PartJoin: An Efficient Storage and Query Execution for Data Warehouses

Ladjel Bellatreche[1], Michel Schneider[2], Mukesh Mohania[3], and Bharat Bhargava[4]

[1] IMERIR, Perpignan, FRANCE
ladjel@imerir.com
[2] LIMOS, Blaise Pascal University 63177, Aubière, FRANCE
michel.schneider@isima.fr
[3] IBM India Research Lab, I.I.T., Delhi, INDIA
mkmukesh@in.ibm.com
[4] Computer Science Department, Purdue University, USA
bb@cs.purdue.edu

Abstract. The performance of OLAP queries can be improved drastically if the warehouse data is properly selected and indexed. The problems of selecting and materializing views and indexing data have been studied extensively in the data warehousing environment. On the other hand, data partitioning can also greatly increase the performance of queries. Data partitioning has advantage over data selection and indexing since the former one does not require additional storage requirement. In this paper, we show that it is beneficial to integrate the data partitioning and indexing (join indexes) techniques for improving the performance of data warehousing queries. We present a data warehouse *tuning strategy*, called *PartJoin*, that decomposes the fact and dimension tables of a star schema and then selects join indexes. This solution takes advantage of these two techniques, i.e., data partitioning and indexing. Finally, we present the results of an experimental evaluation that demonstrates the effectiveness of our strategy in reducing the query processing cost and providing an economical utilisation of the storage space.

1 Introduction

A data warehouse (DW) is an information base that stores a large volume of extracted and summarized data for OLAP and decision support systems [4]. These systems are characterized by complex ad-hoc queries (with many joins) over large data sets. Despite the complexity of queries, decision makers want those queries to be evaluated faster. Fast execution of queries and retrieval of data can be achieved if the the physical design of a DW is done properly. There are two major problems associated with the physical design of a DW, namely, the *selection of warehouse data (i.e., materialized views)* so that all the queries can be answered at the warehouse without accessing the data from underlying sources, and *indexing data* so that data can be retrieved faster. However, the solutions

Y. Kambayashi, W. Winiwarter, M. Arikawa (Eds.): DaWaK 2002, LNCS 2454, pp. 296–306, 2002.

of these two problems pose additional overheads of maintaining the warehouse data (materialized views) whenever the source data changes and storage cost for maintaining index tables. Given the large size of DWs, these costs are non-trivial. This prompts us to ask the following question: *is it possible to reduce the storage and maintenance cost requirements, without sacrificing the query execution efficiency obtained from indexing?*

We address this vital issue in our ongoing *PartJoin* project that combines data partitioning and join indexes in an intelligent manner so that one can reduce the storage cost and improve the query performance. We now briefly outline the status of this project. In [2], we have shown the utility of data partitioning and presented an algorithm for fragmenting a data warehouse modeled by a star schema. In [1], we have presented an indexing scheme called graph join indexes for speeding up the join operations in OLAP queries. These indexes are a generalization of star join indexes [7]. Later on, we figured out that conceptually a star join index or a horizontal fragment of a fact table can significantly improve the performance of queries (see the motivating example in section 2), but the utilization of indexes is constrained by a storage capacity and a maintenance cost. The use of partitioning does not require a storage capacity, but poses the problem of managing numerous partitions. Therefore, it appears very interesting to combine data partitioning and indexes in order to provide a better performance for queries and to minimize the storage and maintenance overheads.

In this paper, the terms fragmentation and partitioning are used interchangeably. We propose a new tuning strategy technique for efficiently executing queries while using space *economically*. Our approach exploits the similarities between join indexes and data partitioning. The crucial and the complex problem of this strategy lies in how it efficiently selects a *better partitioning schema* of a DW modeled by a relational schema and *appropriate join indexes* for a given set of queries. The main contributions of this paper are the following :
a) We have identified the similarities (in terms of performance point of view) between data partitioning and join indexes and the need for combining them to reduce the query processing, storage and maintenance costs.
b) We have proposed a tuning strategy called *PartJoin* for *fragmenting* a star schema under a threshold for the number of fact fragments that the warehouse administrator can manage them easily and then to *select* join indexes.
c) We have evaluated the *PartJoin* methodology using the APB benchmark [5].

