

Workload-based Placement and Join Processing in Node-Partitioned Data Warehouses

Pedro Furtado

Centro de Informática e Sistemas
Engenharia Informática, Univ Coimbra (DEI /CISUC)
pnf@dei.uc.pt

Abstract. Data warehouses (DW) with enormous quantities of data put major performance and scalability challenges. The Node-Partitioned Data Warehouse (NPDW) divides the DW into cheap computer nodes for scalability. Partitioning and data placement strategies are relevant to the performance of complex queries on the NPDW. In this paper we propose a partitioning placement and join processing strategy to boost the performance of costly joins in NPDW, compare alternative strategies using the performance evaluation benchmark TPC-H and draw conclusions.

1 Introduction

Data warehouses (DW) are large repositories of historical data used mainly for analysis. The applications that allow users to analyze the data (e.g. iterative exploration) have to deal with Giga or Terabytes of data, while users require fast answers. As return-on-investment considerations become crucial, it is desirable that scalability be obtained dynamically and at reasonable cost without throwing away low cost platforms.

Over the past there has been an large amount of work on related and intertwined issues of data allocation [1, 4, 6, 11, 16] and query processing strategies in parallel and distributed databases [3, 8, 9, 13]. Data allocation in parallel and distributed databases has been studied extensively, typically in heavy transactional environments. In an early work [10] compares full partitioning with clustering the relations on a single disk, concluding that partitioning is consistently better for multi-user workloads but can lead to serious performance overhead on complex queries involving joins, with high communication overheads. [4] proposes a solution to data placement using variable partitioning. In that work the degree of partitioning (n° of nodes over which to fragment a relation) is a function of the size and access frequency of the relation. Again, experimental results in [4] show that partitioning increases throughput for short transactions but complex transactions involving several large joins result in reduced throughput with increased partitioning.

Heavy query environments such as a data warehouse with complex query patterns can exhibit such a high communication overhead, as complex joins are executed over several relations. Whether the data warehouse is a set of “pure” star sub-schemas, with a set of dimensions describing business elements and very large facts with busi-

ness-related measures, or a less structured warehouse schema such as TPC-H [17], the common features are some very big relations, typically facts, with tens to hundreds of million rows and heavy join activity over those big relations and other smaller ones. This is exactly the kind of environment where speedup and scale-up can be hampered by heavy communication costs. We focus on join processing costs in this paper.

One of the crucial issues with partitioning in the presence of costly joins, which we review more profoundly in the next section, is the need to exchange significant amounts of data between nodes, which is especially costly given slow interconnects. Parallel hash-join algorithms [5, 7] (PHJ) provide an efficient way to process heavy joins in such environments. Data placement algorithms on the other hand typically do not bother with the join strategy and its relationship with the query workload.

In this paper we propose the use of workload-based placement and PHJ processing of joins in the specialized structure and context of the data warehouse to speedup complex joins. We show why the use of PHJ is relevant in the NPDW and take into consideration workload access patterns in a simple but new way when determining data placement or reorganization of the data in NPDW.

Related work on partitioning and processing strategies includes other join processing strategies such as PRS [15] or placement dependency [3, 9]. PRS partitions one relation and replicates the other ones and placement dependency co-locates relation partitions exhibiting dependency [3, 9]. Once again, these strategies assume data placement to be done independently from join processing. In [18] an automated partitioning and placement strategy is proposed for generic databases and in [11] an automated partitioning and placement strategy is designed for DB2 which uses the query optimizer to evaluate costs of alternative allocations. Our work is related to these ones but focuses on a simple allocation strategy targeted at data warehouses and which does not require any tight integration with the query optimizer as in [11].

The paper is organized as follows: section 2 discusses the partitioning issue. Section 3 describes data placement generically in NPDW. Section 4 discusses relevant processing costs in NPDW. Section 5 discusses our solutions to data placement and processing that result in small overheads. Finally, section 6 analyzes some experimental performance results using the TPC-H decision support benchmark [17]. Section 7 contains concluding remarks.

2. The Partitioning Issue

The basic partitioning problem is well described in [9]. Assuming all queries of the form $Q = \{ \text{target} \mid \text{qualification} \}$, where target is a list of projected attributes and qualification is a list of equi-joined attributes, let a query be $Q = \{ R_1.A, R_2.B \mid R_1.A = R_2.A \wedge R_2.B = R_3.B \}$ and assume that R_1 and R_2 reside in different nodes shown in Figure 1. Then, not all of the three relations can be partitioned to process the join, since the first join predicate requires that R_2 be partitioned on A and the second join predicate requires that R_2 be partitioned on B. We may choose to partition R_1 and R_2 on A and replicate R_3 or to partition R_2 and R_3 on B and replicate R_1 . The problem is then to decide which set of relations to partition.

