

Experimental Evidence on Partitioning in Parallel Data Warehouses

Pedro Furtado
DEI/CISUC – Univ. Coimbra
Polo II
Coimbra, Portugal
+351239790000
pnf@dei.uc.pt

ABSTRACT

Parallelism can be used for major performance improvement in large Data warehouses (DW) with performance and scalability challenges. A simple low-cost shared-nothing architecture with horizontally fully-partitioned facts can be used to speedup response time of the data warehouse significantly. However, extra overheads related to processing large replicated relations and repartitioning requirements between nodes can significantly degrade speedup performance for many query patterns if special care is not taken during placement to minimize such overheads. In this paper we show these problems experimentally with the help of the performance evaluation benchmark TPC-H and identify simple modifications that can minimize such undesirable extra overheads. We analyze experimentally a simple and easy-to-apply partitioning and placement decision that achieves good performance improvement results.

Categories and Subject Descriptors

H.2.4 [Systems]: Parallel and Distributed Databases - *retrieval models, search process*

General Terms

Algorithms, Management, Performance, Experimentation.

Keywords

Parallel, Data Warehouse, Performance.

1. INTRODUCTION

Large data warehouses (DW), as repositories of historical data for OLAP and data mining analysis, have to deal efficiently with Giga or Terabytes of data while users require fast answers. Performance becomes a major issue in such systems.

Although structures such as materialized views and specialized indexes improve response time for predicted queries, parallel processing can be used alone or in conjunction to offer a major boost in such systems and guarantee speedup and scale-up. Furthermore, a simple cluster of low-cost nodes can be used to support the system.

When using such an architecture, one must decide how to partition or cluster relations into nodes. There is one major advantage in data warehouses when considering data placement strategies: they are mostly read-only environments with only periodic loads, ensuring that concurrency consistency issues are minor when compared to full-blown transactional systems. Considering a star schema with large facts, we used a simple round-robin full fact-partitioning and dimension replication strategy from our previous work as a basic placement strategy. The analysis of that strategy reveals that although it can handle with linear speedup a simple star with small dimensions, it cannot handle satisfactorily more complex schemas or queries accessing more than one fact. We are interested in obtaining a generic solution to process efficiently any query in the Node Partitioned Data Warehouse (NPDW). This involves considering any data warehouse schema and also less de-normalized schemas. In this paper we engage in experimental analysis of performance issues encountered when applying the basic placement strategy in order to identify the major bottlenecks. The TPC-H decision support benchmark [21] is used as the basis for our analysis and proposal because it is not a “pure” multidimensional star schema, but nevertheless is a very plausible historical record of a company business activity and therefore a plausible data warehouse. Bottlenecks are identified and we propose and analyze experimentally an improved partitioning and placement strategy to obtain an efficient Node Partitioned Data Warehouse system. The paper is organized as follows: section 2 briefly reviews related work. Section 3 describes the basic placement strategy and section 4 analyzes results from applying the basic placement strategy to TPC-H. Section 5 reviews partitioning and cost issues that must be taken into account to minimize the parallel join overheads. In section 6 we propose a modified partitioning and placement strategy and evaluate it in section 7. Section 8 contains concluding remarks.

2. RELATED WORK

Most performance issues related to parallel processing in databases become relevant as well when we implement a data warehouse in such an environment, as a data warehouse is a

specialized database. 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, 6, 12, 19, 20] and query processing strategies [2, 4, 7, 8, 9, 13, 15, 16, 18]. Data allocation in parallel and distributed databases has been studied extensively in the context of generic database processing. 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 overhead. [4] proposes a solution to data placement by variable partitioning. 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.

The most promising solutions to extra join overheads that characterize many successful parallel and distributed database systems in shared-nothing environments involve hash-partitioning large relations to minimize data exchange requirements. These include the Teradata machine [14] as well as research prototypes. Parallel hash-join algorithms such as hybrid hash join or GRACE [18] 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. Placement dependency [8] is another successful relation fragments co-location strategy.

