2009 International Workshop on High Performance Computational Systems Biology

On the performances in simulation of parallel databases: an overview on the most recent techniques for query optimization

A.B.M.Rubaiyat Islam Sadat
Department of Information Engineering and Computer Science
University of Trento
Trento, Italy
Email: sadat@science.unitn.it

Paola Lecca
Center for Computational and Systems Biology
The Microsoft Research-University Of Trento
Trento, Italy
Email: lecca@cosbi.eu

***Abstract*—The query optimization is a very big field in the context of database management.It has been studied in a great variety of contexts and from many different perspectives, giving rise to several diverse solutions in each case. The purpose of this paper is to primarily provide a comprehensive review and discussion of the core problems which the techniques of query optimization generally cope with by simulating a parallel database environment in different processing units. In addition to that this paper focuses on the skewing problem in parallel database architectures with proper survey of literature.**

***Keywords*-parallel database; skew; partition join; query optimization;**

I. Introduction

Many areas of modern biology are concerned with the management, storage and visualization of data in databases [8] and there exists a few appropriate query languages, like Pathway Query Language (PQL) [2] for such complex data structure. In a parallel database system (PDBS), as in a biological database, the effect of skewing is considered one of the most costly goals to achieve. Skew is defined as an uneven distribution of data and/or workload across the system's resources[1]. Due to several insert and delete operations are performed, the balance of data distribution is ruffled. We have studied several algorithms for computing the join operation in a PDBS to achieve a minimal skew effect [1,10]. Taxonomy of data skew in parallel joins includes the aspects of intrinsic and partition skew[10]. We have built a technique for partition join which uses parallel hash algorithm to reduce memory contention which makes it different from the technique described in [12]. Our solution tries to combine tha advantages of hash partitioning and symmetric join which is described in Fragment and Replicate join technique. This technique splits the pairs to be tested over several processors. Each processor computes part of the join, and then the results are merged. Then we analyzed this procedure by comparative studies with other join techniques. We present a classification of skew effects and also load balancing approaches. This paper aims to find the appropriate load balancing methods for different forms of unbalanced data which is called "Skew". As there is no single way to deal with the skewing problem in parallel database system, we have studied different algorithms and simulated them in a parallel database system and we have tried to find the best solution to deal with general skewing. We studied the algorithms so that load balancing is also optimized for parallel databases as in a parallel database system, unbalanced data can lead to a large amount of processing time whereas a smaller amount is needed in a parallel environment.

A. State-of-the-art Parallel Database Architectures and Query Optimization

Parallel database system architectures range between two extremes- the shared-nothing and the shared-memory architectures [3]. A useful intermediate point is the shared-disk architecture. Commercial parallel relational DBMSs such as NCR Teradata [5], IBM DB2 MPP [7], Tandem NonStop SQL [6] are based on the shared-nothing architecture. Relational tables are partitioned across a collection of nodes. Here, some hash-based partitioning function is used to distribute rows. Query processing is the process by which a declarative query is translated into low-level data manipulation operations[4]. Query processing includes query decomposition which takes a query and restructures it into a simplified query expressed in relational algebra after modification due to views, security enforcements and semantic integrity control. Upon the measurement of expected performance, the best algebraic query is found after algebraic transformation rules[12]. The best algebraic query is determined according to a cost function which calculates the cost of executing the query according to that algebraic specification. This is the process of query optimization. Query optimization consists of finding the best one among candidate plans examined by the optimizer[11]. Parallel query optimization exhibits similarities with distributed query processing[9]. It takes advantage of both intra-operation and inter-operation parallelism. Intra-operation parallelism is achieved by executing an operation on several nodes of a multiprocessor machine. Typically, partitioning is performed by applying a hash function on

978-0-7695-3809-9/09 $26.00 © 2009 IEEE
DOI 10.1109/HiBi.2009.25

an attribute of the relation, which will often be the join attribute. Inter-operation parallelism occurs when two or more operations are executed in parallel, either as a dataflow or independently.