To the best of our knowledge, the proposed work is the first article that addresses the problem of selecting a partitioning schema and join indexes in a DW under the fragmentation threshold constraint for the number of fact fragments and the storage constraint for indexes.

The rest of the paper is organized as follows. A motivating example is described in Section 2. Section 3 gives the architecture of our *PartJoin*, describes its components, presents an algorithm that decomposes a star schema. Section 4 outlines the results of our performance study. Section 5 concludes the paper.

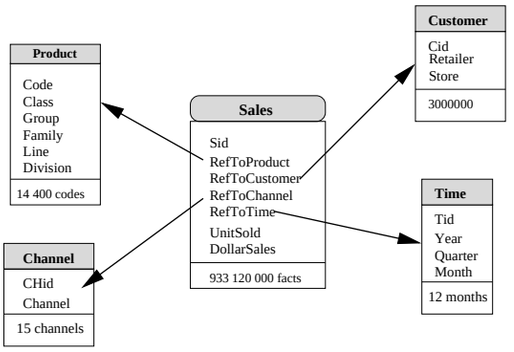

Fig. 1. An Example of a Star Schema (derived from APB-1)

2 A Motivating Example

In this section, we present an example to show how the data partitioning [2] can complement with the join indexes for improving the performance of OLAP queries in data warehousing environments [7]. We assume that the warehouse data is modeled based on a star schema. Figure 1 shows a star schema derived from APB-1 benchmark of OLAP council [5]. This schema consists of four dimension tables *Customer*, *Product*, *Time*, and *Channel* and one fact table *Sales*.

Suppose that the dimension table *Time* is horizontally partitioned using the attribute *Month* into 12 fragments $Time_1, ..., Time_{12}$[1], where each fragment Time_i ($1 \leq i \leq 12$) is defined as follows : $Time_i = \sigma_{T.Month=\text{“}Month_i''}(Time)$. The fact table *Sales* can be partitioned based on the fragmentation schema of the dimension table *Time* into 12 fragments $Sales_1, ..., Sales_{12}$, where each fragment $Sales_i$ ($1 \leq i \leq 12$) is defined as follows: $Sales_i = Sales \ltimes Time_i$, where $\ltimes$ represents the semi-join operation. This type of fragmentation is known as *derived horizontal fragmentation* [8]. The attribute *Month* is called the *partitioning attribute*[2]. A fragment of a fact table and a dimension table are called *fact fragment* and *dimension fragment*, respectively. The fragmentation of dimension and fact tables incurs the decomposition of star schema into set of sub star schemas.

Imagine a star join index between the fact table *Sales* and the dimension table *Time* that correlates the dimension tuples with the fact tuples that have the same value on the common dimensional attribute *Month*. A bitmap join index [7] on the month column in the *Sales* table can be built by using the month column in the *Time* table and the foreign key Time ID (Tid) in the *Sales* table. This representation is quite similar to a derived fragment of the *Sales* table. A star join index is a vertical fragment of a fact fragment and it contains only the foreign keys of the fact fragment.

[1] Our schema models one year sales activities
[2] We can have a fact fragment referring many attributes of several dimension tables.

Therefore, to execute queries, there can be two options: (1) *to partition* the warehouse so that each query can access fragments rather than whole tables, and (2) *to use join indexes* so that each query can access data that uses join indexes. To illustrate these options, let us consider the following query Q : SELECT Tid, Sum(UnitSold), Sum(DollarSales) FROM Sales S, Time T WHERE S.RefToTime = T.Tid AND T.Month = "June" GROUP BY Tid.

1. *With the partitioning option :* In this case, only one fact fragment (sub star schema) is accessed by the query Q. This option has two main advantages: (1) it reduces the query processing cost by eliminating 11 sub star schemas and (2) it does not require an extra storage cost.
2. *With the join indexes option :* Suppose we have a star join index I between the fact table *Sales* and the dimension table *Time* which refers to the sales done during June. It is defined as follows: $I = \pi_{Sid,Tid}(Sales \bowtie (\sigma_{Month="June"}(Time)))$. This index gives about the same performance as the partitioning option for reducing the cost of Q, but it needs to be stored, and updated when tuples are inserted or deleted in the underlying tables (*Sales* and *Time*).