Algorithms and optimizations for parallel processing of multi-way joins are considered in [7] and [13]. In [7] the relations are assumed not fragmented and in [12] they are assumed fragmented but without any specific data placement strategy. Automated partitioning and placement strategies were described in [11] for generic databases and in [22], which requires tight integration with the query optimizer to evaluate costs of alternative allocations. Neither one takes into consideration the specificity of the fact plus dimensions structure of data warehouses. Our work is related to these ones but focuses on a simple schema-based allocation strategy targeted at data warehouses and on the experimental evaluation of speedup that can be achieved using simple placement rules. It also extends our previous work on fact partitioning in Node-Partitioned Data Warehouses [5] by looking at all the join-related performance bottlenecks instead of only facts. We gather experimental evidence (TPC-H) of the factors that should be taken into consideration for a simple and easy-to-apply partitioning and placement strategy to yield almost linear speedup in NPDW.

3. BASIC PLACEMENT STRATEGY

A partitioned Data Warehouse must divide the DW into a set of nodes and provide a query processing middle layer that uses those nodes for fast query processing. Figure 1 shows a star schema with a fact and associated dimensions. The basic placement strategy fully partitions the largest, heaviest processing elements in the DW – fact tables - horizontally into nodes using a round-

robin or random distribution, while the much smaller dimensions, typical targets of join processing, are fully replicated into the nodes, as shown in Figure 1 for node i. Likewise, auxiliary structures such as indexes are replicated or partitioned depending on being associated with replicated dimensions or partitioned facts.

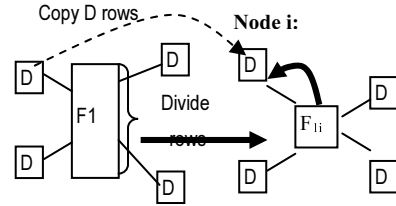


Figure 1 – Star Partitioning

The objective of this design is to allow nodes to process their part of the query independently and therefore achieve a speedup that is expected to be near to linear with the number of nodes. By decreasing the amount of data that needs to be exchanged, this placement reduces extra overheads. Given a typical simple OLAP query formulated as (OP is an aggregation operator such as SUM, COUNT):

```
OP(...)
JOIN (F, D1, ..., Dn)
GROUP (G1,..Gm);
```

Each node needs to apply exactly the same initial query on its partial data and the results are merged by applying the same query again at the merging node with the partial results coming from the processing nodes. Figure 2 illustrates this process.

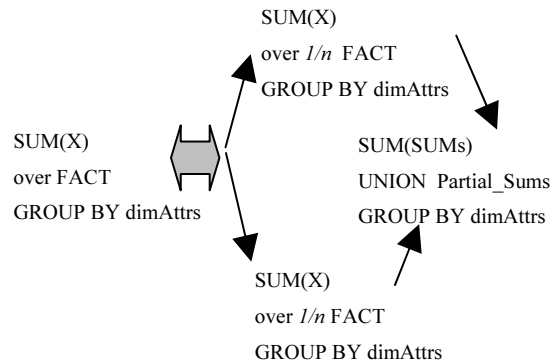


Figure 2 – Typical OLAP Processing in NPDW

The independent execution of partial joins by nodes is supported by the fact that all but one of the relations in the star (that is, all dimensions) are replicated into all nodes. Considering a fully-partitioned relation R_i and all the remaining ones replicated into all nodes, the relevant join property is:

$$R_1 \triangleright \triangleleft_{A_1} R_2 \triangleright \triangleleft_{A_2} R_i \triangleright \triangleleft_{A_i} \dots \triangleright \triangleleft_{A_n} R_n = \bigcup_{j=1}^{all_nodes} R_1 \triangleright \triangleleft_{A_1} R_2 \triangleright \triangleleft_{A_2} R_{ij} \triangleright \triangleleft_{A_i} \dots \triangleright \triangleleft_{A_n} R_n \quad (1)$$

It is easy to infer the processing for other queries. In particular, aggregation queries with aggregation functions other than SUM

can be handled by computing a subset of the primitives - Linear Sum = LS = $\text{SUM}(x_i)$; Squared Sum = SS = $\text{SUM}(x_i^2)$; N° of Elements = N; Maximum = MAX = $\text{MAX}(x_i)$; Minimum = MIN = $\text{MIN}(x_i)$ - as described in [10]. The typical query processing cycle is also shown in Figure 3.