II. Partitioning Techniques

We consider three basic data-partitioning techniques[12]. Let us assume that there are *n* disks, $D_0,D_1,\ldots,D_{n-1}$, across which the data are to be partitioned. The first technique is Round Robin technique. In Round robin strategy, it scans the relation in any order and sends the *i*-th tuple to disk number $D_{i \bmod n}$. The round robin scheme ensures an even distribution of tuples across disks; i.e. each disk has approximately the same number of tuples as others. The second technique, Range partitioning strategy, distributes contiguous attribute-value ranges to each disk. It chooses a partitioning attribute, **A**, as a partitioning vector. The relation is partitioned as follows. Let $\mathbf{[v_0,v_1,...,v_{n-2}]}$ denote the partitioning vector, such that, if $i<j$, then $v_i<v_j$. We consider a tuple *t* such that t[A] = x. If $x<v_0$, then *t* goes on disk D_0. If $x \geq v_{n-2}$, then *t* goes on disk D_{n-1}. If $v_i \leq x \leq v_{i+1}$, then *t* goes on disk D_{i+1}. The third technique, Hash Partitioning declustering strategy, designates one or more attributes from the given relations schema as the partitioning attributes. A hash function is chosen whose range is $[0,1,\ldots,n-1]$. Each tuple of the original relation is hashed on the partitioning attributes. If the hash function returns *i*, then the tuple is placed on disk D_i.

III. Skewing

As skewing is one of the major topics to be considered in parallel database system, sooner or later, it has to be removed. One of the major decisions to be made is when to remove skew. The only way to remove skew is to redistribute or transfer some amount of data from one disk to another[10]. Transfer of data is a time consuming process. So it must be assured that the reduction of processing time after transfer is greater or equal to the time required to transfer some amount of data from one processor to another.

IV. Join Operation

We have studied several algorithms for computing the join of relations and we have analyzed their respective costs[12]. These algorithms take two relations and based on the specificity of the algorithm, the join operation is conducted. The algorithms are nested loop join, block nested loop join, indexed loop join and merge join.

The Nested loop join algorithm examines every pair of tuples in two relations. This technique is a simple and starightforward way to compute join between two relations and performs better when the number of tuples is not so large in the relations. Advantage of this approach is that it is simple but it experinces a huge amount of computational time.

The Block Nested Loop Join is a variant of the nested loop join where every block of the inner relation is paired with every block of the outer relation. Within each pair of blocks, every tuple in one block is paired with every tuple in the other block, to generate all pairs of tuples. All pairs of tuples that satisfy the join condition are added to the result. The block nested loop join algorithm is a generalization of the simple nested loops algorithm that takes advantage of additional memory to reduce the number of times that any relation is scanned. If there are many qualifying tuples for one relation, and particularly if there is no applicable index for the join key on another relation, this Block Nested Loop Join operation will be very expensive.

On the other hand, in Indexed Loop Join, temporary indices are created for the sole purpose of evaluating the join. For each tuple t_r in the outer relation *r*, the index is used to look up tuples in relation *s* that will satisfy the join condition with tuple t_r. So by using indices, it's easy to find the tuples in the relation and join them. This approach makes the performance better in Loop joins.

The next technique is the Merge Join. In this algorithm, the common attributes of two sorted relations are computed by associating one pointer with each relation. This pointer points initially to the first tuple of the respective relations. As the algorithm proceeds, the pointers move through the relation. When the pointer reached to the end of the relation, the result is constructed. The advantage of this approach is that it is faster in case of joining based on matching attributes. The only disadvantage of this approach is the cost for initial sorting of the relations.

V. Parallel Join techniques

Parallel join algorithms attempt to split the pairs to be tested over several processors. Each processor then computes part of the join locally. Then, the system collects the results from each processor to produce the final result.

