Hash-based Placement and Processing for Efficient Node Partitioned Query-Intensive Databases

Pedro Furtado
DEI /CISUC, Universidade de Coimbra, Portugal
pnf@dei.uc.pt, http://eden.dei.uc.pt/~pnf

Abstract

This paper discusses efficient hash-partitioning using workload access patterns to place and process relations in a cluster or distributed query-intensive database environment. In such an environment, there is usually more than one partitioning alternative for each relation. We discuss a method and algorithm to determine the hash partitioning attributes and placement. Among the alternatives, our algorithm chooses a placement that reduces repartitioning overheads using expected or historical query workloads. The paper includes a simulation study showing how our strategy outperforms ad-hoc placement and previously proposed distributed database strategies.

1. Introduction

Relation partitioning schemes have been proposed to achieve a high degree of parallelism and improve response time in parallel and distributed databases. Two important related and intertwined issues are data allocation [1, 3, 5, 10, 17, 18] and query processing strategies [2, 4, 6, 7, 8, 11, 13, 14, 16]. Data allocation in parallel and distributed databases has been studied extensively in the context of generic database processing. In an early work [9] 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 overhead. [3] proposes a solution to data placement by variable partitioning. The degree of partitioning (nº of nodes over which to fragment a relation) is a function of the size and access frequency of the relation. Again experimental results in [3] show that partitioning increases throughput for short transactions but complex transactions involving several large joins result in reduced throughput with increased partitioning.

One of the crucial issues with partitioning in the presence of large 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. Strategies to process joins in such environments include partition and replicate PRS [15], the use of placement dependency [2, 7] and hash-partitioning strategies.

Hash-partitioned architectures characterize many successful parallel and distributed database systems, including the Teradata machine [12] as well as research prototypes. Parallel hash-join algorithms such as hybrid hash join or GRACE [16] consider dynamically partitioning and allocating intervening relation fragments into processors for fast join processing. These strategies typically allocate a hash range to each processor, which builds a hash table and hashed relation fragments are redirected to the corresponding processor. Algorithms and optimizations for parallel processing of multi-way joins are considered in [6] and [11]. In [6] the relations are assumed not fragmented and in [11] they are assumed fragmented but without any specific data placement strategy. In this paper we are concerned with performance benefits that can be obtained from a good data placement strategy in database environments such as data warehouses involving complex and time-consuming queries joining several relations, including many Gygabyte relations. We propose a hash-partitioned data placement and processing method over a node-partitioned architecture that reduces unnecessary repartitioning overheads. The strategy improves on previous algorithms and places a lot less burden on the network/interconnect speed and organization.

2. The Partitioning Issue

The basic partitioning problem is well described in [6]. Assuming all queries of the form Q={ target |

qualification}, where target is a list of projected attributes and qualification is a list of equi-joined attributes, let a query be Q={R1.A,R2.B | R1.A=R2.A Λ R2.B=R3.B}. Then not all of the three relations can be partitioned, since the first join predicate requires R2 be partitioned on A and the second join predicate requires that R2 be partitioned on B. We may choose to partition R1 and R2 on A and replicate R3 or to partition R2 and R3 on B and replicate R1. The problem is then to decide which set of relations to partition.

Relation	Tuples	Site 1	Site 2
R1	12000	R1	
R2	10000		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: $R_1 \text{ x}_A R_2 = (F_{11} \text{ x}_A R_2) \cup (F_{12} \text{ x}_A R_2)$ processed in parallel in sites 1 and 2.

Using hash-partitioning, 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 {Fij} if 1) $R_i=\bigcup_i F_{ij}$; 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 $R_1 \text{ x}_A R_2$ to be processed faster as $(F_{11} \text{ x}_A F_{21}) \cup (F_{12} \text{ x}_A F_{22})$, as fragments are results of hashing the relations such that $F_{11} \text{ x}_A F_{22}= \emptyset$ and $F_{12} \text{ x}_A F_{21}= \emptyset$. 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, although R_1 and R_2 can be partitioned simultaneously at each node.

