Efficient Materialization and Use of Views in Data Warehouses

Márcio Farias de Souza
marcio@dsc.ufpb.br

Marcus Costa Sampaio
sampaio@dsc.ufpb.br

Federal University of Paraíba
Department of Computer Science
58.109-970 Campina Grande Paraíba Brazil

Abstract

Given the complexity of many queries over a Data Warehouse (DW), it is interesting to precompute and store in the DW the answer sets of some demanding operations, so called materialized views. In this paper, we present an algorithm, including its experimental evaluation, which allows the materialization of several views simultaneously without losing sight of processing costs for queries using these materialized views.

1 Introduction

The need to provide integrated access to multiple and heterogeneous databases, as well as other sources of information, has become one of the priority areas of database research. One of the solutions to this problem is based on the so-called *mediator* approach [8], as follows:

1. Accept a query, determine the set of sources of information to answer the query and create the appropriate sub-queries for each source;
2. Obtain the results from the sources, translate, filter and integrate them, and return the final results to the user.

This requires implementation of an efficient mediator, which is a practically impossible task, owing to the scattering of information.

A more viable alternative would be an *anticipated* approach to the integration of data. This approach would function as follows:

1. The information from the sources is periodically extracted, translated, filtered, integrated and aggregated and then stored in a centralized repository, a *Data Warehouse (DW);*
2. When a query is submitted, it is evaluated directly in the DW, without accessing, therefore, the primitive sources of information.

Note that the DW is not actually updated during a user session in order to maintain temporal consistency [3], i.e., during a (possibly long) interactive session the DW should not change its state.

A DW is normally a very large database, even when its information aggregation level is high, in fact it can maintain several years of historical information stored. Historical series are an essential requirement of On-line Analytical Processing (OLAP) applications developed to help decision-making processes [3]. The reduction of the costs of OLAP queries is an important research objective.

1.1 A Typical OLAP Session

Decision Support Systems are those which have the capacity *to alert* the users (in general, high and middle managers) about the occurrence of ***exceptions***, giving them the means to find root causes. OLAP queries are inserted in this context: they are interactive system-user activities in which the user may want more details (*drill down*) or less details (*drill up*) of a report showing business trends. Trends are generally comparisons of data at a certain level of aggregation, throughout time. The user may still want to logically combine interlinked reports (*drill across*) [3].

Consider the following table of a relational DW *Fact(Product#, Store#, Time#, Sale):* Product#, Store# and Time# are respectively the keys of the dimension tables *Product*, *Store* and *Time*, while Sale is an *additive* attribute. Suppose that the aggregation level of the Fact table is *day*. The dimension table Time contains the semantics of each value of Time (week-day, weekend, week, month and other attributes). In this way the Product and Store dimension tables describe, respectively, the semantics of Product and Store.

A user wanting to analyze the sales of some products over a number of months can initiate an interaction with the system. If this type of analysis were quite common, a materialized view of the Fact table could be created *previously*, *PST(Product#, Store#, Time#, Monthly-sale)*, *sorted* by Product, Store and Time. As PST would still be very large, other materialized views of the Fact table could also be created, *PS(Product#, Store#, Sale-in-the-Period)* and *P(Product#, Sale-in-the-Period).* This sequence of interactions — query of the PST, and afterwards of the P (drill up), followed by the PS (drill down), returning to the PST (drill down) — would be efficient.

Unfortunately, given the nature of an OLAP session, user activities are not completely foreseeable. Coming back to our example, the user can, in the middle of a session, direct his attention to stores. The materialized views PST, PS and P do not favor the new OLAP operations desired by the user.

To speed up programming, the *group bys* corresponding to each of the combinations in P, S and T could be computed by a single operator, *cube by* [1]. But the question is still not totally solved: are all the possible *group bys* necessary? Is it better to store PS or SP, taking into account the relative frequency and importance of the queries to the DW? Are there constraints as to the time necessary for the creation of all the views?

1.2 Contributions and Organization of the Paper

We are dealing with the following combinatorial problem: what is the *best global cost* of materializing a set of views (in short, the computing of all or part of the *group bys* of a *cube by* operation), taking into consideration the possibility of creating various simultaneous views and without loosing sight of the cost of queries which will use these views? The advantages of reducing the costs of a query are obvious. Reducing the costs of creating views is also important. Even though this process generally operates in *batch* mode, it can take several hours, and, during this time, the DW would remain unavailable to users.

