

Algorithms for Efficient Processing of Complex Queries in Node-Partitioned Data Warehouses

Pedro Furtado

DEI/CISUC, Universidade de Coimbra, Portugal

pnf@dei.uc.pt, <http://eden.dei.uc.pt/~pnf>

Abstract

In this paper we propose and evaluate an algorithm for efficient processing of complex queries in a partitioned data warehouse. Partitioning allows cheap computer nodes to be able to host and process efficiently large data warehouses. In order for the system to be truly useful, significant speedup should be achieved for all query patterns. We describe the system, identify complex join queries and propose algorithms to deal efficiently with those queries. We also take a set of performance-constrained computer nodes and the query set from the decision support performance evaluation benchmark TPC-H as a study case and evaluation basis for our analysis.

1. Introduction

Processing and performance issues in parallel and distributed databases have received lots of attention in the past. One of the interesting issues was the advantage of using clusters of lower-cost nodes to process efficiently against databases in general, typically under generic OLTP workloads and sometimes more complex analysis queries. The main issues in such environments include data allocation, query processing and load balancing. Not all database access patterns benefit linearly (linear speedup) from parallel architectures and some can have much less than linear speedup unless expensive massively parallel hardware is used. Consider a cluster of low-cost nodes or a LAN environment. There is some evidence that relation partitioning does not work equally well for all workloads in such environments. In an early work [15] 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. [6] 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 [6] shows 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 are precisely that kind of context in which some complex query patterns can exhibit this high communication overhead. In this paper we propose an algorithm with two alternative paths for efficient processing of complex queries on a node-partitioned data warehouse (NPDW). First we describe the partitioning of the NPDW and the complex query issue and then propose our strategy.

The paper is organized as follows: section 2 discusses related work. Section 3 discusses data placement and query processing basics in NPDW. Section 4 reviews relevant query processing costs in NPDW. Section 5 proposes the query processing algorithm alternatives and section 6 shows experimental results involving TPC-H [22]. Section 7 contains concluding remarks.

2. Related Work

A large body of work exists in applying parallel processing techniques to relational database systems. The objective is to apply either inter-query or intra-query parallelism to improve performance. Query processing in parallel and distributed databases has been the focus of much research on the field [1, 3, 5, 8, 21]. Although standardized relational algebra operators used in databases lend themselves well to parallelization, issues are raised concerning mainly join processing and communication overheads associated with both joins and other operation needs to exchange data [6, 15, 16, 18]. Parallel Hash Join algorithms [7, 11, 12, 18, 19] and Placement Dependency algorithms [5, 13, 14] are typical proposals to reduce some of the overhead associated with join operations.

Data placement has been another important issue in parallel and distributed database architectures [2, 6, 10,

17] also reviewed in [21]. Partitioning the data for independent processing of fragments can typically reduce response time drastically but simultaneously increase total work, due to the need to exchange and merge data and partial results [6]. For complex heavy join queries this can result in important speedup deterioration. Data warehouses are precisely that kind of environment in which node partitioning can improve response time but also cause important extra overheads.

In this paper we take a partitioned DW by full-partitioning facts and replicating smaller dimensions as-is to all nodes as in [9, 4] to avoid join communication costs. However, instead of a strict star schema and simple queries involving a single fact and dimensions, we consider more complex and generic schemas and access patterns may involve several partitioned relations and more complex query scenarios involving multiple partitioned relations. We consider such an environment and propose an efficient query processing algorithm and alternatives to deal with such a system.

3. Placement and Query Processing Basics

The Node Partitioned Data Warehouse (NPDW) 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 two facts and conformed dimensions. The largest, heaviest processing elements in the DW – fact tables – are partitioned horizontally into nodes using round-robin partitioning, while the much smaller dimensions, typical targets of join processing, are fully replicated into the nodes, as shown in Figure 1 for node *i*.

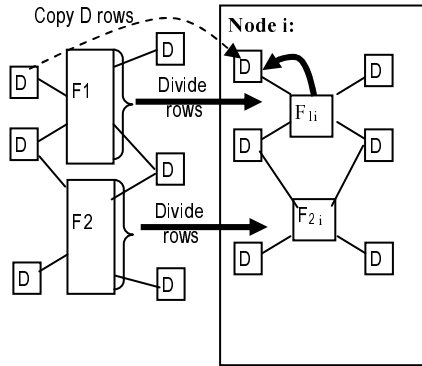


Figure 1 – The NPDW

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, ..., D_n)
 GROUP (G₁...G_m);

Each node would need to apply exactly the same initial query on its partial data and the results would be merged by applying the same query again at the merging node with the partial results coming from the processing nodes. Round-robin or random partitioning ensures a balanced load among processors.