If F_{11} and F_{21} are initially located at node 1 and F_{12} and F_{22} are initially located at node 2, then the join can proceed without any partitioning and communication costs (except to merge the partial results), as the necessary data is already at the processing sites. 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 the 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. Repartitioning is not especially costly and in fact is typically much faster than partitioning the whole relation at a single node.

The query processing issue in [6] is stated as follows: suppose $RRQ=\{R_1,R_2,\ldots,R_n\}$ is a set of relations referenced by a query Q and none is fragmented. A subset PR of these relations will be chosen to be hash-partitioned into d disjoint fragments on their join attributes. The fragments will be assigned to a set of d processing sites such that fragments with the same hash will be assigned the same site. The other relations WR=RR-PR are replicated at each processing site. At each site a subquery, which is the same as Q but referencing the fragments of relations in PR and all the other relations in WR, is executed. The answer to Q is the union of the results to the subqueries ate the d sites. The problem is to decide (1) the set of relations PR to be partitioned; (2) The copy of the relation to be used, if multiple copies exist; (3) the set of processing sites; (4) the number of fragments of the relations that are to be partitioned; (5) the size of each of the fragments to be produced, such that the response time (process and communicate) is minimized. In this paper we do not impose the restriction that relations are not fragmented initially, instead we full-partition most of them at a placement phase to obtain improved performance. This eliminates most of the issues (1) to (5) as in our strategy all sites/nodes process the query and all have fragments. Instead, we concentrate on reducing the query processing overheads by choosing a good placement strategy. For most of the discussion we assume homogeneous nodes, but the model is easily generalized to deal with heterogeneous nodes.

3. Cost Model

Our cost model is adapted from [6] and [11]. The main costs are the partitioning, repartitioning, data communication and local processing costs.

Partition and Repartitioning Cost (PC)– Partitioning a relation consists of retrieving the relation from secondary memory, dividing it into fragments by applying a hash function to a join attribute and assigning buffers for the data to send to other nodes (in our strategy each node, including itself, receive one or a few fragments). This involves scanning the relation only once. A simple cost model defines a partition cost that 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 Q is the largest partition cost among the nodes participating simultaneously.

Repartitioning Cost (PC) – Repartitioning is similar to partitioning, but involves a fragment in each node instead of the whole relation. It is used to (re)partition simultaneously all fragments to re-organize the hash-partitioned relation by a different 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 also assumed to be monotonically increasing with the size of the data transferred and equal between any number of nodes.

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. It is also assumed to be monotonically increasing with the size of the relations involved (based on the fact that a linear cost model is sufficiently robust to cover the usual join algorithms [6]).

Total Cost - The total cost of processing a subquery at a site is assumed to be a weighted sum of the partitioning and repartitioning delay PC, the communication cost DC and the local processing cost LC. As in [6] and [11] we define weighting parameters: α, which is used to denote the ratio of partitioning cost to local processing cost. The smaller the value of α, the smaller the partition cost relative to local processing cost. The second parameter is β, which is used to denote the ratio of transmission cost to local processing cost. The smaller the value of β, the smaller the transmission cost relative to the local processing cost. Such a model is sufficiently generic to be able to represent several alternative physical architectures.

4. The Node-Partitioned Query-Intensive Database

We have mentioned in the introduction that complex queries may involve significant amount of data exchange between nodes in a node-partitioned database. In this section we discuss some aspects of this kind of database environment using a data warehouse as a typical situation. A data warehouse is a repository of historic business-related data. It is periodically loaded from online transaction processing systems (OLTP) and used in a read-only, querying fashion. The importance of the parallel and distributed processing issues in data warehouses stems from the fact that they typically have some extremely large relations and a lot of heavy join processing requirements. Queries can take hours to execute if fast architectures are not adopted, centralized access can result in potential bottleneck and the heavy join processing poses performance issues in a node-partitioned architecture. Data placement and efficient join processing becomes a critical issue in such context.