Relation	Tuples	Site 1	Site 2
R1	12000	R1	
R2	10000		R2

Fig. 1. Example Layout of R1 and R2

The partition and replicate strategy (PRS) [15] involves partitioning R_1 into two fragments F_{11}, F_{12} and sending F_{12} to site 2. Relation R_2 is sent to site 1 and then each site processes its part of the join: $R_1 \times_A R_2 = (F_{11} \times_A R_2) \cup (F_{12} \times_A R_2)$ processed in parallel in sites 1 and 2, where \times_A denotes an equi-join on attribute A.

Using hash-partitioning [5, 8, 12], it is possible to obtain a more efficient result. We first define hash-partitioning.

Definition 1. A relation R_i is hash-partitioned on attribute A into d disjoint fragments $\{F_{ij}\}$ if 1) $R_i = \cup F_{ij}$; 2) $F_{ij} \cap F_{ik} = \emptyset$ for $j \neq k$; and 3) $\forall T \in F_{ij}; h(T.A) = c_{ij}$, where $h()$ is a hash function and c_{ij} is a constant for a given j .

Hash partitioning allows the expression $R_1 \times_A R_2$ to be processed faster as $(F_{11} \times_A F_{21}) \cup (F_{12} \times_A F_{22})$, as fragments are results of hashing the relations such that $F_{11} \times_A F_{22} = \emptyset$ and $F_{12} \times_A F_{21} = \emptyset$. The join cost at each site is now smaller as only a fragment of R_2 is joined instead of the whole relation. Communication costs are also reduced, as only a fragment instead of the whole relation R_2 must be moved. On the other hand, hash-partitioning R_1 and R_2 introduces some overhead. However, if a query joins R_1 to R_2 by another join attribute or one of the relations participates in a join with other relation on a different attribute, it is necessary to repartition the fragments and redistribute them before this other join can take place. For instance, R_1 can be repartitioned by repartitioning fragments F_{11} and F_{21} in parallel and exchanging data to build F'_{11} and F'_{21} in nodes 1 and 2 respectively. Although repartitioning can occur in parallel at all nodes, it entails processing and data communication costs.

Unbalanced placement and processing load is a concern with hash-partitioning which can be minimized by using a suitable hash function, managing and allocating a number of fragments larger than the number of nodes for better load balancing. With this concern taken into consideration, hash-partitioning can be advantageous. Our objective is to minimize extra communication overheads due to partitioning or repartitioning needs as well as join processing costs. Join processing cost is minimized by joining only fragments from all except small relations. Communication overhead is minimized in the above example when F_{11}, F_{21} are initially located at node 1 and F_{12}, F_{22} are initially located at node 2, as the join can proceed without any (re)partitioning and communication costs (except to merge the partial results). It is therefore important to determine the best hash-partitioned placement in the presence of a query workload.

3. NPDW Placement Basics

A Node Partitioned Data Warehouse (NPDW) is a data warehouse divided into a set of nodes and a query processing middle layer that uses those nodes for fast query

processing. In order to process joins efficiently, we propose hash-partitioning every large relation and replicating small ones. If we consider a strict star schema with only small dimensions and a single large fact, Figure 2 shows the resulting data placement. Join processing in the system of Figure 2 is similar to the PRS strategy, but with relations already replicated at the processing nodes. As a result, all nodes can process independently and return their partial answer for a posterior merge step to obtain the final answer.

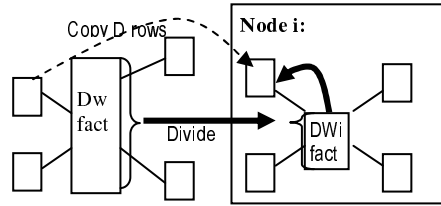


Fig. 2. Basic Star-Partitioning

Actual schemas are typically more complex than a pure star, resulting in not so independent node processing of partial queries and higher repartitioning and communication costs. Schemas can have several stars with cross-querying, possibly larger dimensions and possibly a less pure star design. Joins with large dimensions and with more than one fact are frequent. The most important concern in such an environment is how to partition facts and large dimensions horizontally to solve the partitioning issue described in section 2 as efficiently as possible. More generically, the concern is how to handle data placement and join processing for larger relations and therefore minimize repartitioning and communication costs?