In section 3 of the paper, we shortly present an algorithm which yields a solution to the combinatorial problem mentioned above[1]. We will show that the best solution is not always possible, as there could be irreconcilable conflicts between the two objectives: efficient view creation *versus* efficient queries to the DW. In this case, the algorithm will seek to find a solution as close as possible to the best global cost for view creation. We include a glossary (section 2) before section 3, to precisely describe various terms which are used in explaining the algorithm. Three additional sections complete the article: section 4 presenting experimental evaluation of the algorithm, section 5 about related works, and section 6 presenting conclusions.

2 Glossary

In this section we give the meaning of various terms and expressions used throughout the article. Some of these expressions ("clustered table", "sorted query" and "clustered query") are not commonly used, while the rest of the terms and expressions are frequently found in the DW literature.

Clustered Table In such a table the records are physically clustered by some clustering criteria involving one or more of its attributes.

Clustered Query A sorted query to a clustered table in which the sorting criteria and the clustering coincide. The query processing cost is optimal.

Cube by An operator associated with a set of *n* cube dimensions which computes 2^n *group bys* for each of the combinations of the *n* dimensions, including the *group by null*. For example*, select ... from table ... cube by d1, d2* will result in a computation of $2^2 = 4$ *group bys*, *group by d1, d2; group by d1; group by d2* and *group by null.*

Cuboid Each of the combinations of the dimensions of a *cube by* operator. For *cube by d1, d2,* the *cuboids* are *d1d2, d1, d2, null.*

Data Cube Synonym of a *multidimensional* database seen as a *n-dimensional cube*, or simply *cube*; each dimension is an aggregation criterion, and each *cell* of the *cube* contains *numeric measures* or *facts* associated with a value for each *cube* dimension. A *cube* can be represented by a relational schema called *star schema.*

Dimension Table A table containing a dimension key and the attributes to describe the semantics of each key value.

Fact Table A table characterized by a key composed of foreign keys for each dimension, and of attributes which are generally *additive.*

Materialized View The name given to a clustered table corresponding to a *cuboid* and stored in a DW. In the rest of this paper, when we refer to a *cuboid* we will be thinking of the materialized view associated with it.

Relational Data Warehouse A cube with a star schema.

Sorted Query A query with an *order by* clause.

Star Schema A representation of a *cube*, composed of a *fact table* (the star) and *dimension tables* related to the fact table (the star satellites).

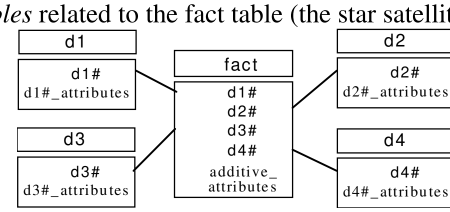

Figure 1: Star schema.

Fig. 1 is an illustration of a star schema with the fact table *fact* and four dimension tables *d1,d2,d3* and *d4.*

3 Quasi-optimal Algorithm

1 - Full details about the algorithm can be found in *http://www.dsc.ufpb.br/~lsi/sbbd98-sigmod.ps*

We propose an algorithm (referred to as *quasi-optimal*) which either optimizes the global cost of computing a *cube by* or approximates, in a strong sense, the computed global cost to the optimal cost. The computation may be *total* (all of the *cuboids*) or *partial* (some *cuboids*). With respect to the global cost computation, the algorithm considers two costs: the cost of direct and indirect derivation of a *cuboid* from another *cuboid* — these costs will be explained up further on — , taking into account the use made of *cuboids* by the users.

