
1. repeat until |V| = 1
2. begin
3. Choose the join (Ri, Rj) from G = (V, E)
   such that |(Ri, Rj)| - |Ri| - |Rj| = min{|(Rp, Rq)| - |Rp|
   - |Rq|}∀ Rp, Rq ∈V
4. Perform (Ri, Rj) and merge result into G
5. Update profile accordingly
6. End
```

The resulting join tree from the join graph (Fig. 5) using GMC is (((R2,R4),(R5,R6)),(R1,R3)). The resultant join trees using GME and GMR are equivalent in this example: ((((R1,R3),R6),R5),(R2,R4)). Tests of the three algorithms were performed [7] on queries containing 4, 6, 8 and 10 relations. For each of the four query sizes, 300 queries were randomly generated. Test results for each of three algorithms, measuring average execution costs, were compared to the optimal costs produced by GOPT. The results obtained with GMR are better than the other two for each of the four query sizes. In second place is GME followed by GMC. The results are summarized in the table below (Fig. 6).

Number of relations	**GMC**	**GMR**	**GME**
4	107.3 %	106.8 %	107.1 %
6	107.9 %	105.9 %	106.1 %
8	124 %	110.9 %	112 %
10	134 %	124 %	128 %

Fig. 6 Percent of optimal cost in each algorithm

The GP greedy parallel multi-way join algorithm described earlier uses an interesting notion of cost. Two types of algorithms are presented; one based on total cost, GP_T and one based on partial cost, GP_P. The total cost of a QEP is computed as the sum of the costs of each synchronized step. It can be quite expensive to compute the total cost at each iteration, therefore a second version of GP, GP_P computes the partial cost of a QEP by calculating the cost of QEPs which join *k* pairs of relations concurrently only in the first step. In GP_T, the total cost of a QEP, *Costplan*, is computed as the sum of the costs at each step *i, Costi*. Even when the number of joins is small, enumerating all possible combinations of joins at each step for all possible sequences of steps can be very expensive. An important heuristic used to limit the search space in GP_T is to consider only those QEPs that execute *k* joins concurrently at the first step and all remaining joins sequentially. For such QEPs the plan cost is given by:

$$\text{Total}_k = \text{Par\_Join\_cost}(R_k) + \text{Seq\_Join\_cost}(T - R_k \bigcup \text{Join\_result}(R_k))$$

where Par_Join_cost (R_k) returns the cost of joining the *k* pairs of relations in the set R_k in parallel and Seq_Join_cost(R) returns the cost of joining the relations in the set R sequentially. T is the working set, and initially contains all the base relations. In the GP_T algorithm, the minimum *total* cost of a QEP is computed. Although the computation is simplified by considering only those QEPs that join *k* relations in parallel in the first step, it can still be expensive to compute such total costs at each iteration of the GP algorithm, especially as the number of joins increases. To further reduce computation overhead, the second version of GP, GP_P only estimates the *partial* cost of a plan, $Partial_k$, and uses this as the approximation of *Cost*plan. $Partial_k$, the cost of a QEP which joins *k* pairs of relations in the first step in parallel, is represented by

$$Partial_k = \text{Par\_Join\_cost}(R_k)$$

Finally, we present the GP algorithm below. The function *Minimum_cost*, can employ either the total cost or partial cost metrics described above.

GP: Greedy Parallel

Input: A join graph G = (V, E) where the set of vertices in V are relations and the edges in E represent joins (see Figures 4 and 5).
Output: S, the join sequence consisting of relation pairs

```
1. S ∅
2. while Size(T) > 3 do
3.        R    Select_Rel_Pairs(G)
4.        S    S ∪ R
5.        G   (G with each pair of relations in R replaced by their
              join result)
6. end
7. R   Two_Way_Sequential(G)
8. S    S ∪ R
```

The function Two_Way_Sequential is called to determine the join sequence when the working set T contains less than four relations.

Select_Rel_Pairs

Input: A join graph, G
Output: R, a set of relations pairs to be joined concurrently

```
1. k    0
2. repeat
3.     k    k + 1
4 .    $C_k$  Minimum_Cost(G, k, $R_k$)
5.     if ($R_k$ does not contain all relation in G) then
6.         $C_{k+1}$  Minimum_Cost(G, k+1, $R_{k+1}$)
7. until ($C_{k+1} > C_k$ ) or ($R_{k+1}$ contains all pairs in G)
8. if ($C_{k+1} > C_k$ ) then
9.     return $R_k$
10. else
11.    return $R_{k+1}$
```

Function *Minimum_Cost* takes as input the reduced join graph G, and the number of relations to be joined concurrently in the first step, *k*. It returns the minimum cost plan that joins *k* pairs first and determines those *k* pairs of relations as well as join methods for each of the *k* joins. The two versions of GP, GP_T and GP_P differ in the *Minimum_Cost* function. Simulated testing [7] of GPT and GP_P on QEPs that joined no more than ten relations showed that both produce plans with cost no more than 120% of the optimal plans. Algorithm GP_T outperforms GP_P in most cases. However, the time to generate plans is longer, especially as the number of relations in the QEP is increased. Up to 90% of the results generated by GP_P are 80% near GP_T.