TPC-H [17] provides an example schema to better illustrate our rationale. Figure 3 summarizes the schema, which represents ordering and selling activity (LI-lineitem, O-orders, PS-partsupp, P-part, S-supplier, C-customer).

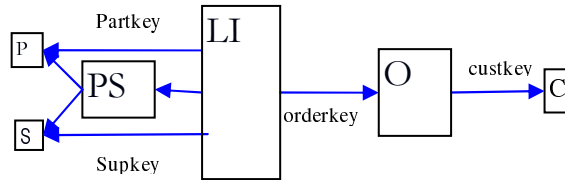


Fig. 3. TPC-H schema

For simplicity, we will portray relations PS, LI and O as facts and the remaining ones (P, S, C) as dimensions. For simplicity also we will assume that dimensions P, S and C are sufficiently small to deserve being replicated into each node. On the other hand, relations LI, PS and O should be horizontally partitioned into the nodes using a strategy that minimizes data exchange requirements while processing joins involving them. Joins involving only one of the horizontally-partitioned relations and any number of replicated relations require no data exchange, regardless of the partitioning scheme used. Therefore, we concentrate on minimizing data exchange needs for joins involving more than one partitioned relation (LI, O, PS) besides dimensions.

4. Relevant Costs in the NPDW

The main processing costs are the partitioning, repartitioning, data communication and local processing costs.

Partition Cost (PC), Repartitioning Cost (PC) - Partitioning a relation involves scanning the relation only once. The partitioning cost is monotonically increasing on the relation size. Since there can be two or more relations to be partitioned and they can be processed in parallel in two or more nodes, the partition delay PC for a given query is the largest partition cost among the nodes participating simultaneously. Repartitioning is similar to partitioning, but involves a fragment in each node instead of the whole relation. It is used to re-organize the partitioned relation, hashing on a different equi-join attribute. The fragments resulting from this repartitioning need to be redistributed to other nodes to process a hash-join.

Data Communication Cost (DC)- The data communication cost is monotonically increasing with the size of the data transferred and equal between any number of nodes. We assume a switched network (and broadcast capability).

Local Processing Cost - The local processing cost typically depends on whether the join is supported by fast access paths such as indexes, and the size of the relations participating in the join.

Merging Cost - The merging cost is related to applying a final query to the collected partial results at the merging node.

5. Placement and Processing in NPDW

Under NPDW all nodes participate in the computation of every query, so that typical queries must be “broken” into subset queries to distribute into all nodes and merging components to merge result sets coming from nodes. We define Global Processing as a processing “cycle” that involves sending data to nodes, processing in each node, collecting results from those nodes and merging them. Local Processing is the independent processing in a node. Query transformation produce processing and merging query components.

The global processing DCCM - Distribute-Compute-Collect-Merge – shown in Figure 4 involves parsing the query, transforming it into a “local query” to be processed in each node, running the local query independently in each node, collecting the corresponding partial results in the submitting node and merging them.

Set operators, typically UNION ALL, are useful for merging partial results from all nodes. Selection and aggregation query row sets will typically be processed independently by nodes and the partial results are merged as the union of partial result sets, followed by aggregation in the case of aggregation queries.

With this strategy, most query operations (selections, projections, aggregations) can be processed independently by each node, followed by a merge phase that receives independent results and merges them into a final result. Some operations, including most subqueries, require additional DCCM steps.

In this paper we are concerned with the placement and processing of joins in the context of NPDW. In the remaining of this paper we propose the solution to these issues and evaluate it.

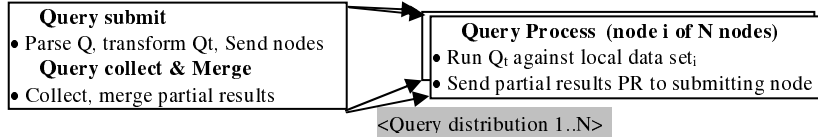


Fig. 4. The DCCM Procedure

5.1. Partitioning and Placement in NPDW

In order to determine the most appropriate partitioning strategy for a schema, knowledge of the query workload is required. In our example and experiments with TPC-H we considered the query set with equal weight for each query. In a practical situation knowledge of the query set would be accompanied by expected or historic frequency of execution of each query. More generically, it should be possible to know the query set and expected workload for a given data warehouse.