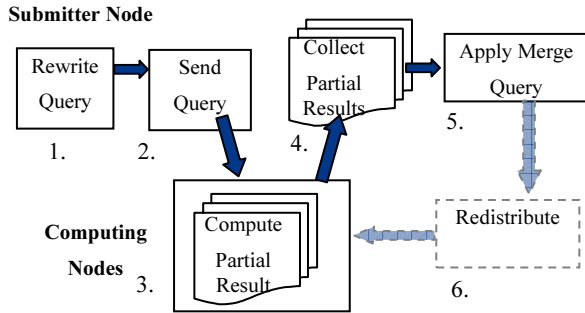


Figure 3 – Query Processing Steps in NPDW

Although it is out of the scope of this paper to fully describe the processing of all possible query patterns, it is relevant to our discussion to mention that the processing of subqueries frequently involves additional cycling over steps 3 to 5, including an additional redistribution step 6 that returns merged subquery results to nodes for further processing of the outer query.

4. ANALYZING NPDW

We are interested in obtaining a generic solution to process efficiently any query in a Node Partitioned Data Warehouse (NPDW), considering more complex and also less de-normalized schemas as well as the “pure star schema”. Figure 4 summarizes the TPC-H schema, which represents ordering and selling activity (LI-lineitem, O-orders, PS-partsupp, P-part, S-supplier, C-customer). There are two additional very small relations – NATION and REGION – not depicted in the Figure. It is not a “pure” multidimensional star schema as the one shown in Figure 1 but nevertheless is a plausible historical record of the company business activity. The TPC-H query set includes frequent joins between two or more large relations.

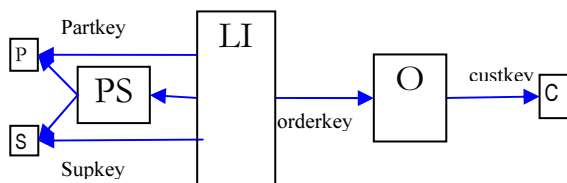
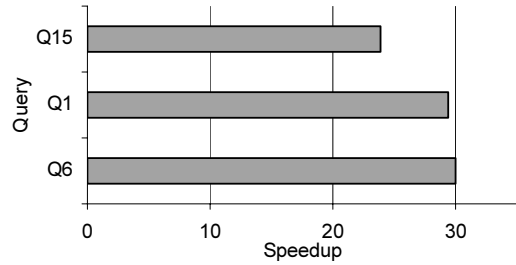


Figure 4 – TPC-H schema

The basic node partitioning strategy delineated before determines that dimensions are identified and replicated into all nodes (e.g. P, S, C and O) and facts are fully partitioned (PS and LI). As in Figure 1, joins involving only one of the horizontally-partitioned relations and any number of replicated relations require no data exchange besides merging partial results, regardless of the partitioning scheme used. Joins between LI and PS incur in repartitioning overheads.

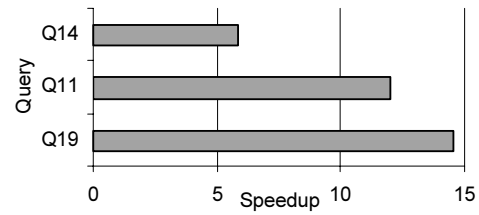
Figure 5 shows our experimental speedup results considering 25 nodes and the basic TPC-H configuration described before. For

these experiments we measured processing and data exchange time on a node with Intel Pentium III 866 MHz CPU, three IDE hard disk s(2x40 GB, 1x80GB) and 512 MB of RAM, running Windows XP professional and Oracle 9i (9.2) DBMS and a 100Mbps switched network.

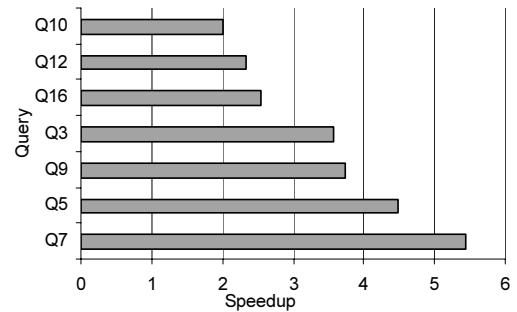


(a)

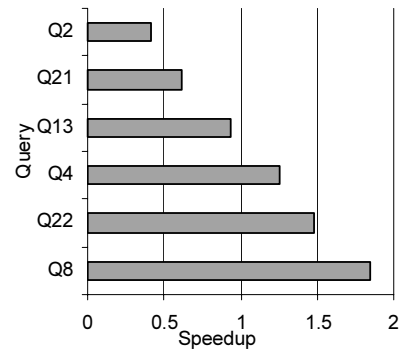
Largest Speedup (25-30)



(b) Speedup (6-15)



(c) Speedup (2-5.5)



(d) Speedup (0.4-2)

Figure 5 – TPC-H Speedup on Basic Placement

Given that these experiments concern 25 nodes, a desirable linear speedup would be about 25. The results show that of 19 queries only a few (3) queries were able to obtain those levels of speedup and most achieved very low speedup. In fact, these results can be explained if we look at the relations involved in the queries, as shown in Figure 6(a) and the relation sizes in Figure 6(b), including fragment sizes when LI and PS are fully partitioned.