A data warehouse is better organized as a set of multidimensional schemas, typically star schemas, in which central fact relations measuring quantities (e.g. sales) are linked to dimensions listing entity instances (e.g. one dimension for Products, another for Customers and one for Time). Facts determine for instance how many sales of each product were made for each customer at each recorded time period. Facts typically grow very large while dimensions are typically much smaller and the system is used for all types of analyzes – Online analytical processing OLAP. The TPC-H schema depicted in Figure 1 [19] used for performance benchmarking illustrates a data warehouse (not a pure star) and includes interesting access patterns of a typical data warehouse. It represents ordering and selling activity (LI-lineitem, O-orders, PS-partsupp, P-part, S-supplier, C-customer). Arrows represent references from one relation to the other (e.g. An Order Lineitem in LI has a reference "orderkey" to the corresponding Order in O). We have added a fact De (Delivery) to help illustrate later our partitioning algorithm.

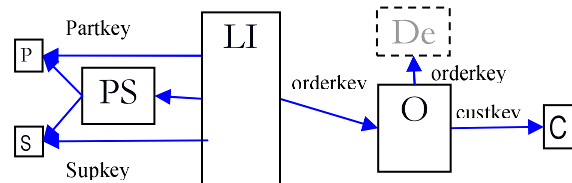


Figure 1 – TPC-H schema

For simplicity we will portray relations PS, LI and O as facts and the remaining ones (P,S,C) as dimensions. As usual in data warehouses, TPC-H facts are very big (LI>O>PS), while dimensions are much smaller. This is illustrated in the figure by the size of each relation.

Typical queries can be illustrated by the following TPC-H example computing sales for each nation and product brand:

Select Nation, Brand, sum(l_eprice *(1 - l_discount))
From orders, lineitem, supplier, region
Where Join Conditions and Row Conditions
group by Nation, Brand;

In contrast to OLTP systems, data warehouses are mostly read-only except for periodic loading moments and queries are typically join-intensive and time-consuming, frequently requiring full table scan access to large relations. The most expensive operations in OLAP processing typically involve full scanning large facts and joining several relations.

The Node-Partitioned Data Warehouse (NPDW) is an experimental system which is the focus of our work on the subject. It partitions the data warehouse into nodes and provides a middle layer that uses them transparently for fast query processing. The basic data placement involves hash-partitioning horizontally all except very small relations and placing them into the nodes. The advantage of the data placement step is to promote pre-partitioning of large relations to minimize on-the-fly (query processing) data partitioning and communication costs. By dividing the facts, NPDW completely parallelizes full

fact scans and joining activity. For instance, considering the TPC-H schema of figure 1, our strategy can replicate the dimensions C, S and P and partition the much larger relations PS, LI, O and De. A possible query processing strategy involves first joining each node fragment of a big fact with the set of small dimensions in parallel for all nodes. This step typically produces a much smaller intermediate result set. It is possible to do it in parallel without data exchange between nodes if dimensions are replicated in all nodes. Afterwards, if there are other partitioned relations, take the partitioned intermediate result of the previous join to process the next join with a partitioned relation in parallel in all nodes. So the goal of the strategy is to achieve a pre-partitioning of large relations that avoids much repartitioning overhead during join processing of large relations. In the next section we present the strategy.

Considering facts with size F_i, N nodes and the linear cost model presented in section 3, we can obtain a simple expression for the cost or repartitioning versus the cost of local processing without requiring repartitioning when the placement favors local processing. For simplicity, the following expressions consider only two facts. The fact fragment size is F_i/N. The join-processing cost for queries requiring the join between equi-partitioned facts with sizes F_{ij} such that $\sum_{j=1}^{f} F_{ij} = F_i$ and l dimensions di is:

$$Cost_{equipart} = \alpha \times \left(\frac{F_1}{N} + \frac{F_2}{N} + d_1 + \ldots + d_l \right) \quad (1)$$