We have included two facts in the example of Figure 1 on purpose, to introduce the discussion on how to process more complex queries such as one involving both facts F1 and F2. The previous query example was very efficiently processed because $F \times_A D_1 \times_B \dots \times_z D_n \Leftrightarrow U (F_i \times_A D_1 \times_B \dots \times_z D_n)$, where x_A is an equi-join on attribute A, F is a fact and D_i are dimensions. Each node processes the multi-way join $(F_i \times_A D_1 \times_B \dots \times_z D_n)$ independently and a merging phase simply operates a UNION on the result. The two fact query example cannot be handled this way. It requires a possibly large amount of data to be exchanged and we deal with such problem in this paper.

The generic partitioning issue that is the source of the problem is stated as follows: consider two partitioned relations R_1 and R_2 and let a query include $R_1 \times_A R_2$. Unless R_1 and R_2 are partitioned on the equi-join attribute (hash-partitioning or placement dependency), the join cannot be processed locally. Then, either one of the relations is reconstructed in every node to process the join (with large data exchange and join overhead) or repartitioning is used to allow faster parallel hash-join to be used.

We point out that it is frequently possible and important to partition facts F_1 and F_2 initially (placement) by some equi-join attribute in order to allow the query to be processed at each node independently as in the simple one fact case. However, in this paper we concentrate on round-robin or random placement scheme to guarantee load balance. Additionally, the strategy depicted here will be useful even if the relation is partitioned by one equi-join attribute because the problem remains when a different equi-join attribute is required for a different join.

Next we 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 .

Repartitioning is intended to hash-partition the relations by the equi-join attribute. Consider two nodes. Repartitioning allows the expression $R_1 \times_A R_2$ to be processed faster by the nodes as $(R_{11} \times_A R_{21}) \cup (R_{12} \times_A R_{22})$, as fragments are results of hashing the relations such

that $R_{11} \times_A R_{22} = \emptyset$ and $R_{12} \times_A R_{21} = \emptyset$. The join cost at each node is now smaller as only a fragment of R_2 and R_1 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.

We are interested in obtaining a generic solution to process efficiently any query in NPDW, considering less de-normalized schemas than a “pure star” as well. A good example of a more complex schema (not a pure star schema at all) and query set that we want our system to handle efficiently is the one illustrated here by TPC-H [22]. Figure 2 summarizes the TPC-H schema, which represents ordering and selling activity (LI-lineitem, O-orders, PS-partsupp, P-part, S-supplier, C-customer). It is not a “pure” multidimensional star schema as the one shown in Figure 1 but is a plausible historical record of the company business activity. The TPC-H query set includes frequent joins between two or more large relations. In practice, even if the data warehouse is made of “pure star” schemas and small dimensions, each fact describes only a small part of the business and joining different stars is often required.

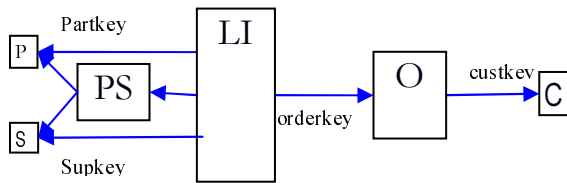


Figure 2 – TPC-H schema

For simplicity, we will portray relations PS, LI and O as facts and the remaining ones (P,S,C) as dimensions in TPC-H. The basic node partitioning strategy defines that P, S and C be copied into each node. On the other hand, relations LI, PS and O should be horizontally partitioned into the nodes. 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. However, at least half of the TPC-H queries include joins involving two or more partitioned facts and we can imagine that even if LI and O are co-located based on orderkey all queries involving PS and LI have the problem that is handled in this paper.

4. Relevant Costs in the NPDW

It is important to quickly review the costs involved in processing possibly complex queries in NPDW. The main processing costs in NPDW are the partitioning, repartitioning, data communication and local processing costs.

Partition 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 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 (PC) – 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.

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.

