Multi-query Optimization for On-Line Analytical Processing

Panos Kalnis and Dimitris Papadias

Technical Report HKUST-CS01-12, November, 2001

Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{kalnis, dimitris}@cs.ust.hk

Abstract

Multi-Dimensional Expressions (MDX) provide an interface for asking several related OLAP queries simultaneously. An interesting problem is how to optimize the execution of an MDX query, given that most data warehouses maintain a set of redundant materialized views to accelerate OLAP operations. A number of greedy and approximation algorithms have been proposed for different versions of the problem. In this paper we evaluate experimentally their performance using the APB and TPC-H benchmark concluding that they do not scale well for realistic workloads. Motivated by this fact, we develop two novel greedy algorithms. Our algorithms construct the execution plan in a top-down manner by identifying in each step the most beneficial view, instead of finding the most promising query. We show by extensive experimentation that our methods outperform the existing ones in most cases.

Contact Author: Dimitris Papadias
Tel: ++852-23586971 http://www.cs.ust.hk/~dimitris/
Fax: ++852-23581477 E-mail: dimitris@cs.ust.hk

The Hong Kong University of Science & Technology
Technical Report Series
Department of Computer Science

1. Introduction

Effective decision-making is vital in a global competitive environment where business intelligence systems are becoming an essential part of virtually every organization. The core of such systems is a data warehouse, which stores historical and consolidated data from the transactional databases, supporting complicated ad-hoc queries that reveal interesting information. The so-called On-Line Analytical Processing (OLAP) [2] queries typically involve large amounts of data and their processing should be efficient enough to allow interactive usage of the system.

A common technique to accelerate OLAP is to store some redundant data, either statically or dynamically. In the former case, some statistical properties of the expected workload are known in advance. The aim is to select a set of views for materialization such that the query cost is minimized while meeting the space and/or maintenance cost constraints, which are provided by the administrator. [8, 5, 7] describe greedy algorithms for the view selection problem. In [6] an extension of these algorithms is proposed, to select both views and indices on them. [1] employs a method which identifies the relevant views of a lattice for a given workload. [22] uses a simple and fast algorithm for selecting views in lattices with special properties.

Dynamic alternatives are exploited in [18, 3, 11]. These systems reside between the data warehouse and the clients and implement a disk cache that stores aggregated query results in a finer granularity than views.

Most of these papers assume that the OLAP queries are sent to the system one at a time. Nevertheless, this is not always true. In multi-user environments, many queries can be submitted concurrently. In addition, the API proposed by Microsoft [13] for Multi-Dimensional Expressions (MDX), which becomes the de-facto standard for many products, allows the user to formulate multiple OLAP operations in a single MDX expression. For a set of OLAP queries, an optimized execution plan can be constructed to minimize the total execution time, given a set of materialized views. This is similar to the multiple query optimization problem for general SQL queries [16, 19, 20, 17], but due to the restricted nature of the problem, better techniques can be developed.

[26] was the first work to deal with the problem of multiple query optimization in OLAP environments. They designed three new join operators, namely: *Shared scan for Hash-based Star Join*, *Shared Index Join* and *Shared Scan for Hash-based and Index-based Star Join*. These operators are based on common subtask sharing among the simultaneous OLAP queries. Such subtasks include the scanning of the base tables, the creation of hash tables for hash based joins and the filtering of the base tables in the case of index based joins. Their results indicate that there are substantial savings by using these operators in ROLAP systems. In the same paper they propose greedy algorithms for creating the optimized execution plan for an MDX query, using the new join operators.

In [12] three versions of the problem are examined: In the first one, all the simple queries in an MDX are assumed to use hash based start join. A polynomial approximation algorithm is designed, which delivers a plan whose evaluation cost is $O(n^{\varepsilon})$ times worse than the optimal, where n is the number of queries and $0 < \varepsilon \leq 1$. In the second case, all simple queries use index-based join. They present an approximation algorithm whose output plan's cost is n times the optimal. The third version is more general since it is a combination of the previous ones. For this case, a greedy algorithm is presented. Exact algorithms are also proposed, but their running time is exponential, so they are practically useful only for small problems.