	Fact	Dim		Fact	Dim
Q6	LI	-	Q19	LI	P
Q1	LI	-	Q11	PS	S
Q15	LI	S	Q14	LI	P
	Fact	Dim		Fact	Dim
Q7	LI	O C S	Q8	LI	O P C S
Q5	LI	O C S	Q22		OC
Q9	LI PS	O P S	Q4	LI	O
Q3	LI	O C	Q13		OC
Q16	PS	P S	Q21	LI	O S
Q12	LI	O	Q2	PS	P S
Q10	LI	O C			

(a) Fact and Dimensions Accessed

	GB
Lineitem (LI)	7.8/0.312
Partsupp (PS)	0.75/0.03
Orders (O)	1.8
Part (P)	0.2
Supplier (S)	0.01
Customer (C)	0.15

(b) Approximate Relation Sizes (10GB)

Figure 6 – Relations Accessed by Queries (TPC-H)

From Figure 6(b) we can see that, before partitioning, organizing relation sizes in descending order yields the ordered set (LI, O, PS, P, C, S). After replicating relations (O, P, S, C) and partitioning into 25 nodes relations (LI, PS), node fragment size ordering becomes (O, LI, P, C, PS, S).

Queries achieving the largest speedups (25-35) access only the fully-partitioned Lineitem fact (LI), one of them in conjunction with the smallest dimension of all (S). These speedups result from each node having to deal with only 1/25 of the data (LI). Although query Q15 also joins with a full S relation, S is nevertheless small. On the other hand, most queries with the worst speedups (0.4-2 and 2-5.5) join with large replicated dimensions (O, P or C). The placement strategy is not adequate to deal with this schema and query workload because, although the facts are partitioned, reasonably large relations such as O and P are replicated into all nodes. From Figure 6(b) we can see that the size of O, P and even C become of the same order of magnitude as the partitioned fragments of LI. We did not take sufficient advantage of the 25 nodes because we deal with full dimensions. We can go a bit further in this analysis by comparing the low speedup queries with the medium speedup range (6-15) of Figure

5(b). The relevant pattern is that low speedup queries involving the LI relation also involve O, while medium speedup queries involving LI also involve P. The reason for this is that P is much smaller than O and also smaller than LI fragments. The fact that those medium speedup queries have to join LI fragments with full P degrades the speedup but not as much as joining LI to O. A similar pattern can be seen for queries involving relation PS, whose fragments are very small. Queries joining PS with P have very small speedup because P is comparatively large, while query Q11 joining PS and S achieves medium speedup because S is reasonably small compared to PS. We have inspected informally the processing times of some queries to verify that indeed these join size factors were responsible for the relevant differences observed.

5. PARTITIONING ISSUE AND COSTS

In this section we review the relevant partitioning issue that must be handled in order to improve the speedup of NPDW over the basic strategy. Then we review relevant cost factors to NPDW.

5.1. Partitioning Issue

In order to improve the parallelism results significantly, we must partition most relations in NPDW. However, this invalidates expression (1) which allowed joins to be processed independently and in parallel by all nodes and then merged. Instead, repartitioning may become necessary to process many of the joins between partitioned relations. For this reason we discuss in this section the partitioning issue.

The basic partitioning problem is well described in [7]. 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 $Q = \{ R_1.A, R_2.B \mid R_1.A = R_2.A \wedge R_2.B = R_3.B \}$. Then not all of the three relations can be partitioned, since the first join predicate requires 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
R_1	12000	R_1	
R_2	10000		R_2

The partition and replicate strategy (PRS) [17] 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.

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 $\{F_{ij}\}$ if 1) $R_i = \bigcup_j F_{ij}$; 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, 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.

5.2. Relevant Costs in the NPDW

Given that most relations become partitioned, the main processing costs incurred in NPDW are the partitioning, repartitioning, data communication and local processing costs.

Partitioning 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. This involves scanning the relation only once. The partition 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 Cost (RC) – 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.