In this paper we concentrate on costs related to join processing and do not take into account merging costs and costs related to processing other query clauses for simplification. Join processing is the fundamental issue.

5. Efficient Processing of Complex Queries in NPDW

In this section we describe the processing strategy for complex queries in NPDW. In [18] heuristic algorithms are proposed to solve the problem of finding the minimum cost spanning tree from the query graph given a star or general graph query. In NPDW the problem is much more constrained than the general one by the objective of promoting independent node processing of query parts (by replicating dimensions) and seeking balanced homogeneous workloads. The algorithm developed in this section is a generalization of the strategy we used to handle single-fact queries. Both the single fact and the proposed multiple fact algorithms assume that the dimensions are much smaller than facts. Figure 3 shows two examples of query graphs where vertices correspond to relations and edges are equi-joins. For simplicity, we assume all equi-joins are on different attributes.

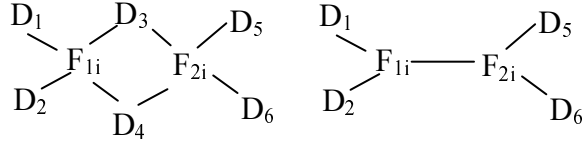
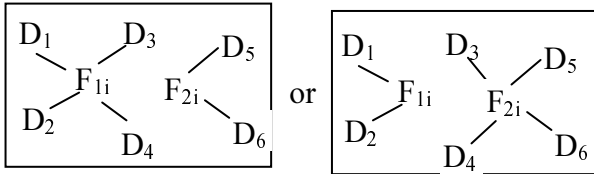


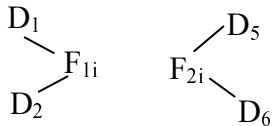
Figure 3 – Query Example Graphs

The first step occurs in parallel at all nodes. We take advantage of the replication of (small) dimensions to promote partial joins of facts with equi-joined dimensions first. This is based on applying the same property of section 3 to each partitioned fact in each node: $F x_A D_1 x_B \dots x_z D_n \Leftrightarrow U (F_i x_A D_1 x_B \dots x_z D_n)$. It promotes joining with smaller relations (dimensions) first, which most frequently introduces restrictions that result in very good intermediate result selectivity. Each partitioned fact is equi-joined to the dimensions it is linked with. At the end of this phase we are left with intermediate results for each partitioned fact.

Figure 4 illustrates the first step applied to the query graph of Figure 3. Figure 4(a) shows two alternatives as dimensions D3 and D4 can be equi-joined with one or the other fact. The actual join order between each fact and equi-joined dimensions and which of the alternatives in Figure 4a is chosen should be determined cost-based.



(a) Alternative single node processing of 1st query



(b) Single node processing of 2nd query

Figure 4 – Independent Node Query Processing

The first step produces as many sets of intermediate results (IR) in each node as there are partitioned relations, each of the form $(F_{1i} x_A D_x x_B \dots x_z D_y)$. These IRs form a chain (with only two facts in the examples of Figure 4) which cannot be processed locally because they are not partitioned on (the same) equi-join keys. These are now processed using the parallel hash-join algorithm PHJ [7, 11]. To apply the PHJ considering a join between two IRs, both are repartitioned on the equi-join attribute (hash-partitioning) and assigned a join processing node according to the hash-value (repartitioning cost). The join is then processed in parallel by the nodes. The exact order of execution of long chains should be determined based on costs (e.g. chain algorithm in [18]). The complete algorithm with alternative paths is:

1. Produce Reduced Intermediate Results (IR)

Process in parallel all locally computable joins around the partitioned relations (with replicated dimensions). In this step apply the selection operators (restrictions on the dimensions and on the partitioned fact itself), producing selective intermediate results (IRs);

2a. Parallel Hash-join IRs (PHJ-IR):

Determine best join order for the IRs.

For each IR join step

Repartition intervening IRs by join attribute

Process the join in parallel at all nodes

After 2a the query processing proceeds and at some point partial results are merged.

This algorithm 2a can be confronted with an alternative algorithm which involves replicating all but one of the IRs obtained in step 1 (broadcast) instead of repartitioning IRs. Once all IRs are replicated, the strategy can process the rest of the joins locally without further repartitioning.

2b. Replicate IRs, use Parallel Associative Join (PAJ-IR):

For all but one of the IRs,

Broadcast the partial IRs to all nodes to fully reconstruct the IR.