Under these scenarios, we need to answer the following question : *for a given set of queries, which option should be used ?*

We consider three classes of queries when we are dealing with HP [2,6]: *best match queries*, *partial match queries*, and *worst match queries*

1. *Best match queries:* A query belonging to this type references all partitioning attributes, i.e., query selection predicates match with fragmentation predicates[3]. Ideally, a best match query accesses only one fact fragment (the above query is a good example of this class). For this type of queries, join indexes are not needed because we do not have any join operation and thus, the data partitioning may give equal or better performance than join indexes and without an additional storage cost.
2. *Partial match queries:* In this case, a query references a subset of the fragmentation attributes. Data partitioning may not very efficient for this type of queries, because we need to perform some join operations between fact and dimension fragments that can be costly. Therefore, the utilization of join indexes *may be suitable* to speed up these operations.
3. *Worst match queries:* In this type, a query does not contain any selection predicate, or it has some selection predicates defined on non fragmented attributes. By considering the partitioning option, we need to execute a query locally (on each sub star schema) and then assembly all local results by using the union operation. In this case, the data partitioning may perform badly and therefore the utilization of join indexes is recommended.

Our *PartJoin* tuning strategy gives to the DW administrator a *new option* to speed up his/her queries (best match, partial match and worst match) by combining the partitioning option (partitions the warehouse schema) and the join

[3] The fragmentation predicates are the predicates that are used in fragmentation process

index option (builds indexes on top of partitions). This new option guarantees query performance, a good utilization of the space storage and reduces the maintenance overhead.

3 The PartJoin Tuning Strategy

In this section, a formulation of the *PartJoin* problem and its architecture are described.

3.1 Formulation of PartJoin Problem

Given the following inputs, the *PartJoin* problem is formulated as follows:
Inputs: (1) A star schema $S : (F, D_1, ..., D_d)$, (2) a set of most frequently used OLAP queries $\{Q_1, ..., Q_l\}$ and their frequencies $\{f_1, ..., f_l\}$, (3) selectivity factors of simple and join predicates defined in the queries, (4) a storage capacity constraint C for join indexes, and (5) a fragmentation threshold W which represents the maximal number of fact fragments that the administrator can maintain.
Goal : The *PartJoin* problem consists in partitioning the star schema S into several sub star schemas $\{S_1, S_2, ..., S_N\}$ and in selecting join indexes on top of these sub schemas in order to minimize the processing cost of all queries. The selected indexes should be accommodated in C (storage capacity).

3.2 PartJoin Architecture

An architectural overview of the *PartJoin* system is shown in Figure 2. It has two main modules : *a partitioning module* and *an index selection module*.
The first one is responsible for fragmenting the warehouse schema and the second one for selecting the appropriate join indexes for partition(s) (sub star schema(s)). Our system works as follows:
The partitioning module identifies selection predicates used by the input queries. Using these predicates, it partitions first the dimension tables of the warehouse schema, and then the fact table using the derived fragmentation technique (see Section 2). If the data partitioning cannot satisfy all queries, the *index selection module* selects join indexes which respect storage constraint (C) to ensure that the performance of the rest of queries (those are not satisfied by partitioning module) is not deteriorated. Finally, we obtain a set of sub star schemas and join indexes that minimizes query processing and storage costs.

To decompose the star schema S into a set of sub star schemas $\{S_1, S_2, ..., S_N\}$, the partitioning module starts by enumerating all the selection predicates used by the set of input queries. These predicates are classified based on their dimension tables (each dimension table has its own selection predicates). It takes into account only dimension tables having a non empty set of selection predicates. Therefore, it uses the COM-MIN procedure developed by Özsu et al. [8] to partition each dimension table. Once the dimension tables are partitioned, we then derive the horizontal fragments of the fact table.