A. Partitioned Join

This technique works in this way: the system partitions the relations *r* and *s* each into n partitions, denoted $r_0, r_1,\ldots,$ r_{n-1} and $s_0, s_1,\ldots, s_{n-1}$. The system sends partitions r_i and s_i to processor P_i, where their join is computed locally. The partitioned join works correctly only if the join is an equi-join and if we partition *r* and *s* by the same partitioning function on their join attributes. In a partitioned join, there are two different ways of partitioning *r* and *s*:

- Range partitioning on the join attributes
- Hash partitioning on the join attributes

In either case, the same partitioning function must be used for both relations. For range partitioning, the same partition vector must be used for both relations. For hash partition, the same hash function must be used on both relations. Once the relations are partitioned, we can use any join technique like

hash-join, merge-join, or nested-loop join, locally at each processor P_i to compute the join of r_i and s_i.

B. Fragment-and-Replicate Join

If the join condition is an inequality, such as $r \bowtie_{r.a<s.b} s$, where a and b are attributes, it is possible that all tuples in r join with some tuple in s and vice versa. Thus, there may be no easy way of partitioning r and s so that tuples in partition r_i join with only tuples in partition s_i. We can parallelize such joins by using a technique called fragment and replicate. The general case of fragment and replicate join works this way: The system partitions relation r into n partitions, $r_0,r_1,\ldots,r_{n-1}$, and partitions the relation s into m partitions, $s_0,s_1,\ldots,s_{m-1}$. This is specifically called the symmetric join. Any partitioning technique may be used on r and s. The values of m and n do not need to be equal, but they must be chosen so that there are at least m∗n processors. Let the processors be $P_{0,0},P_{0,1},\ldots,P_{0,m-1},P_{1,0},P_{1,1},\ldots,P_{n-1,m-1}$. Processor $P_{i,j}$ computes the join of r_i with s_j. Each processor must get the tuples in the partitions it works on. To do so, the system replicates r_i to processors $P_{i,0},P_{i,1},\ldots,P_{i,m-1}$ and replicates s_i to processors $P_{0,i},P_{1,i},\ldots,P_{n-1,i}$. Any join technique can be used at each processor $P_{i,j}$ locally. One variant of this fragment and replicate join is asymmetric fragment and replicate join where one relation, s, remains non-fragmented and the other relation r is fragmented and this fragmented tuples are sent over the network for parallel computation.

VI. Experimental results

We set up a parallel environment where we simulated parallel databases in 25 different computers in which data was stored in simple text files. The text files were created like real databases. The texts were arranged in two dimensional matrix with tuples denoted in the rows and attributes denoted in columns. The computers were connected through LAN and the configuration of the each of the CPUs was: Pentium-4 processor with clock speed of 1.86GHz, and as main memory RAM 1GB each. We simulated a simple program by which we generated random texts written in text files described above which were saved in the disk and acted as the database. Then we computed the time required to execute a join query in one of the nodes. By varying different parameters ,we simulated the different join techniques in parallel environment. After getting these values of transfer time and assemble time, we got the plots as follows.

First we like to present the experimental values obtained on single processor. The experiment was conducted several times by changing the parameters like block transfer time, disk access time, number of processors, number of tuples related to the query that has been sent via the network to the other nodes. By symmetric, we mean the symmetric Fragment and Replicate join and by asymmetric, we mean the asymmetric Fragment and Replicate join. For all the techniques, we considered only equi-join in which only equality comparisons in the join-predicate are considered.

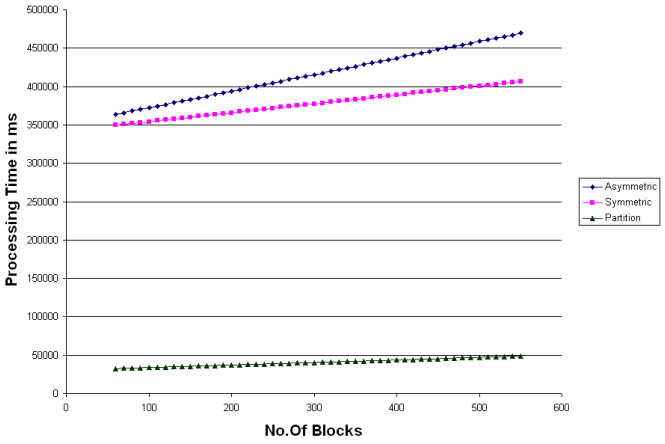

Figure 1. In this figure the block transfer time is varied. In the X-axis, we have No. of blocks that are varied and in the Y-axis, we have the observed processing time in millisecond