From the query workload we can determine a query workload join graph. A join graph $G_J = (V_J, E_J)$ is a graph where vertices V_J correspond to attributes R . A participating in equi-joins and the edges E_J depict the set of equi-joins between those attributes. We add weights to the edges of the join graph, which correspond to the number of occurrences of the corresponding equi-join in the query workload. A component is a set of connected edges in the join graph. Figure 5 shows the weighted join graph for TPC-H. In this join graph, replicated relations (P, S, C) are not represented because they pose no repartitioning issues for processing joins. Only those relations that are partitioned must be represented. Relation sizes are also relevant for the placement algorithm.



Fig. 5. Weighted Join Graph for TPC-H

This join graph can be produced either manually or automatically, as long as replicated relations are well identified. The next step is to decide which partitioning attributes should be chosen for each relation. The choice is based on the relation sizes and frequency values of the join graph. The most frequent equi-join component including the largest relation has top priority. In Figure 3 it is clear that LI and O (Orders) should be partitioned by orderkey (O), so that joins between these two relations do not require any repartitioning. But if LI is partitioned by orderkey (O), the other, less frequent equi-join in Figure 3 (LI with PS) will require repartitioning of LI. More generically, the algorithm is:

1. For the largest relation, find largest frequency component in the join graph;
2. Partition intervening relations by the corresponding equi-join attribute;

3. Delete partitioned relations from all edges in all components of the join graph, as their partitioning attribute is already decided;
4. If done, stop. Otherwise, go to 1.

This strategy can be effectively applied automatically to any schema and query set. It implies simply some knowledge of the query workload. In this paper we use TPC-H to exemplify the use of such strategy.

5.2. Join Processing in NPDW

Given the placement of relations, join processing is based on cost minimization and uses parallel hash-join when required, as discussed in [13]. The focus is on minimizing the repartitioning and communication costs. The complexity and relevance of the join processing strategy in such a system has led us to treat the subject in depth elsewhere. However, we describe the basic strategy in this section using examples.

Intuitively, the most selective joins should be processed first. For illustration purposes we do not take into account different selectivities in the following examples.

Assuming that LI and O are pre-partitioned by orderkey, a join involving relations LI and O (and any other number of replicated dimensions) in the TPC-H schema of Figure 3 is transformed into the union of the independent node joins $\bigcup_{\text{all nodes}} (F_{LI} \times_o F_O)$, where F_X represents a fragment of X.

A more complex join involving all partitioned relations and dimensions P, S and C of Figure 3 can be processed as $(P \times_{ps} (S \times_S F_{PS})) \times_{PS} (F_{LI} \times_o (F_O \times_c C))$, based in processing joins with the smallest relations first. This processing path requires a single repartitioning. As P, S and C are replicated in all nodes, each binary join involving them can be processed independently in each node, which accounts for $(F_O \times_c C)$ and $(P \times_{ps} (S \times_S F_{PS}))$. Then, $(F_{LI} \times_o (F_O \times_c C))$ can also be processed independently without repartitioning because the placement algorithm co-located O and LI fragments by hash-partitioning on orderkey (o). Finally, the last join requires repartitioning of the intermediate result $(F_{LI} \times_o (F_O \times_c C))$ by the (ps) key to join with $(P \times_{ps} (S \times_S F_{PS}))$.

6. Experiments

In this section we compare the performance of alternative placement and processing strategies in NPDW using typical TPC-H queries over the schema of Figure 3. For all of them, we considered a setup in which the smaller relations P, S, C were replicated. The alternatives evaluated are:

HWP (best strategy): The Hash-based workload partitioning strategy. It involves placement based on join-wise workload-based partitioning of large relations and join processing based on PHJ. The only extra overhead comes from repartitioning when needed. The placement algorithm tries to minimize such need;

H: Parallel Hash-Join with hash-partitioned placement based on primary keys. It involves placement based on primary-key partitioning and join processing based on PHJ. The extra overhead comes from repartitioning when the primary-key is not the equi-join attribute be-

tween partitioned relations. The objective is to compare the performance of placement with workload knowledge (HWP) to “blind” placement (H);

PRS: The PRS strategy [15] considering that only the largest relation (LD) is pre-partitioned by placement. This strategy involves replicating intermediate results from all but one of the large relations into all nodes (broadcast). The major overheads are replication and processing costs as each node has to process un-fragmented intermediate results for all except the pre-partitioned relation;

Round-robin PRS (RR-PRS): This strategy involves placement by round-robin partitioning of facts. Join processing is similar to PRS;

Round-robin Hash (RR-H): This strategy involves placement by round-robin partitioning of facts. Join processing with more than one partitioned fact is based on PHJ but involves repartitioning overhead for all partitioned facts.