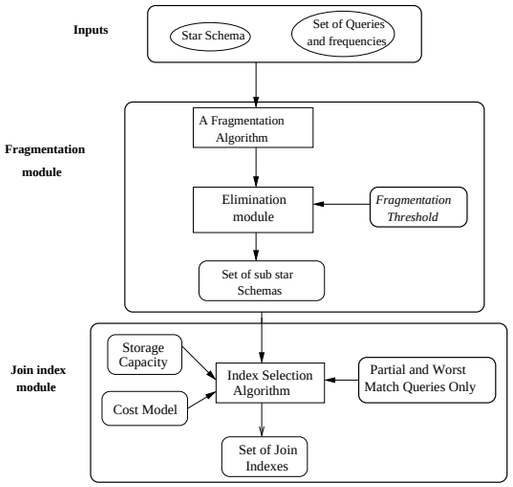

Fig. 2. The PartJoin architecture

To ensure query processing reduction of all queries, the join index module selects indexes after the partitioning process. When the warehouse is partitioned, the join index selection problem will be different than the join index selection in non partitioned warehouse. The main differences concern : The *number of queries* taken into consideration during the selection process, and *the warehouse schema* (with partitioning, we have N sub star schemas and not a single star schema. However, a sub star schema has the same characteristics as a whole star schema, thus we can apply any join index selection algorithm [1]).

Concerning the number of queries, we have previously mentioned that join indexes are not needed for best match queries. So, these queries will be removed from the initial set of queries Q and then a join index selection algorithm will take into account only queries that do not get benefit from the fragmentation process (worst match and partial match queries).

Figure 2 summarizes the main steps of the *PartJoin* tuning strategy.

3.3 Fragment Merging

Let g be the number of dimension tables participating in the fragmentation process. The number of fact fragments (N) generated by the partitioning algorithm is given by the following equation: $N = \prod_{i=1}^{g} L_i$, where L_i represents the number of fragments of the dimension table D_i. This number (N) can be very large. For example suppose we have the following partitioning schemas of dimension tables (cf. Figure 1: *Customer* is partitioned into 1440 fragments using the *store* attribute, *Time* into 12 fragments using the *month* attribute (we have one year of sales activities), *Product* into 50 fragments using the *family* attribute, *Channel*

into 15 fragments using the channel attribute. This implies that the fact table will be decomposed into 12 960 000 (1440 * 12 * 50 * 15) fragments.

Consequently, it will be very hard for the warehouse administrator to install and maintain these fragments (sub star schemas). Thus it is important to reduce the number of fact fragments. This reduction is done by the *partition elimination* sub module, a part of the partitioning module (see Figure 2). To achieve this elimination, the warehouse administrator considers a fragmentation threshold denoted by W that represents the maximal number of fact fragments that he/she can maintain. By considering this threshold, the partitioning module task is to decompose the star schema S into N sub star schemas such that: $\mathbf{N} \leq \mathbf{W}$.

Our goal is to reduce the number of fact fragments in order to satisfy the fragmentation threshold W. This reduction can be done by merging some fragments. The merging operation of two (or several) fragments is done by using the union operator of these fragments [8]. In this paper, this operation is done statically.

Since we have an important number of fact fragments, we should have a metric that identifies which fragments should be merged. Intuitively, a fact fragment is interesting if it reduces the cost of workload significantly. To compute the contribution of each fragment F_i $(1 \leq i \leq N)$, we define the following metric: $Cont(F_i) = (||F|| - ||F_i||) \times \sum_{j=1}^{l} a_{ij} \times Freq(Q_j)$, where $||F||$ and $||F_i||$ represent the sizes of fact table and fact fragment F_i. The element a_{ij} $(1 \leq i \leq N, 1 \leq j \leq l)$ can have a binary value (1 if the fragment F_i is accessed by the query Q_j, 0 otherwise). $Freq(Q_j)$ is the access frequency of the query Q_j. Recall that l is the number of queries in the workload. The merging operation is done as follows: each fragment F_i is assigned by a contribution $cont(F_i)$. We sort then these N fragments in a decrease order. We keep the $(W-1)$ first fragments and we merge the rest into one fragment. By doing like this, the fragmentation threshold is satisfied.