Local Processing Cost (LC) - 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 (MC)- The merging cost is related to applying a final query to the collected partial results at the merging node.

As in [13] we define weighting parameters: β , which is a partitioning cost weight and α local processing weight, so that β/α denotes the ratio of partitioning cost to local processing cost (e.g. ~ 2 in [15]). Considering large relations with size R_i , N nodes and the linear cost model described above, we can obtain a simple

expression for the cost of repartitioning versus the cost of local processing without requiring repartitioning when the placement favors local processing. For simplicity, the following expressions consider only two large relations. The fragment size is R_i/N . The join-processing cost for queries requiring the join between equi-partitioned large relations and small relations r_i is:

$$\text{Cost}_{\text{equipart}} = \alpha \times \left(\frac{R_1}{N} + \frac{R_2}{N} + r_1 + \dots + r_l \right) \quad (2)$$

The cost when large relations are not equi-partitioned on a switched network includes repartitioning and local processing cost factors with corresponding weights as shown in (3). The IR symbol in the repartitioning cost factor is an intermediate result from doing independently at all nodes a locally-processable part of the joins (those involving replicated and equi-partitioned relations). The IR must then be repartitioned. The value IR/N is the fraction of the IR that is at each node. About $1/N$ of that fraction ($1/N \times IR/N$) has the correct hash-value for the node, therefore requiring no repartitioning.

$$\text{CRep} = \left(\frac{IR}{N} - \frac{IR}{N^2} \right) \times \beta + \alpha \times \left(\frac{R_1}{N} + \frac{R_2}{N} + r_1 + \dots + r_l \right) \quad (3)$$

The increase in cost of (2) over (1) is therefore $\left(\frac{IR}{N} - \frac{IR}{N^2} \right) \times \beta$, which can frequently be avoided if the relations are carefully placed.

6. PROPOSED MODIFICATIONS TO NPDW

From the discussion of the previous section, it is clear that large relations should be partitioned in NPDW, regardless of whether they are considered facts or dimensions. A small relation in this perspective is one whose contribution to a join processing time cost is expected to be reasonably small either in absolute terms or in comparison with fragment sizes. The placement strategy described next can easily be automated:

Dimensions: Non-small dimensions can simply be hash-partitioned by their primary key. This is because that attribute is expected to be the only one used in every equi-join with facts. The references in Figure 1 from fact to dimensions correspond to foreign keys on the fact referencing those equi-join primary keys of the dimensions.

Facts: The objective is to find the hash-partitioning attribute that minimizes repartitioning costs. A reasonable approximation to this objective is to determine the most frequent equi-join attribute used by the relation. To do this, the placement strategy looks at the frequency of access to other partitioned relations and chooses the most frequent equi-join attribute with those relations as the partitioning attribute. By co-locating relation fragments that are frequent equi-join targets, this simple strategy minimizes repartitioning requirements.

Let us apply this placement strategy to TPC-H schema. We arbitrated “small” as less than 200MB for TPC-H 10 GB and ≤ 1 GB for TPC-H 50GB, so that dimensions C and S are considered “small” and replicated. This assumption is enough for us to show the advantages of the strategy, but we point out that smaller thresholds allow further speedup in systems with many nodes.

From Figure 6(b) we can see that with our proposed thresholds S and C are replicated and P is partitioned by P key. The O relation, if considered a dimension, is immediately key-partitioned (orderkey attribute). This relation could also be considered a fact because it links to dimension C, but the resulting partitioning would be the same if it were considered a fact.

Finally, facts should be partitioned according to the most frequent join. From Figure 6(a) we can see that fact LI is most frequently joined to O and therefore should be partitioned by O key. Fact PS is partitioned by P key.

- SUPPLIER – replicated;
- CUSTOMER – replicated;
- PART – partkey;
- LINEITEM – orderkey;
- ORDER – orderkey;
- PS – partkey;

This partitioning result in two sets of equi-partitioned relations (LI, O) and (PS, P) and a set of replicated dimensions (S, C). Joins between relations within each of these sets can be done without repartitioning and joins with any of the replicated relations can also be done without any repartitioning. Repartitioning becomes necessary when elements from the first and second sets above are joined.