The differences between the strategies were apparent in the experimental results when processing queries with multiple partitioned relations. PRS and RR-PRS strategies typically incurred high replication and join processing costs in those cases; The RR-H strategy incurred repartitioning overhead for all partitioned relations; The H strategy incurred repartitioning overhead for some of the intermediate results (joins including the LI relation); HWP had the best results because it uses PHJ as H and RR-H but chooses an initial workload-dependent placement that minimizes repartitioning needs. With this strategy, the parallel hash-join algorithm is more efficient because it incurs much smaller overheads.

The experimental evaluation was conducted by measuring the node and communication costs for the strategies assuming similar nodes on a 100Mbps switched network and the TPC-H decision support benchmark™ [17]. Four data sets were generated for these experiments corresponding to a 50GB TPC-H generation into 1, 5, 10, 25 nodes. Our experiments also involved transforming TPC-H queries to run against the NPDW.

6.1. Performance Results

First of all we analyze queries accessing a single fact relation, as for these queries all but one relation (the fact) is replicated to all nodes and all the strategies have similar results. Such queries account for half the TPC-H query set. Figure 6a shows the response time results for some of these queries with a nearly linear speedup with the number of nodes. Another large fraction of the TPC-H queries involve joining relations LI and O besides the replicated dimensions. Figure 6b shows the response times of HWP for some of those queries. Once again the speedup is near to linear, as only 1/N fragment of the facts has to be processed in each of N nodes.

Figure 7a shows response time results for these queries comparing HWP, H, RR and PRS for 10 nodes. As can be seen from the Figure, the relative performance of the strategies varies with the query. This is for two main reasons: the extra repartitioning and processing overhead of the strategies centers on different relations (H on LI, PRS on O, RR on LI and O) and we applied local joins and restrictions (selection operators) on the relations and dimensions before exchanging intermediate results between nodes in order to minimize extra overheads. In spite of this, the conclusion is that HWP was consistently better: on average it was 25% better than H and 46% bet-

ter than PRS (and RR-PRS) for these queries. The only cases in which the difference is smaller is when the slower strategies were able to obtain a very small intermediate result locally before incurring the large exchange and processing overhead. HWP is also better than the “blind” hash strategy (H) or RR-H, which require repartitioning before processing the join.

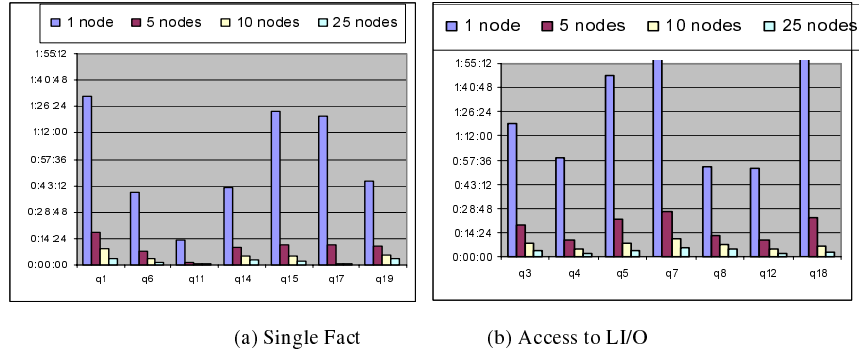


Fig. 6. Response Time Single Fact Queries

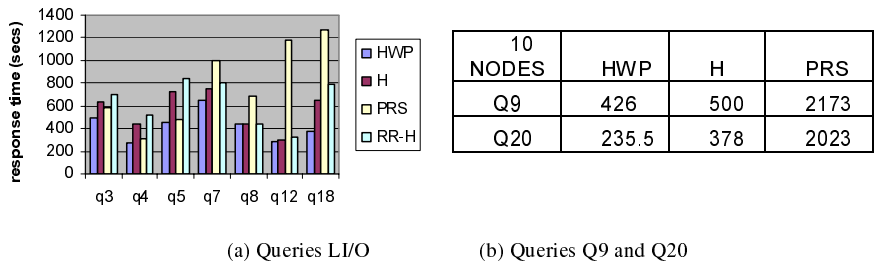


Fig. 7. Response Time Comparison, Queries L/I/O

Finally, queries joining more than two facts using different join attributes require repartitioning in HWP as well as H and replication of all but one fact in PRS/RR-PRS. TPC-H contains two such queries: Query Q20 joining Lineitem to Partsupp by ps and query Q9 joining all three relations Orders, Lineitem and Partsupp. Figure 7b shows the response times for these queries (secs). Our strategy was on average 26% better than H and 84% better than PRS/RR-PRS for these queries.