After applying merging operations and satisfying the fragmentation threshold, the performance of certain queries may be affected. Therefore, the index module satisfies these queries.

4 Performance Evaluation of PartJoin Method

In this section, we present some performance results of *PartJoin* tuning method, that considers both query processing costs and storage requirements. The query processing cost is estimated in terms of the number of rows used during query execution. We then also compare the results of this method with *partitioning* and *join index* methods and the results show that *PartJoin* tuning strategy is better than these two methods for the best and partial match queries.

4.1 Experimental Setup

In our experiments, we use dataset from APB benchmark [5] (see Figure 1). The database size here refers to the size of the raw data, and thus does not include

any overhead that may be added once the data is loaded. The number of rows of each table is given in Figure 1. In these experiments, the storage cost of a join index is estimated in terms of the number of rows in that index. The fact table is partitioned based on three dimension tables *Time*, *Product* and *Channel*, where each dimension table is fragmented as follows: *Time* into 12 fragments using the attribute Month, *Product* into 4 fragments using the attribute Family[4], *Channel* into 15 fragments using the attribute Channel. Therefore the fact table *Sales* is partitioned into 720 fragments ($W = 720$).

The fragmentation predicates are those referencing the attributes *Month, Family* or *Channel*. In all our experiments, we assume that the cardinality of fact fragments is uniform.

The workload used in our experiments is given in [3] (due to the space constraint).

4.2 Experimental Results

Evaluation of the three options. In first experiment, we compare the quality of solutions (in terms of query processing cost reduction) produced by (a) *partitioning option (PO)*, (b) *join index option (JIO)* and (c) *partjoin option (PJO)*. For JIO, we assume that a join index exists for each query.

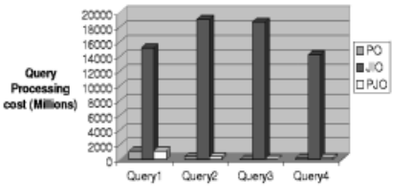

Fig. 3. Cost for best match queries

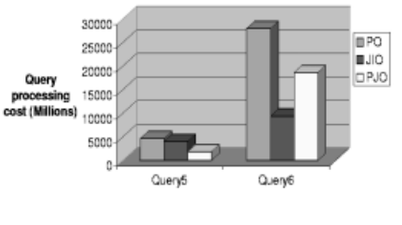

Fig. 4. Costs of partial and worst match queries

Figure 3 shows clearly that the partitioning option *outperform very well compared to join index option for best match queries* (queries 1 - 4, see [3]). Therefore, for this type of query, the PO is recommended. Note that for best match queries, the PO and PJO give the same performance, because for these type of queries, join indexes are not needed (see Section 2). For worst queries (query 6), the partitioning option performs badly compared to the join index and partjoin options (Figure 4). The utilization of join indexes improves the performance of worst match queries by at most 60% compare to the reduction obtained by PO. For the worst match queries, the join index option is recommended. We have also

[4] We suppose that products are grouped into four major families: child products, female products, male products, and mixed products.

observed that the partjoin option gives a better performance than other two options for partial match queries (query 5). Figure 4 shows that, the performance of partial match queries is improved by at most 50-55% compare the solution obtained by JIO and PO.

One of the interesting points that can be observed in Figure 3 is that when a query has selection predicates defined on all fragmented dimension tables, the*PartJoin* and partitioning options perform ideally.

Storage requirements of each option. To compute the storage requirements of each option, we suppose that indexes exist for each query (no storage constraint). We compute the storage cost for JIO by summing up the storage cost of each index used on evaluating the queries in the workload. For the PJO, we compute the size of join indexes selected by the index module.

In Figure 5, we observe that the partitioning option is much more efficient in terms of storage requirements (we can guarantee performance for free!), whereas the join index option requires a lot of space. The partjoin option is in between. By using the partjoin option, we save more at most 55% of space required for join indexes. Therefore the saved space can be used for other structures that need space (materialized views, indexes on single tables, etc.).