Process the query locally with only one partitioned IR, full dimensions and full broadcasted IRs;

Alternative 2a may have smaller communication overhead if a switched interconnect/network is used and has smaller processing overhead as it processes only fragments from all IRs in each node. However, it incurs repartitioning costs not incurred by alternative 2b. As we will show by our experimental results, the best algorithm from these two alternatives depends on intermediate results selectivity, so that cost estimation procedures should be followed to determine which one should be used.

The previous algorithm was described sequentially, but the two steps described above should be pipelined for maximum efficiency. For instance, step 2 repartitioning or replication can be overlapped with step 1 as IRs are being constructed.

6. Experimental Analysis of the Algorithms

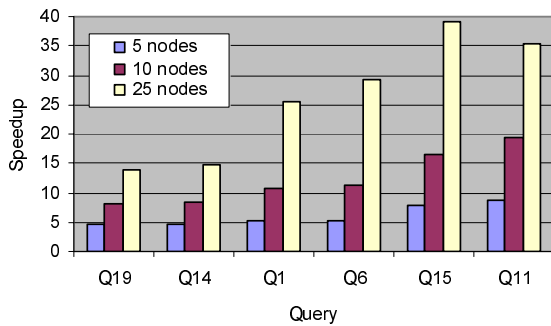
Four data sets were created for the experiments corresponding to a 50GB TPC-H generation into 1, 5, 10 and 25 nodes. The experimental evaluation was conducted by measuring the node and communication costs for the

algorithms assuming similar nodes on a 100Mbps network. Nodes were Intel Pentium III 866 MHz CPU with three IDE hard disk s(2x40 GB, 1x80GB) and 512 MB of RAM, running Windows XP professional and Oracle 9i (9.2) DBMS, and the TPC-H decision support benchmark™ [22]. The experiments also involved transforming TPC-H queries to run against the NPDW.

We have adjusted instance parameters for best performance given the available resources and created all primary and foreign key indexes for all data sets. After data set loading and index creation and before each experiment schemas were analyzed to collect statistics for best cost-based optimization.

The performance of a single node running the whole data set is compared with that of a set of nodes running the same data set under NPDW.

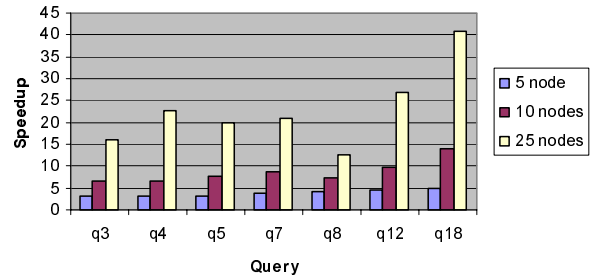
Our experiments have shown typically near to linear speedup for single fact queries, as shown in Figure 5. The average speedup values for the queries tested are also shown to be near to linear (slightly larger than linear).



Average Speedup	5 nodes	10 nodes	25 nodes
	6±1.8	12.4±4.5	26.4±10.5

Figure 5 – NPDW Speedup with Simple Queries

We would like our algorithm to achieve similar results for more complex queries involving more than one partitioned relation. However, the repartitioning or replication of IRs is sure to introduce additional overhead. Figure 6 shows the results for several complex TPC-H queries (more than one partitioned relation). The speedup is much smaller than the single partitioned relation speedup but the system is still capable of obtaining a reasonable speedup.



Average Speedup	5 nodes	10 nodes	25 nodes
	3.9±0.7	8.7±2.3	22.8±9

Figure 6 – Complex Queries Speedup using PHJ

We also compared the two alternative IR processing algorithms (PHJ-IR and PAJ-IR). Figure 7 shows the results of the comparison considering 25 nodes. PHJ-IR obtained better average performance but not for all queries. When the IRs that need to be replicated are small PAJ-IR can obtain better results than PHJ-IR as it does not have the repartitioning costs of this algorithm.