A. Varying block transfer time

In figure 1, we have found that the time for partition join remains almost unchanged and asymmetric and symmetric joins require more processing time than partition join.

B. Varying disk access time

In figure 2, Here, we found that the partition join requires less processing time than both of asymmetric and symmetric join and remains almost unchanged. In comparison between symmetric and asymmetric join, it has been found that symmetric join is better with respect to the processing time for asymmetric join.

C. Varying number of processors

When we varied the number of processors, we observed that in the beginnig the partition join required more time and it decreases if we increase the number of processors. In all the cases it takes less processing time than both of asymmetric and symmetric join. It is evident from the Figure 3 that symmetric join performs better than asymmetric join.

D. By varying R (the number of tuples that is sent to other nodes)

In figure 4, we found that the partition join requires less processing time than both of asymmetric and symmetric join and remains almost unchanged. In comparison between symmetric and asymmetric join, it has been found that symmetric join is better with respect to the processing time for asymmetric join.(Figure 5)

Now we conducted the experiments in parallel environment where the query is optimized for parallel execution and

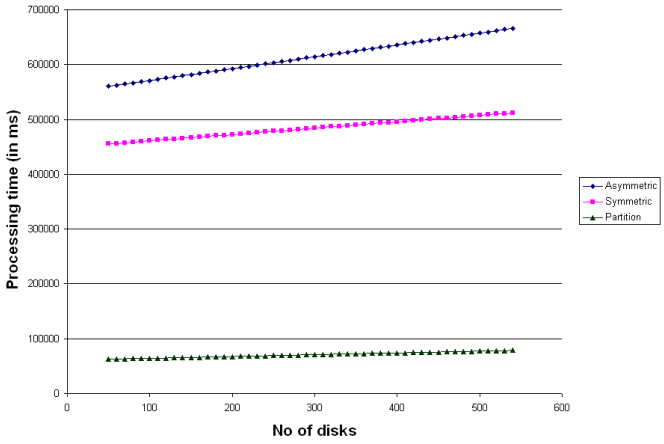

Figure 2. In this figure the no. of disks is varied so that the disk transfer time changes. In the X-axis, we have No. of disks that are varied and in the Y-axis, we have the observed processing time in millisecond

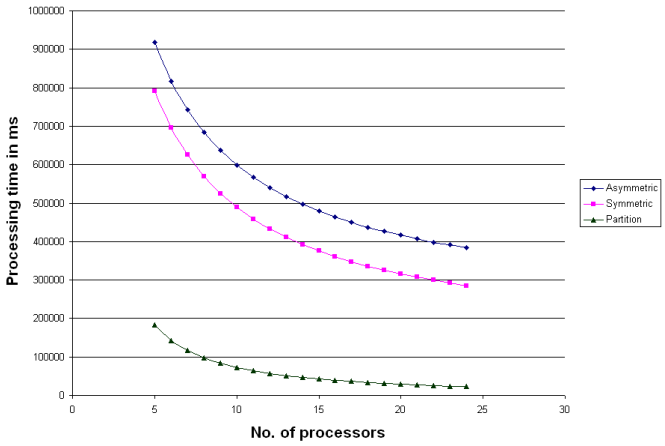

Figure 3. In the X-axis, we have variable no. of processors and in the Y-axis, we have the observed processing time in millisecond

in this case we considered the number of processors and the number of tuples in R as these two parameters are the most important ones for the simplicity of comparison.