Derivation Graph

The algorithm's input is a derivation graph (G, π), where each vertex $v_k \in$ G is a *cuboid* of a *cube by*. The edges in π are *derivation-oriented edges* (v_i, v_j), i $\neq$ j, from v_i to v_j, if v_j has all the dimensions of v_i, less one. Each edge is labelled with two derivation costs: *Direct* and *Indirect.* The cost D(v_i,v_j) means that v_j is a prefix of v_i; consequently v_j can be computed directly from, and simultaneously with, v_i. On the other hand, if v_j is not a prefix of v_i, then the cost of computing v_j from v_i is indirect, I(v_i,v_j), meaning that v_i should be re-sorted to create v_j directly from it. It is then obvious that the direct cost *D* to derive a *cuboid* is always less than the indirect cost *I*. This derivation graph is similar to the hierarchy of *group by* operations in [7]

Besides the derivation graph, one must indicate a list of the sorted queries related to some or all of the *cuboids*. There could be more than one sorted query to a given *cuboid*.

See the derivation graph in Fig. 2. Consider *cuboid* C2; the edges from C2 (a *deriver cuboid*) to *cuboid*s C5 and C6 *(derived cuboids*), respectively, are derivation edges which indicate that C5 and C6 are derived from C2, that is, C5 (C6) has all the dimensions of C2, less one. These edges are labelled (*D, I*). The cost of deriving C5 (C6) is D, when C5 (C6) is a prefix of C2; when C5 (C6) is not a prefix of C2, then the cost is *I*. There could be more than one sorted query to C2.

We make a fairly reasonable hypothesis that, given a deriver *cuboid* and all which are derived from it, all the deriver–derived edges have the same *D* and *I* costs. Thus, for example, for the edges from C2 to C5 and C6, the value of (*D, I*) is common to both.

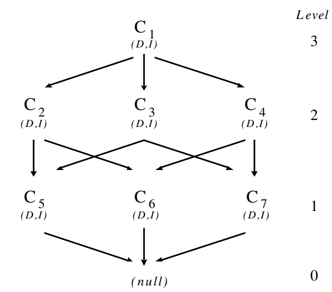

Figura 2: A derivation graph, input to the quasi-optimal algorithm.

The level *k* of a derivation graph represents all the *cuboids* with *k* dimensions. Note that given a *n*-dimensional *cube by*, the number of levels of the graph (G, π) is always *n+1* (level *0*, level *1*, ..., level *n*). The level *0* represents the *cuboid null.* The level *n* represents the deriver *cuboid* of the *cuboids* of other levels, direct or indirectly. By examining the number of levels for the graph in Fig. 2, it can be concluded that the cube has three dimensions.

For lack of space, we will not detail how to estimate the derivation costs from a *cuboid*. We will only say that they are dependent on the *cuboid* size (actually, the size of the result of the *group by* associated with it) and on the characteristics of the hardware (memory size and processing speed). [7] presents a procedure for estimating the size of a *cuboid*.

Quasi-optimal Derivation Graph

One of the algorithm's goals is to guarantee that queries supplied remain clustered to the corresponding derivation graph *cuboids*. The other goal is to explore to the full the possibility of deriving *cuboids* directly, or simultaneously, from others, aiming to diminish the creation costs of materialized views. Ideally, both goals would be fully achieved. Unfortunately, these goals may be conflicting.

The algorithm's output is a sub-graph called *quasi-optimal derivation graph,* generated from the *derivation graph* given as input to the algorithm, in which the global sum of all of the costs *D* and *I* is *minimal*, or is a value as close as possible to the minimum, in relation to the two goals being pursued. In this graph, each *cuboid*, with the exception of the highest level *cuboid*, is connected to a *single* higher level *cuboid* (the deriver *cuboid*), the edge being labelled as *D* or *I*. To distinguish between a *D* value and an *I* value*,* the edges with costs *D* are solid edged while the edges with costs *I* are dashed. The deriver *cuboid* of all of the other *cuboids*, direct or indirectly, is a *cuboid* in the highest level of the quasi-optimal derivation graph. An example of a quasi-optimal derivation graph (there are others) for the derivation graph in Fig. 2 is shown in Fig. 3.

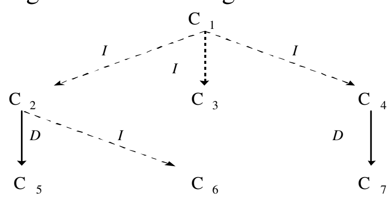

Figura 3: A quasi-optimal derivation graph.