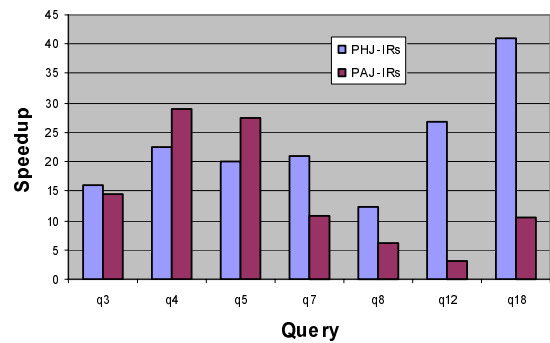


Figure 7 – Comparing PHJ-IR to PAJ-IR on 25 nodes NPDW

Finally, Figure 8 shows average speedup results of PAJ-IR with 5, 10 or 25 nodes to compare with those of PHJ-IR. It is interesting to see that the average speedup with 5 nodes was actually better than the one of PHJ-IR (4 against 3.9) but with 10 and 25 nodes PHJ-IR is clearly superior. This is mainly because PHJ-IR processes IR fragments, which become smaller as more nodes are available, whereas PAJ-IR processes whole IRs.

Average Speedup	5 nodes	10 nodes	25 nodes
	4+-1.2	7.9+-3.6	14.5+-8.5

Figure 8 – Average Speedup Results PAJ-IR

7. Conclusions

In this paper we have proposed and evaluated an efficient strategy to process complex queries with more than a single partitioned relation in a node partitioned data warehouse. We have analyzed experimentally the performance of two alternative implementations of the algorithm and concluded that the strategy is able to maintain a reasonably large speedup in spite of the increased burden of repartitioning needs.

8. References

- [1] P.M.G. Apers, A.R. Hevner, and S.B. Yao, "Optimization Algorithms for Distributed Queries," IEEE Trans. Software Eng., vol. 9, no. 1, pp. 57-68, Jan. 1983.
- [2] Apers, P. M. G.. Data allocation in distributed database systems. ACM Transactions on Database Systems, 13(3):263--304, September 1988.
- [3] P.A. Bernstein, N. Goodman, E. Wong, C. Reeve, and J.B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-I)," ACM Trans. Data&se Systems, vol. 6, no. 4, pp. 602-625, Dec. 1981.
- [4] J. Bernardino, Pedro Furtado and Henrique Madeira, "Approximate Query Answering Using Data Warehouse Striping", "Journal of Intelligent Information Systems ", V 19-2, Elsevier Science, September 2002.
- [5] 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.
- [6] Copeland G. P., William Alexander, Ellen E. Boughter, Tom W. Keller: Data Placement In Bubba. SIGMOD Conference 1988: 99-108.
- [7] David J. DeWitt, Robert Gerber, Multiprocessor Hash-Based Join Algorithms, Proceedings of the 11th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Stockholm.
- [8] 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.
- [9] Furtado, Pedro: "Implementing and Evaluating Warehouses and Summaries Over a Cluster". Simposium Brasileiro de Bancos de Dados, SBBD 2003.
- [10] Hua, K. A. and Lee, C., "An Adaptive Data Placement Scheme for Parallel Database Computer Systems" Proc. VLDB Conf., Brisbane, Australia, 1990.
- [11] M. Kitsuregawa, H. Tanaka, and T. Motooka. Application of hash to database machine and its architecture. New Generation Computing, 1(1):66-74, 1983.
- [12] Liu, Chengwen and Hao Chen, "A Hash Partition Strategy for Distributed Query Processing", International Conference on Extending Database Technology (EDBT) 1996.
- [13] 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.
- [14] 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.
- [15] Livny, Miron, Setrag Khoshafian, Haran Boral: "Multi-Disk Management Algorithms". In Procs. Of ACM SIGMETRICS 1987, pp69-77.
- [16] S. Pramanik and D. Vineyard, "Optimizing Join Queries in Distributed Databases," IEEE Trans. Software Eng., vol. 14, no. 9, pp. 1,319.
- [17] Sacca, D. and Wiederhold, G.: Database Partitioning in a Cluster of Processors. ACM TODS, Vol. 10, No. 1, pp. 29-56, Mar 1985.
- [18] 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.
- [19] Teradata Corporation. Database Computer System Concepts and Facilities. Document C02-0001-01, Teradata Corporation, Los Angeles, Oct. 1984.
- [20] 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).
- [21] Zhou S., M.H. Williams, "Data placement in parallel database systems," Parallel Database Techniques, IEEE Computer Society Press, 1997.
- [22] Transaction Processing Council Benchmarks, www.tpc.org.