E. By varying number of processors

In this case we observed some oscillations in case of symmetric join. The reason behind is that when the multiplication of the number of tuples for both relation matches or very close to the number of processors, it takes less time for computation in symmetric case. Otherwise, the symmetric join acts as the aymmetric join. But as a whole the partition join acts better regarding the processing time. As a matter of fact, the prcessing time is greater when the number of processors is low, but it decreases with the increase of the processing units. (Figure 5)

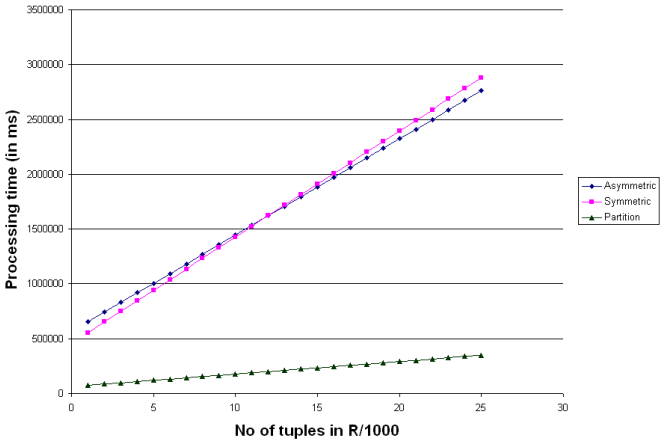

Figure 4. In this figure the no.of tuples that are sent to other nodes through network is varied. In the X-axis, we have No. of tuples in R normalized by the factor 1000 and in the Y-axis, we have the observed processing time in millisecond

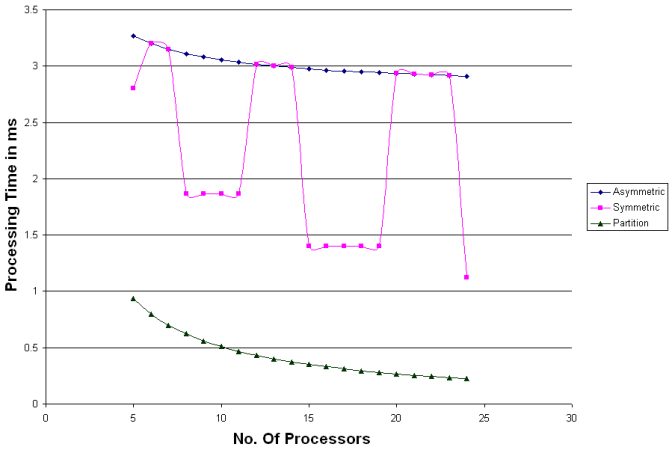

Figure 5. In this figure the no.of processors is varied in case of parallel join. In the X-axis, we have No. of processors that are varied and in the Y-axis, we have the observed processing time in millisecond

F. Varying the number of tuples in R

At the beginning, we observed some oscillations in case of symmetric join, the reason of which is same as subsection E. In some cases symmetric join performed better than asymmetric join as the number of tuples matched with the total number of processors used in the simulation. As the number of tuples in relation R grew more, the performances of symmetric and asymmetric join became linear and identical. Here, too, the partition join performs better regarding the processing time. (Figure 6)

If we consider all the above graphs, it is obvious that partition join acts better in all the cases. And also if the parallelism is adopted, the processing time reduces in a greater extent.

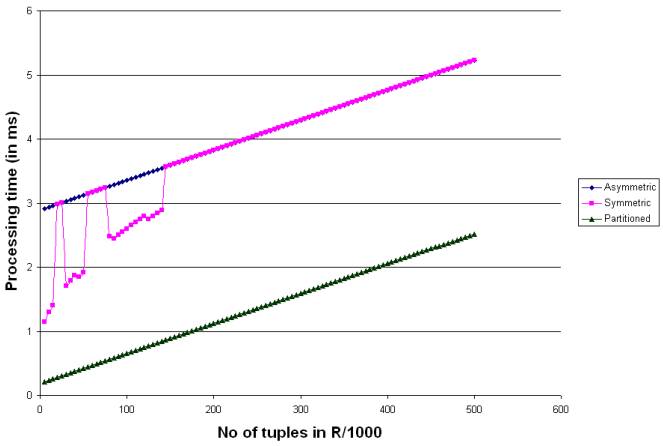

Figure 6. In this figure the no.of tuples that are sent to other nodes through network is varied in case of parallel join. In the X-axis, we have No. of tuples in R normalized by the factor 1000 and in the Y-axis, we have the observed processing time in millisecond