Imagine four sorted queries, to C1, C2, C5 and C6. If the two goals of the algorithm had been fully achieved, then among all the related combinations of edges in Fig.2, the combination of edges in Fig.3 would have a minimal cost (in other words, the sum of all edge costs would be minimal), and the queries would also be clustered to their respective *cuboids*. In the case of a conflict between the two objectives, the queries would be still clustered to their *cuboids*, but the sum of all edge costs would not be exactly minimal.

To understand the way in which a quasi-optimal derivation graph is created, we need to describe the algorithm.

Algorithm Description

In order to obtain a quasi-optimal derivation graph from a derivation graph (G, π), the quasi-optimal algorithm uses iteration; each iteration considers two consecutive levels of (G, π), from level $k = 0$ to level $k = n-1$, where n is the number of dimensions of a *cube by* represented by (G, π). For each level k, the algorithm finds the mininal cost to derive level k from level $k+1$. This problem is solved through an algorithm implementing the *Hungarian method* [4], which determines what is called minimal-cost-matching in an directed bipartite graph.

With the help of the quasi-optimal algorithm's *Minimal_Cost_Matchings* procedure, which implements the Hungarian method, we can arrive at a minimal cost matching, as shown in Fig. 4, between the two sets of *cuboids* at levels *1* and *2*, respectively, of the derivation graph in Fig.2.

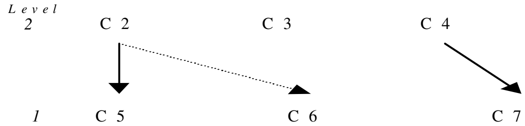

Figure 4: A minimal cost matching for the *cuboids* at levels *1* and *2*, respectively, of the derivation graph in Fig.2.

As we can see, this matching is not unique, and the *Minimal_Cost_Matchings* procedure actually finds all matchings. The existence of several minimal-cost matchings allows us to exhaustively test the alternatives in order to resolve conflicts between the order of the *cuboids* and the order of the queries.

Detection and Solution of Conflicts

There exist two kinds of conflict which are detected by the quasi-optimal algorithm's *Conflict_Detector* procedure: the first is a conflict among the several sorted queries to the same *cuboid*; the second type of conflict is between the order of the *cuboids* and the order of the queries.

To illustrate the first type of conflict and its solution by the *Conflict_Resolver* procedure, imagine two queries to *cuboid* C2. It is clear that only one of them can be clustered with C2. The solution to the conflict, as determined by the *Conflict_Resolver* procedure, is the following: the procedure chooses the query which is clustered with C2, maintaining the minimal cost matching among the *cuboids* at levels 1 and 2. With respect to the other query, a careful choice of indices to access C2 can make its processing cost acceptable [6].

For the second type of conflict and its solution, consider that the orders of C4 and the query submitted to it are different. In this case, C4 is re-sorted to cluster it with the query to it, and consequently the direct derivation cost of C7 is substituted by this *cuboids*' indirect derivation cost (that is, the edge from C4 to C7 changes from solid to dashed). As a result, the matching between the *cuboids* at levels *1* and *2* is not now of minimal cost. Note that the solution for these conflicts will always privilege the queries.

It is important to point out that a conflict of the second type is still frequently solved by maintaining the minimal cost matching. Observe again the bipartite graph in Fig.4: we could change the order of C2, and as such invert the derivation costs of C5 (now a dashed edge) and of C6 (now a solid edge) — it is obvious that the sum of all costs will remain unaltered.

The *Conflict_Detector* and *Conflict_Resolver* procedures are applied to each matching generated by the *Minimal_Cost_Matchings* procedure. The final solution for levels $k+1$ and k is the one closest to the minimal cost matching.

3.1 Partial Computing of a *Cube by* Operation

Often, a DW administrator is interested in materializing only some views, among all of the views corresponding to each of the *cuboids* of a *cube by*. If this is the administrator's option, the algorithm's output is a *partial* quasi-optimal derivation graph, constructed from the quasi-optimal derivation graph, when the option is a total *cube by* computation. This partial graph consists of *cuboids* corresponding to the desired materialized views, as well as the intermediate *cuboids* necessary to create them, along with the appropriate edges. The following refinement consists of changing the dashed edges of the partial graph into solid edges, wherever possible. The example in the following sub-section gives additional details about this operation of the algorithm.