4. RESOURCE ALLOCATION

As discussed in the beginning of the **Search Strategies** section, optimization methods fall into one of two categories, one-phase and two-phase. With a two-phase approach, the second phase is resource allocation, which takes as its input an operator or join tree and divides it into a set of schedulable objects. Most of the work done in this area assumes that input to the resource allocation phase is an optimal join-tree.

The basic approach used in [5] begins with a fixed number of *shelves*, each of which will be assigned one task initially. Each shelf will eventually contain one or more tasks that are to be executed concurrently by all processors. For each of the remaining tasks that have not been assigned a shelf, it is either assigned an existing shelf or a new shelf without violating the precedence constraints inherent in the QEP. When a task is added to an existing shelf it will be competing for resources with other tasks. As tasks are assigned, the total number of shelves may increase. The initial number of shelves is determined by the height of the task tree since each task along the longest path must be processed one after the other due to precedence constraints. The join tree decomposition and task tree formulation steps are similar the method presented in [4] but of a much coarser granularity. Tasks in this context are basically right-deep segments of a join tree. When there are more tasks than the ones on the longest path, the task with the highest work estimate is selected. The work estimate of a task is the resource consumption of the task and the work estimate of a shelf is the total of the work estimates of tasks on that shelf.

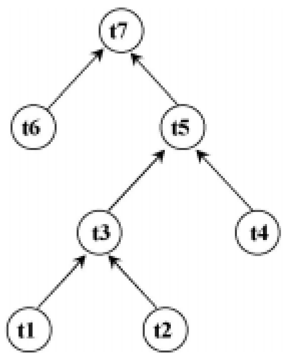


Fig. 7 The Task Tree

The three *shelf-scheduling* strategies we will discuss are *Max-Shelf*, *Min-Shelf* and *Flexi-Shelf* [4], [5]. In the *Max-Shelf* strategy the total number of shelves is maximized, hence each shelf contains a single task. Shelves are processed one at a time and there is no inter-operation parallelism. The *Min-Shelf* method is the other extreme; it produces the minimum number of shelves. Starting with as many shelves as tasks on the longest path, task are assigned to each of these shelves by adding a task to the shelf with the lowest work estimate. This strategy makes full use of inter-operation parallelism. The *Flexi-Shelf* algorithm, as implied by its name, uses a more flexible method for assigning tasks to shelves. Again, the initial numbers of shelves are determined by the height of the task tree. A task *t*, is assigned to an occupied shelf (that satisfies precedence) with a set of tasks *T*, only if it is cheaper to process *T* and *t* in parallel than to process *T* and *t* sequentially. When there is more than one candidate shelf available then the one with the most gain over sequential execution is chosen.

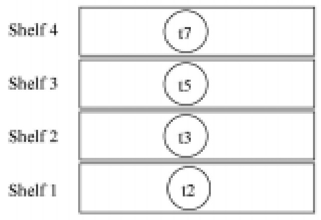
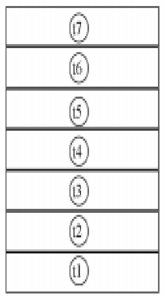
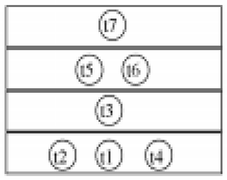


Fig. 8 Initial Schedule





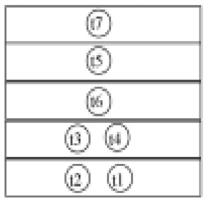


Fig. 9 Max-Shelf Fig.10 Min Shelf Fig. 11 Flexi - Shelf

The *Max-Shelf* algorithm executes tasks serially. This performs well on systems with large memory, especially when pipelining can be employed for several relations that fit into memory. When memory is scarce or when relation and join results are very large, inter-operation parallelism may produce better results. The *Min-Shelf* employs as much inter-operator parallelism as possible; however, it does so without considering reduced resources, which can result in high processing costs. *Flexi-Shelf* is a hybrid of *Min-shelf* and *Max-Shelf* so it has each of their advantages but avoids their shortcomings and performs significantly better than the other two, both on large memory systems as well as those where memory is scarce.