The cost when facts are not equi-partitioned and considering a switched network is:

$$C_{Rep} = \left(\frac{IR}{N} - \frac{IR}{N^2} \right) \times \beta + \alpha \times \left(\frac{F_1}{N} + \frac{F_2}{N} + d_1 + \ldots + d_l \right) \quad (2)$$

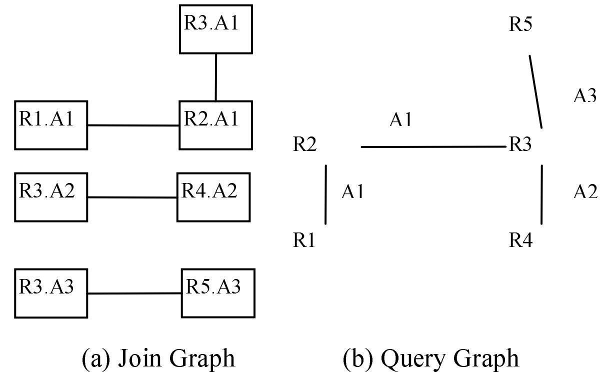
This expression includes the repartitioning overhead of an intermediate result IR (either the fact or the result of joining with some dimensions locally before repartitioning) and the local processing cost of 1/N of the facts. The increase in cost if (2) is required instead of (1) is therefore $\left(\frac{IR}{N} - \frac{IR}{N^2} \right) \times \beta$, which can be avoided many times if the facts are carefully placed.

5. Hash Pre-Partitioning (HPP) Algorithm

In this section we describe the data partitioning algorithm that determines a data placement resulting in reduced overhead during query processing. For a specific query we can consider a query and a join graph:

Definitions. $G_Q = (V_Q,E_Q)$ is a query graph of query Q, where V_Q are the relations intervening in the query and E_Q is the set of edges depicting the set of equi-joins between the relations in V_Q.

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. The following example is for Q= { R1.A2, R2.A4 | R1.A1=R2.A1 Λ R2.A1=R3.A1 Λ R3.A2=R4.A2 Λ R3.A3=R5.A3}:



(a) Join Graph (b) Query Graph

Figure 2 – Join and Query Graphs for the Example

The join graph consists of a set of connected components. We note that for an acyclic query Q, all nodes of a component of a join graph form a set of relations that can be joined without requiring repartitioning and redistribution.

The focus of our algorithm is to partition the whole relations in the join graph in a way that results in reduced repartitioning cost and redistribution requirements. In other words we answer the question: should R3 be partitioned by attribute A1, A2 or A3? In order to determine this, we must have some idea of a query workload instead of a single query. It can be a query workload prediction or statistics obtained from previous executions. For instance, considering the set of 22 queries of the TPC-H benchmark, it is possible to build a join graph for the query workload. In a real situation we would also be able to know or estimate the frequency of occurrence of each query to use in constructing a join graph with occurrence frequency information. The result of the algorithm can be used either for initial data placement or data reorganization.

Definition: The join graph of a query workload $G_w = (V_w,E_w,W)$ is a generalization of the join graph of a query: it is a graph where vertices V_w correspond to attributes R.A participating in equi-joins, the set of edges E_w is the set of equi-joins between those attributes, components are connected edges corresponding to join patterns occurring and W is the set of weights given to

components. A weight is the frequency of occurrence of the corresponding component on the query workload (either a percentage or number of occurrences).

Figure 3 shows an example query workload join graph with four components (c1, c2, c3, c4) with (3 ,10 ,8 ,6) occurrences respectively.