3.2 An Application Example of the Algorithm

Take four dimensions, Product (P), Promotion (Pr), Store (S) and Time (T), and six very frequently sorted queries *pspr*, *sprp*, *ps*, *spr*, *p* and *s* to a relational DW. Also consider the table *Fact* (*P#, Pr#, S#, T#, additive_attributes*) and the dimension tables *Product, Promotion, Store* and *Time*. The DW administrator then wishes to materialize the views PPrS, PS, PrS, P and S, so as to guarantee good performance for the queries.

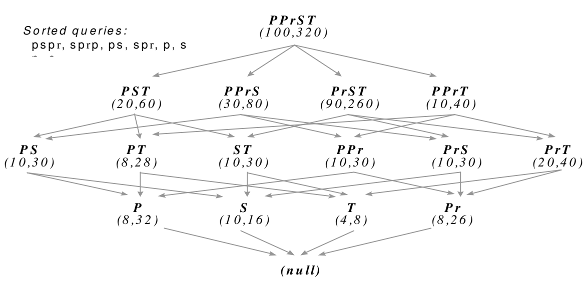

Figure 5: Derivation graph for *Cube by* P, Pr, S, T.

The input to the quasi-optimal algorithm is the derivation graph in Fig. 5, with the list of six queries.

Each pair of numbers is of the type (D, I). Thus, for example, (20, 60) under the *cuboid* PST means that the direct cost of derivating PS, or PT, or ST from PST is 20, while the indirect cost of derivating PS, or PT, or ST from PST is 60.

Figure 6 is a quasi-optimal derivation graph created by the algorithm, whose semantics are the optimal cost plan for the construction of all materialized views concerning the *cube by* P, Pr, S, T operation, taking into account the given queries and the possibility of simultaneously creating several views. Among all alternatives for the construction of optimal plans, the algorithm chooses this one, which privileges the *pspr* query instead of *sprp*, and in which the queries *pspr*, *ps*, *spr*, *p* and *s* are clustered respectively with PSPr, PS, SPr, P and S.

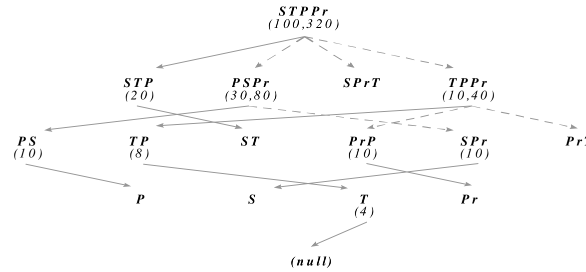

Figure 6: The best plan to materialize the views of *Cube by* P, Pr, S, T.

As an example of a plan which is not exactly optimal, add a sorted query *sprpt* to the given queries. In this case, we would have a *cuboid* SPrPT and the cost of deriving STP of SPrPT would be indirect (320, instead of 100), and the global cost would not be minimum.

From the graph in Fig. 6, the algorithm creates a partial graph as in Fig. 7 where the *cuboids* correspond to the given queries. An intermediate *cuboid* necessary to the first ones is also shown.

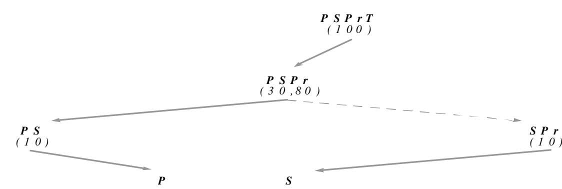

Figure 7: Graph to partial computing of *Cube by* P, Pr, S, T.

Observe that the intermediate *cuboid* PSPrT is necessary to simultaneously derive the *cuboids* PSPr, PS and P.

4 Experimental Evaluation

The critical point in terms of the quasi-optimal algorithm's performance is the *Minimal_Cost_Matchings* procedure which implements the Hungarian method to compute the minimal cost matchings between two sets of vertices of a bipartite graph. The complexity of the procedure, for two consecutive levels k and $k+1$ of the derivation graph, is cubic, $O(n^3)$, where n is the number of vertices at level $k+1$.