7. EXPERIMENTS WITH THE MODIFIED SCHEMA

In this section we show results of our experiments on the TPC-H schema (50GB) and query set using the new partitioning and placement scheme and compare with the results of the old placement scheme. For these experiments we measured processing and data exchange time on a node with Intel Pentium III 866 MHz CPU, three IDE hard disk s(2x40 GB, 1x80GB) and 512 MB of RAM, running Windows XP professional and Oracle 9i (9.2) DBMS and a 100Mbps switched network. These experiments were also based on measuring the local processing performance over a single node and data exchange costs between nodes.

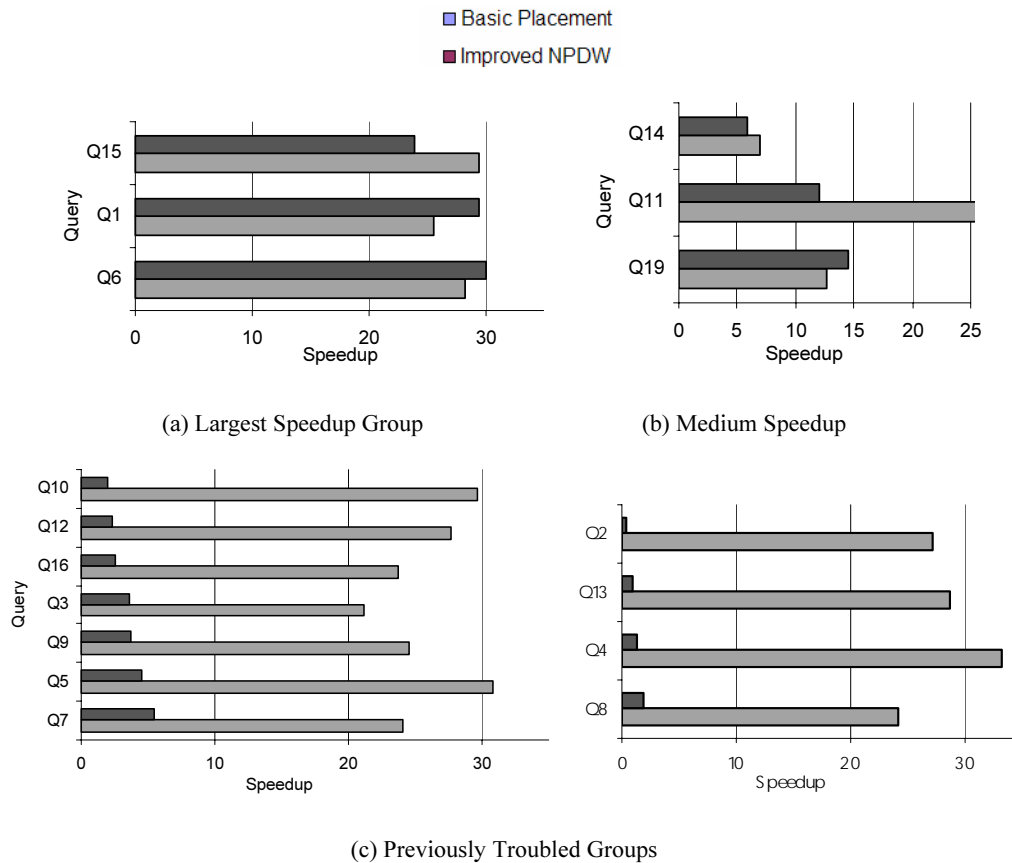
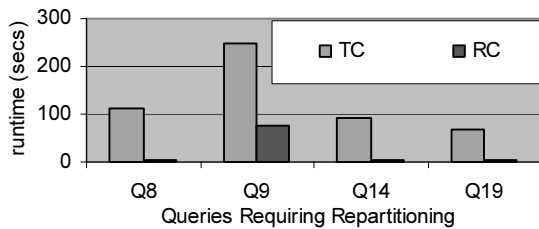


Figure 7 - TPC-H Speedup on Improved Placement

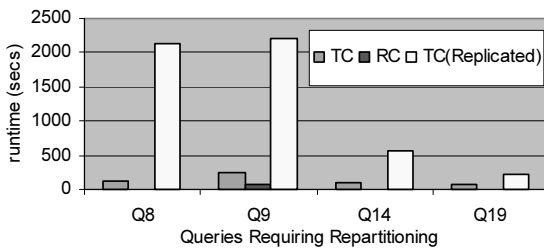
These results show that, by applying the simple placement algorithm proposed in section 6, most speedup problems are minimized and speedup becomes much nearer to linear for most query patterns. The major improvements in Figure 7(c) have to do with the fact that the Orders (O) relation is no longer fully replicated, being instead partitioned by the equi-join key to Lineitem (LI). Other queries benefited from PS being equi-partitioned with P. At the same time, queries which were the fastest under the basic placement strategy (a) are still as fast, because they access partitioned relations in both cases.