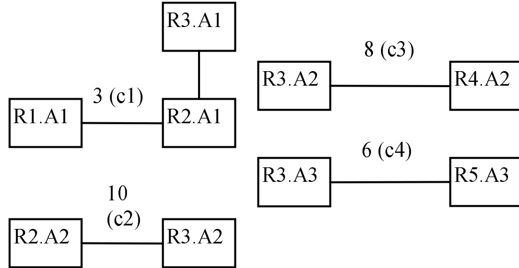


Figure 3 – Join Graph for the Query Workload

Consider the set of relation sizes SW for the query workload join graph and the set of component weights W. The HPP algorithm simply chooses the join attribute of the most frequent join component for each relation. First of all, the algorithm builds a join graph including only large relations, as small ones will be fully replicated and joins with these handled locally. Consider the initial set of all components C = $\{c_1,\ldots,c_n\}$ where $c_i=\{e_x,\ldots,e_y, A_k, w_i\}$ is a set of connected edges with equi-join attribute A_k and weight w_i. Each edge is made of a tuple $e_i=(R_u, R_p, A_k, S_u, S_p)$, where S_u and S_p are the relation sizes. The determination of the partitioning attributes is obtained now by the following simple steps:

Step 0 – Order the list of components by descending order of weight;
Step 1 – For each relation R
Pick the component with highest weight containing the relation
The relation R is to be partitioned by the components' equi-join attribute;

If $RR_Q=\{R_1,R_2,\ldots,R_n\}$ is the set of relations in the schema, the algorithm determines a set PR of pairs (R_i,A_k) describing how each relation in the set should be hash-partitioned. Then the placement strategy divides each relation into all nodes. If there are *d* nodes, a hash function can be applied to determine *d* disjoint fragments to be distributed into the nodes. In practice, a multiple of *d* fragments should be used so that each node can have more than one fragment for dynamic load balancing. Dynamic load balancing, which is out of the scope of this paper, is necessary to accommodate placement, selectivity, redistribution and join skews. We have also mentioned that, for simplicity, we assume homogeneous nodes. The only relevant modification to accommodate heterogeneous nodes would be to determine varied fragment sizes whose ratios would be proportional to the performance ratios between nodes.

Applying the algorithm to the example of figure 3, the resulting partitioning is (R1.A1, R2.A2, R3.A2, R4.A2, R5.A3). Component c1 join will require repartitioning R2 and R3 but not R1. Joins of the most frequent components c2 and c3 will not require repartitioning. Finally, c4 joins will require repartitioning relation R3.

4. Performance of the Algorithm

In this section we engage in a simulation to evaluate the strategy, comparing it to both previous algorithms in [21, 6] and basic hash-partitioning. We compare the following algorithms by simulation:

1. Our optimized hash pre-partitioning algorithm (HPP);
2. Runtime Hash-partitioning RTHP (e.g. [6]) - Relations are not partitioned and reside in one node. Some relations have a copy in other nodes as well. An algorithm chooses partitioning and processing sites such that relations from one component of the join graph are hash-partitioned and routed to the processing nodes for join processing, while the others are replicated and fully processed;
3. Key Hash-partitioning KHP – Relations are horizontally partitioned "blindly" on their primary keys. The most relevant difference to HPP is that the partitioning does not take join costs into account. Therefore, join processing frequently requires repartitioning relations when the join attribute is not the primary key;

Experiments are setup as follows. The dimension of the initial data distribution for the experiment is n_i x m_j, where n_i, is the number of relations ($2<=n_i<=10$), and m_j is the number of nodes ($2<=m_j<=10$). We have generated relations and corresponding data distributions with sizes from 10G to 100GB and small fully replicated ones with sizes 10M to 100M. Typical join patterns were also generated where small relations represented between 30% and 90% of the mix. Since linear cost models are used for relation partitioning, data communication and local processing, we used the cost model parameters to characterize the experimental environment: α to denote the ratio of partitioning cost to local processing cost. The smaller the value of α, the smaller the partition cost relative to local processing cost; β to denote the ratio of transmission cost to local processing cost. The smaller the value of β, the smaller the transmission cost relative to the local processing cost.