The test involved the materialization of 5 views PSPr, PS, SPr, P and S, with and without the help of the cost plan in Fig. 7. The hardware for the test was the IBM RS/6000, with 32 MB of memory, while the software used for table creation was the Postgresql DBMS [5]. Assuming that all record fields in the Fact table are 4 bytes, the tests were made for the Fact table with 54,750,000 records (≈ 1,1 GB).

Firstly, we created a file PSPrT, sorted by P, S, Pr, and T, with the contents of the Fact table. From this file, a Postgresql clustered table, PSPrT, was created. The remaining Postgresql tables, PSPr, PS, SPr, P and S, were created according to the quasi-optimal derivation graph in Fig. 7. Let Time1 be the total creation time for these 5 tables.

For the tests without the Fig. 7 plan, a Postgresql clustered table, PPrST, was initially created. Afterwards, each of the 5 tables, PSPr, PS, SPr, P and S were created separately. Let Time2 be the total creation time for these 5 tables.

Since Time2 is more than 8 times larger than Time1, the test unambiguously shows that the creation of materialized views according to the plan exposed is vastly superior.

5 Related Works

A large DW research project is being developed at Stanford University, USA. With respect to the *cube by* operator and materialized views, refer to the work by [2], who presents an algorithm to decide which *group bys* should be pre-computed and indexed. This research, however, does not deal with the optimization of related *group bys*.

[7] and [6], in which the last complements the first, are related to one another. The focus of [7] is the *cube by* operator, which may be seen as a hierarchy of *group by* operations. It thus considers related *group bys*, but the performance of complex queries to a DW is outside its scope. On the contrary, [6] exclusively considers the efficiency of complex queries to DWs, supposing that the materialized views have already been created.

6 Conclusions

In this paper we present an algorithm for the efficient computing of multiple *group bys* of a *cube by*. The results of all or some of the *group by*s of a *cube by* should be stored in the Data Warehouse (materialized view process), aiming for their later use in queries to the Data Warehouse.

However, it would not be worthwhile to materialize views without considering the queries that could use them. Without this care, it could happen that (1) a view would be used very little or not at all; and (2) that a view and a query that frequently uses it could have incompatible orders, requiring the sorting of very large temporary files, thus resulting in large query processing costs.

The algorithm takes the following two costs into account: the cost of creating a materialized view and the cost of processing a query to a Data Warehouse. It has as input a graph called *derivation graph* which indicates that the result of a *group by* may be derived *directly* or *indirectly* from another (that is, in the case where it *is not* or *is* necessary to re-sort the last to obtain the first), with their respective costs, as well as a list of frequent queries to the Data Warehouse which could use these views.

As output, the algorithm produces an *optimal*, or *quasi-optimal*, plan to create the views. In an optimal plan the view creation cost is mathematically the least in relation to all the other costs, and the view orders are compatible with the query orders; in a quasi-optimal plan, the cost is closest to the least cost, and the view orders are still compatible with the query orders.

The tests with the algorithm unambiguously show its superior performance when compared with *ad hoc* view materialization processes.

Acknowledgements This work is supported by CAPES and CNPq, Brazil. We thank anonymous reviewers of the XIII Brazilian Symposium on Database Systems for the careful reading of the full paper and the fruitful comments.

References

[1] J. Gray et al. *Data Cube: a Relational Operator Generalizing Group-by, Cross-Tab and Sub-Totals*. Proc. of the 12^{th} Int. Conf. On Data Engineering, 152-159, 1996.

[2] V. Harinarayan et al. *Implementing Data Cubes Efficiently*. Proc. of the 1996 ACM-SIGMOD Conference, 1996.

[3] R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, Inc., 1996.

[4] C.H.Papadimitriou and K.Steiglits. Combinatorial Optimization: Algorithms and Complexity, chapter 11, pages 247-254, 1982

[5] Postgres Programmer's Guide. http://www.postgresql.org

[6] B. Salzberg e A. Reuter. *Indexing for Aggregation*. Working Paper, 1996.

[7] S. Sarawagi et al. *On Computing the Data Cube*. Research Report, IBM Almaden Research Center, 1996.

[8] G. Wiederhold. *Mediators in the architecture of future information systems.* IEEE Computer, 25(3):38-49, March 1992.