While the most important performance issue in the initial replicated placement scheme was the need to process large full relations at each node, repartitioning requirements may be an important issue in the improved partitioning and placement scheme. However, not only it is possible that many queries do not require repartitioning, but also repartitioning may be much less expensive than having (possibly very) large relations completely replicated.

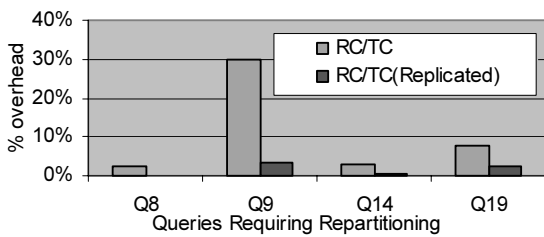
Figure 8 shows the results obtained by us concerning the only queries from the TPC-H query set of Figure 6 that required repartitioning (the cost is measured in runtime). These results compare the repartitioning overheads with the total runtime of the improved scheme (a) and of the basic replicated scheme (b), while Figure 8(c) shows percent results.



(a) Runtime Cost



(b) Runtime Cost, including Replicated



(c) % of Runtime Cost

Figure 8— Repartitioning Overhead

The repartitioning cost in Figure 8(a) is clearly a reasonably small fraction of the total cost (TC) for most queries, except for Q9. Figure 9(c) shows that RC is about 30% of the total cost (TC) for Q9, less than 10% for Q19 and less than 5% for the other two queries. But if you look at Figure 8(b), you can see that even for Q9 the RC cost is very small compared with the total cost of the initial replicated scheme. This is apparent also from Figure 8(c), where we can see that $RC/TC(\text{Replicated})$ is below 5% even for Q9.

In summary, concerning TPC-H, most queries required no repartitioning at all with the improved placement and those which did require repartitioning still obtained a speedup that was orders of magnitude faster than the initial replicated placement scheme.

8. CONCLUSIONS

In this paper we have analyzed join overheads in Parallel Data Warehouses with the help of experimental simulation data. We have setup TPC-H for parallel execution under a basic placement strategy that dictated replicated dimensions and round-robin or random horizontal partitioning of facts and analyzed the shortcomings of such a simple scheme when considering more complex schemas than the single large fact, multiple very small dimensions scenario. We proposed simple and easy to apply modifications to the basic placement schema, based on hash-partitioning and favouring most frequent equi-join keys as the partitioning keys. We applied the strategy to TPC-H and observed large improvements in the worst behaving queries. We conclude that, although it is not possible to guarantee linear speedup for all query patterns, workload-friendly placement can prevent very low speedup and provide near to linear speedup for most queries in Node Partitioned Data Warehouses.

9. 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] Pedro Furtado, "Workload-based Placement and Join Processing in Node-Partitioned Data Warehouses", in *International Conference on Data Warehousing and Knowledge Discovery (Dawak)*, 2004.
- [6] Hua, K. A. and Lee, C., "An Adaptive Data Placement Scheme for Parallel Database Computer Systems" *Proc. VLDB Conf., Brisbane, Australia, 1990.*

- [7] Liu, Chengwen and Hao Chen, "A Hash Partition Strategy for Distributed Query Processing", International Conference on Extending Database Technology (EDBT) 1996.
- [8] 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.
- [9] 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.
- [10] Livny, Miron, Setrag Khoshafian, Haran Boral: "Multi-Disk Management Algorithms". In Procs. 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] Clement T. Yu, C. C. Chang: Distributed Query Processing. ACM Computing Surveys 16(4): 399-433 (1984).
- [16] 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).
- [17] 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.
- [18] Yu, Clement T. and Weiyi Meng. "Principles of Database Query Processing for Advanced Applications", 1998 Eds. Morgan Kaufmann ISBN 1-55860-434-0.
- [19] 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.
- [20] Zhou S., M.H. Williams, "Data placement in parallel database systems," Parallel Database Techniques, IEEE Computer Society Press, 1997.
- [21] Transaction Processing Council Benchmarks, www.tpc.org.
- [22] 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.