Number of relations	Number of nodes								
	2	3	4	5	6	7	8	9	10
2	19.0%	28.9%	34.5%	40.7%	44.6%	48.1%	49.9%	53.1%	56.1%
3	25.7%	38.3%	44.6%	52.3%	57.4%	60.7%	63.7%	66.0%	67.9%
4	31.0%	47.7%	52.0%	59.4%	64.8%	68.5%	71.3%	73.5%	75.3%
5	34.2%	51.1%	56.7%	63.8%	68.9%	71.6%	74.3%	76.4%	78.1%
6	36.0%	52.7%	58.5%	65.4%	70.0%	72.7%	75.4%	77.4%	79.1%
7	37.7%	53.9%	59.9%	66.7%	70.8%	73.5%	76.1%	78.2%	79.8%
8	38.7%	54.5%	60.9%	67.6%	71.6%	74.4%	76.9%	78.9%	80.5%
9	39.4%	55.1%	61.9%	68.5%	72.5%	75.0%	77.5%	79.5%	81.1%
10	40.2%	55.9%	63.7%	70.1%	74.0%	76.3%	78.8%	80.7%	82.2%

Figure 4 – Comparison KHP vs RTHP, α=β=0.2

The two parameters are varied from 0.1 to 1 with increment of 0.1. We arbitrarily generated 1000 sets of query graphs and frequencies for each given combination of n (the number of referenced relations) and m (the number of nodes). We show representative sets of the experimental results. The results are shown as "improvement factor". Given the average response time of the query sets over A1 and A2, the improvement factor of A2 over A1 is (A1-A2)/A1.

We first look at the impact of the number of relations and the number of sites on the improvement factor. The RTHP algorithm assumes relations are not partitioned initially. The relations in a join graph component of the query must be partitioned and relations must be distributed into the processing sites for each query. In contrast, the two other strategies compared in this simulation (KHP and HPP) hash-partition relations in a placement phase (except small relations). This yields large performance advantages, not only because some relations do not incur large repartitioning cost but also because repartitioning and redistributing is done in parallel by all nodes containing the partitions. Figure 4 shows the improvement of KHP relative to RTHP in the same context as the previous experiment. The larger the number of nodes, the larger the parallel processing advantage obtained by having pre-partitioned everything.

Algorithms KHP and HPP are much more similar in nature. Both hash-partition data into all nodes and the main difference is that HPP is able to avoid repartitioning costs as large as those incurred by KHP.

Figure 5 shows results of comparing HPP to KHP. HPP benefits from choosing the most appropriate hash partitioning attributes, reducing the repartitioning costs significantly.

Number of relations	Number of nodes				
	2	4	6	8	10
2	14.81	17.04	17.75	18.09	18.30
4	15.28	17.68	18.42	18.79	19.01
6	15.44	17.88	18.64	19.01	19.24
8	15.52	17.98	18.75	19.13	19.35
10	15.56	18.04	18.82	19.19	19.42

Figure 5 – Comparison HPP vs KHP, α=β=0.2 (%)

As expected, HPP is more efficient than KHP. The exact improvement depends largely on the data set characteristics. The improvement factor increases with the number of nodes and with the number of relations, although at a decreasing improvement rate The increase in improvement factor is due to the larger repartitioning and redistribution overheads of KHP.

The next experiment, shown in Figure 6, evaluates the effect of transmission speed (β) and repartitioning speed (α) (relative to local processing speed) on the comparison between KHP and HPP. In each case, one of the parameters is fixed at 0.1 and the other varied between 0.1 and 2. The most important conclusion from these results is that HPP is much more immune to the network/interconnect speed and organization than KHP. Both the network/interconnection speed and the repartitioning speed have a much larger impact on KHP than on HPP. This is as expected because KHP incurs in much larger repartitioning and redistribution costs than HPP. Therefore the advantage of HPP over KHP is much larger in the presence of slow networks/interconnects.