VII. Discussion

When we considered a single processor, the nested loop join, block nested loop join, indexed loop join and merge join showed almost the same characteristics in case of varying the block transfer time. When we varied disk access time and the number of tuples in R (the relation that has to be sent over the network) in single processor, the above join techniques and also the asymmetric and symmetric joins showed the same behavior. Both in single processor and parallel environment, partitioned join technique showed far better performance in all the variations of the different parameters. We find that highly dynamic scheduling methods based on observed execution times are superior in both complexity and attainable load balance. We also suggest the tuning of database schemata as a new anti-skew measure.

In practice, the speedup is less dramatic because

- Overhead is incurred in partitioning the work among the processors.
- Overhead is incurred in collecting the results computed by each processor.
- If the split is not even, the final result cannot be obtained until the last processor has finished.
- The processors may compete for shared system resources.

VIII. Conclusion

In this paper, we simulated a parallel database and developed a partition join where both skew effects and load balancing techniques were adopted. Then we compared various techniques to achieve a conceptual framework for the assessment of existing processing approaches and for the development of new algorithms. We found that highly dynamic techniques have great advantages with respect to their own complexity as well as to the expected success of load balancing because they rely on observed execution times rather than inaccurate cost estimates. We particularly favor on-demand scheduling methods treating execution skew and strongly recommend to pursue this approach further. In addition, we noted an unexplored potential for skew treatment in the design of database schemata and materialized views. We consider this an interesting line of research for efficient parallel query processing.

Acknowledgment

The authors would like to thank Dr. A.S.M.Latiful Hoque, Associate Professor of Department of Computer Science and Engineering, BUET for his utmost contribution for the arrangement of permission to use the computer laboratory where the simulation was taken place.

References

[1] Holger Märtens, *A Classification of Skew Effects in Parallel Database Systems*, Volume 2150/2001. Springer Berlin / Heidelberg, 2001.

[2] Ulf Leser, *A query language for Biological networks*, Vol-21, Suppl.2. Bioinformatics, 2005.

[3] D. DeWitt and J. Gray, *Parallel Database Systems: The Future of High-Performance Database Systems*, Communications of ACM, 35(6):85−98, June 1992.

[4] M.T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[5] A. Shatdal, *Order Based Analysis Functions in NCR Teradata Parallel RDBMS*. In EDBT 2000.

[6] D. Wildfogel, and R. Yerneni, *Efficient Testing of High Performanfce Transaction Processing Systems*. VLDB 1997.

[7] C. Baru and G. Fecteau, *An overview of DB2 parallel edition*. SIGMOD 1995.

[8] M. Y. Galperin, *The molecular biology database collection: 2008 update*. Nucleic Acids Research, 36:D2−D4, 2008.

[9] P. Roy, S. Seshadri, S. Sudarshan, S. Bhobe, *Efficient and extensible algorithms for multi query optimization*. In SIGMOD, p.249−260. ACM, 2000.

[10] C. B. Walton, A. G. Dale, R. M. Jenevein, *A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins*. Proc. 17th VLDB Conf., Barcelona, 1991.

[11] S. Ganguly, W. Hasan, R. Krishnamurthy, *Query optimization for parallel execution*. In 1992 ACM SIGMOD Conference, pages 9-18, May 1992.

[12] A. Silberschatz, H. F. Korth, S. Sudarshan, *Database Systems Concepts*, 5th ed. McGraw-Hill, ISBN 0-07-295886-3, 2005.