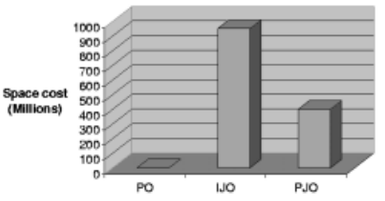

Fig. 5. Space requirements for each option

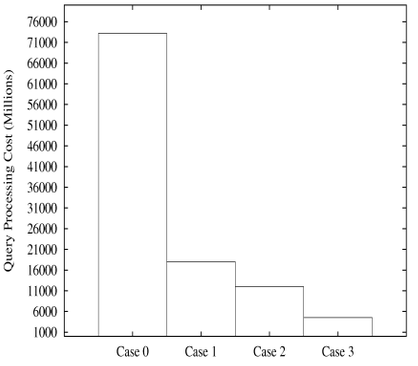

Fig. 6. The effect of fragmented dimension tables

The effect of number of fragmented dimension tables. In this experiments, we study the effect of the number of dimension tables participating on the fragmentation process. To do this, we will concentrate on the following cases: *Case 0:* all dimension tables are unpartitioned, in this case we consider the JIO, *Case 1:* only one dimension table is partitioned, for example, *Time* and the fact table *Sales* is fragmented based on that table, *Case 2:* two dimension tables are

partitioned, for example, *Time* and *Product*. Therefore the fact table is partitioned based on fragmentation schemas of these two tables, and *Case 3 :* the three dimension tables *Time*, *Channel* and *Product* are partitioned and similarly, the fact table is fragmented based on these tables.

We evaluate our queries by considering each case and we compute the cost of executing all queries. From Figure 6, we observe that the number of partitioned dimension tables has a great impact on reducing the query processing cost. As we see, the PJO performance increases while the number of fragmented dimension tables increases.

5 Conclusion

In this paper we have introduced a new data warehouse tuning strategy called *PartJoin* which combines data partitioning and join indexes for executing efficiently a set of OLAP queries and for reducing the storage requirements significantly. *PartJoin* exploits the similarities between join indexes and data partitioning. Data partitioning avoids the use of join indexes and therefore guarantees an economical utilization of space. But with the data partitioning, the number of fact fragments can be very large and difficult to maintain. Therefore the proposed tuning strategy finds a compromise between the utilization of data partitioning and join indexes. To satisfy this compromise, we have developed a data partitioning algorithm for decomposing dimension tables and fact table that guarantees that the number of fact fragments are less than a threshold representing the maximal number of fragments that the data warehouse administrator can maintain. From our results, it appears that *PartJoin* method gives better performance than the partitioning option and the join index option for certain class of queries under the storage and data maintenance constraints.

We believe that our tuning method is directly applied to commercial databases with a little effort. PartJoin method does not presently take into account the maintenance cost. In the future work, we will extend our system in that direction by incorporating this cost on the merging process.

References

1. L. Bellatreche, K. Karlapalem, and Q. Li. Evaluation of indexing materialized views in data warehousing environments. *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery (DAWAK'2000)*, pages 57–66, September 2000.
2. L. Bellatreche, K. Karlapalem, and M. Mohania. What can partitioning do for your data warehouses and data marts? *Proceedings of the International Database Engineering and Application Symposium (IDEAS'2000)*, pages 437–445, September 2000.
3. L. Bellatreche, M. Schneider, M. Mohania, and B. Bhargava. Partjoin: An efficient storage and query execution for data warehouses. *extended version, available at http://www.imerir.com/∼ladjel*, January 2002.

4. S. Chaudhuri and V. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. *Proceedings of the International Conference on Very Large Databases*, pages 146–155, August 1997.
5. OLAP Council. Apb-1 olap benchmark, release ii. *http://www.olapcouncil.org/research/bmarkly.htm*, 1998.
6. S. Guo, S. Wei, and M. A. Weiss. On satisfiability, equivalence, and implication problems involving conjunctive queries in database systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):604–612, August 1996.
7. P. O'Neil and D. Quass. Improved query performance with variant indexes. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 38–49, May 1997.
8. M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems : Second Edition*. Prentice Hall, 1999.