β	Number of sites				
	2	4	6	8	10
0.1	12.7%	14.1%	14.5%	14.7%	14.9%
0.25	16.6%	19.5%	20.3%	20.8%	21.0%
0.5	22.0%	26.3%	27.6%	28.2%	28.6%
0.75	26.3%	31.5%	32.9%	33.6%	34.0%
1	29.9%	35.4%	36.9%	37.6%	38.0%
2	39.5%	45.1%	46.5%	47.1%	47.5%

α	Number of sites				
	2	4	6	8	10
0.1	11.6%	14.3%	15.2%	15.6%	15.8%
0.25	17.1%	19.4%	20.1%	20.5%	20.7%
0.5	24.7%	26.4%	27.0%	27.2%	27.4%
0.75	30.7%	32.1%	32.5%	32.7%	32.8%
1	35.7%	36.7%	37.0%	37.2%	37.3%

Figure 6 – Comparison HPP vs KHP with varying α and β for n=5 relations

5. Conclusions

In this paper we have proposed a hash-partitioning data placement method to reduce repartitioning overhead during query workload execution in node-partitioned databases. The strategy is useful in query-intensive node partitioned databases, typically data warehouses. For a given query workload, our algorithm first determines the join patterns occurring in the workload. Using this information it partitions the relations according to their most frequent joins.

Simulated experiments were conducted to evaluate the performance of the algorithm. As expected, the strategy outperforms previously proposed strategies that partition on-the-fly and only partially (one relation for PRS, one join component for RTHP). We also show improvement results when comparing the strategy to a blind hash-pre-partitioning strategy that doesn't take into account join patterns at all. We show that, by using a careful hash pre-partitioning, the strategy reduces drastically the overall transmission and repartition overheads, therefore being much more immune to interconnect/network organization and speed than the other strategies.

6. References

[1] Apers, P. M. G.. Data allocation in distributed database systems. ACM Transactions on Database Systems, 13(3):263--304, September 1988.

[2] 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.

[3] Copeland G. P., William Alexander, Ellen E. Boughter, Tom W. Keller: Data Placement In Bubba. SIGMOD Conference 1988: 99-108.

[4] Epstein. R., M. Stonebraker, and E. Wong. Distributed query processing in a relational database system. In Proceedings of the ACM-SIGMOD International Conference on Management of Data, pages 169--180, 1978.

[5] Hua, K. A. and Lee, C., "An Adaptive Data Placement Scheme for Parallel Database Computer Systems" Proc. VLDB Conf., Brisbane, Australia, 1990.

[6] Liu, Chengwen and Hao Chen, "A Hash Partition Strategy for Distributed Query Processing", International Conference on Extending Database Technology (EDBT) 1996.

[7] 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.

[8] Liu, Chengwen and Hao Chen, "A Heuristic Algorithm for Partition Strategy in Distributed Query Processing". In ACM Selected Areas in Cryptography SAC 1996, Feb 1996.

[9] Livny, Miron, Setrag Khoshafian, Haran Boral: "Multi-Disk Management Algorithms". In Procs. Of ACM SIGMETRICS 1987, pp69-77.

[10] Sacca, D. and Wiederhold, G.: Database Partitioning in a Cluster of Processors. ACM TODS, Vol. 10, No. 1, pp. 29-56, Mar 1985.

[11] 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.

[12] Teradata Corporation. Database Computer System Concepts and Facilities. Document C02-0001-01, Teradata Corporation, Los Angeles, Oct. 1984.

[13] Clement T. Yu, C. C. Chang: Distributed Query Processing. ACM Computing Surveys 16(4): 399-433 (1984).

[14] Yu, Clement T., Keh-Chang Guh, Weining Zhang, Marjorie Templeton, David Brill, Arbee L. P. Chen: Algorithms to Process Distributed Queries in Fast Local Networks. IEEE Transactions on Computers 36(10): 1153-1164 (1987).

[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] Yu, Clement T. and Weiyi Meng. “Principles of Database Query Processing for Advanced Applications”, 1998 Eds. Morgan Kaufmann ISBN 1-55860-434-0.

[17] Wong, E. and Katz, R. H.: Distributing a Database for Parallelism. In Proc. 1983 ACM SIGMOD Conference, pp 23-29, San Jose, CA, May 1983.

[18] Zhou S., M.H. Williams, "Data placement in parallel database systems," Parallel Database Techniques, IEEE Computer Society Press, 1997.

[19] Transaction Processing Council Benchmarks, www.tpc.org.