5. CONCLUSION

Paper has discussed some of the major optimization issues currently being addressed in parallel, shared-nothing databases. Paper focused how each stage in optimizing a query is significantly affected by choice of search algorithm, resource allocation strategy and cost models. It also focused how addressing these issues play an important role in implementing a parallel database system. As it turns out there is no best way to implement a parallel DBMS. There are many different ways, each best suited to a given description of the data and the available hardware. It seems that parallel database machines are the future, but with this future comes the challenge of devising strategies which fully exploit the potential of these systems. Rising to this challenge we have attempted to bring into focus some of the interesting research problems related to parallel database query optimization.

6. REFERENCES

[1] S.Ganguly, W.Hasan, R. Krishnamurthy, "Query Optimization for Parallel Execution", ACM SIGMOD 6/92 California, USA

[2] H. Lu, M-C. Shan, K-L Tan, "Optimization of Multi-Way Join Queries for Parallel Execution" Proceedings of the 17th international Conference on Very Large Databases, Barcelona, September 1991

[3] M.N. Garofalakis, Y.E. Ioannidis, " Multi-Dimensional Resource Scheduling for Parallel Queries", SIGMOD '96, 6/96 Montreal, Canada

[4] T.M Niccum, J. Srivastava, B. Himatsingka, J. Li, "A Tree-Decomposition Approach to the Parallel Execution of Relational Query Plans" Department of Computer Science, University of Minnesota, Technical Report TR93-016, 1993

[5] K-L Tan, H. Lu, "On Resource Scheduling of Multi-Join Queries in Parallel Database Systems", Information Processing Letters 48,1993

[6] R.S.G. Lanzelotte, et al "Industrial-Strength Parallel Query Optimization: Issues and Lessons", Information Systems Vol. 19 No.4, 1994

[7] Ming-Syan Chen, Philip S. Yu, Kun-Lung Wu, "Scheduling and Processor Allocation For Parallel Execution of Multi-Join Queries" 8th International Conference on Data Engineering, 1992

[8] H. Lu, K-L Tan, "Load-Balanced Join Processing in Shared-Nothing Systems" Journal of Parallel and Distributed Computing 23, 382-398, 1994

[9] D. DeWitt, et al, "The Gamma Database Machine Project" Computer Sciences Department, University of Wisconsin, 1990.

[10] D. DeWitt, J. Gray, "Parallel Database Systems: The Future of Performance Database Systems" Communications of the ACM, June 1992, Vol.35, No. 6

[11] Ming-Syan Chen, Philip S. Yu, Kun-Lung Wu, "Scheduling and Processor Allocation For Parallel Execution of Multi-Join Queries" 8th International Conference on Data Engineering, 1992

[12] Hongjun Lu, Kian-Lee Tan, "Dynamic and Load-balanced Task-Oriented Database Query Processing in Parallel Systems" Thirst International conference on Extending Database Technology" 357-372, 1992

[13] Kien A. Hua, Chiang Lee, "Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning", 17th International Conference on Very Large Databases, 525-535, 1991

[14] G. Hal lmark, "Oracle Parallel Warehouse Server", IEEE 1997

[15] H. Lu, M-C. Shan, K-L Tan, "Optimization of Multi-Way Join Queries for Parallel Execution" Proceedings of the 17th international Conference on Very Large Databases, Barcelona, September 1991

[16] M.N. Garofalakis, Y.E. Ioannidis, " Multi-Dimensional Resource Scheduling for Parallel Queries", SIGMOD '96, 6/96 Montreal, Canada.

AUTHOR PROFILE

Sunita Mahajan is currently Principal, Institute of Computer Science, at MET, Mumbai. She worked in Bhabha Atomic Research Centre for 31 years. She did her PhD in Parallel Processing in 1997 from SNDT Women's University and M.Sc. degree from Mumbai University in Physics in 1966. She is a member of Indian Women Scientists Association, Vashi. Her research areas are Parallel Processing, Distributed Computing, Data mining and Grid Computing.

Vaishali P. Jadhav is a Lecturer in St. Francis Institute of Technology, Borivali. She is a Ph.D. student in NMIMS University Vile Parle, Mumbai. She has completed her Masters in Computer Engineering from Thadomal Shahani College of Engineering Bandra, Mumbai. She is a life time member of ISTE. Her area of research is Database Management, Advanced Databases, Distributed Computing, Operating systems, Artificial Intelligence.