The PRS/RR-PRS overhead increases quickly with the number of large relations requiring replication not only due to larger communication overhead but also due to much larger node processing overheads.

7. Conclusions

We have proposed and studied experimentally a partitioning, placement and join processing solution to process join queries efficiently in a node-partitioned data

warehouse. The strategy is workload-based. Comparative evaluation against TPC-H has shown the advantages of the strategy and alternatives.

8. References

1. Apers, P. M. G.. Data allocation in distributed database systems. ACM Transactions on Database Systems, 13(3):263--304, September 1988.
2. P.A. Bernstein, N. Goodman, E. Wong, C. Reeve, and J.B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," ACM Trans. Data&se Systems, vol. 6, no. 4, pp. 602-625, Dec. 1981.
3. Chen, Hao, Chengwen Liu: An Efficient Algorithm for Processing Distributed Queries Using Partition Dependency. Int'l Conference on Parallel and Distributed Systems, ICPADS 2000: 339-346.
4. Copeland G. P., William Alexander, Ellen E. Boughter, Tom W. Keller: Data Placement In Bubba. SIGMOD Conference 1988: 99-108.
5. David J. DeWitt, Robert Gerber, Multiprocessor Hash-Based Join Algorithms. Proceedings of the 11th Conference on Very Large Databases, Morgan Kaufman pubs. Stockholm.
6. Hua, K. A. and Lee, C., "An Adaptive Data Placement Scheme for Parallel Database Computer Systems" Proc. VLDB Conf., Brisbane, Australia, 1990.
7. M. Kitsuregawa, H. Tanaka, and T. Motooka. Application of hash to database machine and its architecture. New Generation Computing, 1(1):66-74, 1983.
8. Liu, Chengwen and Hao Chen, "A Hash Partition Strategy for Distributed Query Processing", International Conference on Extending Database Technology (EDBT) 1996.
9. Liu, Chengwen, Hao Chen, Warren Krueger, "A Distributed Query Processing Strategy Using Placement Dependency", Proc. 12th Int'l Conf. on Data Eng, pp. 477-484, Feb. 1996.
10. Livny, Miron, Setrag Khoshafian, Haran Boral: "Multi-Disk Management Algorithms". In Proc. Of ACM SIGMETRICS 1987, pp69-77.
11. Rao, Jun. , Chun Zhang, Nimrod Megiddo, Guy M. Lohman: Automating physical database design in a parallel database. SIGMOD Conference 2002: 558-569.
12. Sacca, D. and Wiederhold, G.: Database Partitioning in a Cluster of Processors. ACM TODS, Vol. 10, No. 1, pp. 29-56, Mar 1985.
13. Shasha, Dennis and Wang, Tsong-Li: Optimizing Equijoin Queries In Distributed Databases Where Relations Are Hash Partitioned. ACM Transactions on Database System, Vol. 16, No. 2, pp. 279-308, June 1991.
14. Teradata Corporation. Database Computer System Concepts and Facilities. Document C02-0001-01, Teradata Corporation, Los Angeles, Oct. 1984.
15. Yu, C. T., Guh, K. C., Brill, D. and Chen, A. L. P.: Partition strategy for distributed query processing in fast local networks. IEEE Transactions on Software Engineering, Vol. 15, No. 6, pp. 780-793, June 1989.
16. Zhou S., M.H. Williams, "Data placement in parallel database systems," Parallel Database Techniques, IEEE Computer Society Press, 1997.
17. Transaction Processing Council Benchmarks, www.tpc.org.
18. Zilio, Daniel C., Anant Jhingran, Sriram Padmanabhan, Partitioning Key Selection for a Shared-Nothing Parallel Database System *IBM Research Report RC 19820 (87739) 11/10/94* ,T. J. Watson Research Center, Yorktown Heights, NY, October 1994.