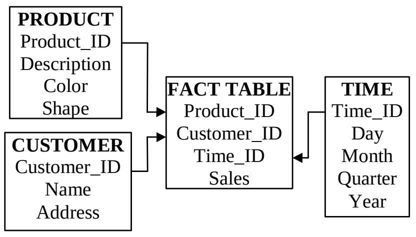
In this paper we use the TPC-H [24] and APB [15] benchmarks in addition to a 10-dimensional synthetic database, to test the performance of the above algorithms under realistic workloads. Our experimental results suggest that the existing algorithms do not scale well when more views are materialized. We observed that in many cases when the space for materialized views increases, the execution cost of the plan derived by the optimization algorithms is higher than the case where no materialization is allowed!

Motivated by this fact, we propose a novel greedy algorithm, named *Best View First* (*BVF*) that doesn't suffer from this problem. Our algorithm follows a top-down approach by identifying the most beneficial view in each iteration, as opposed to finding the most promising query to add to the execution plan. Although the performance of *BVF* is very good in the general case, it deteriorates when the number of materialized views is small. To avoid this, we also propose a multilevel version of *BVF* (*MBVF*). We show by extensive experimentation that our methods outperform the existing ones in most realistic cases.

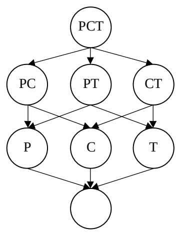
The rest of the paper is organized as follows: In section 2 we introduce some basic concepts and we review the work of [26] and [12]. In section 3 we identify the drawbacks of the current approaches and in section 4 we describe our methods. Section 5 presents our experimental results while section 6 summarizes our conclusions.

2. Background

For the rest of the paper we will assume that the multi-dimensional data are mapped on a relational database using a star schema [10]. Let D_1, D_2, …, D_n be the dimensions (i.e. business perspectives) of the database, such as *Product*, *Customer* and *Time*. Let M be the measure of interest; *Sales* for example. Each D_i table stores details about the dimension, while M is stored in a fact table F. A tuple of F contains the measure plus pointers to the dimension tables (figure 1a).



(a) The star schema



(b) The data-cube lattice

Figure 1: A data warehouse schema. The dimensions are *Product*, *Customer* and *Time*

There are $O(2^n)$ possible group-by queries for a data warehouse with n dimensional attributes. A detailed group-by query can be used to answer more abstract aggregations. [8] introduces the search lattice L, which represents the interdependencies among group-by's. L is a directed graph whose nodes represent group-by queries. There is a path from node u_i to node u_j if u_i can be used to answer u_j (figure 1b).

```
NEST ({Venkatrao, Netz}, {USA_North.CHILDREN, USA_South, Japan})
ON COLUMNS {Qtr1.CHILDREN, Qtr2, Qtr3, Qtr4.CHILDREN}
ON ROWS
CONTEXT SalesCube
FILTER (Sales, [1991], Products.ALL)
```

Figure 2: A multidimensional expression (MDX)

A multidimensional expression (MDX) provides a common interface for decision support applications to communicate with OLAP servers. Figure 2 shows an example MDX query, taken from the Microsoft document [13]. MDX queries are independent from the underline engine thus they do not contain any join attributes or conditions. In terms of SQL statements, we identify the following six queries:

1. The total sales for Venkatrao and Netz in all states of USA_North for the 2nd and 3rd quarters in 1991.
2. The total sales for Venkatrao and Netz in all states of USA_North for the months of the 1st and 4th quarters in 1991.
3. The total sales for Venkatrao and Netz in region USA_South for the 2nd and 3rd quarters in 1991.
4. The total sales for Venkatrao and Netz in region USA_South for the months of the 1st and 4th quarters in 1991.
5. The total sales for Venkatrao and Netz in Japan for the 2nd and 3rd quarters in 1991.
6. The total sales for Venkatrao and Netz in Japan for the months of the 1st and 4th quarters in 1991.

Therefore, an MDX expression can be decomposed into a set Q of group-by SQL queries. We need to generate an execution plan for the queries in Q, given a set of materialized views, such that the total execution time is minimized. The group-by attributes of the queries usually refer to disjoint regions of the data-cube [4] and the selection predicates can be disjoint. These facts complicate the employment of optimization techniques for general SQL queries [16, 19, 20, 17] while more suitable methods can be developed due to the restricted nature of the problem.

Recall that we assumed a star schema for the warehouse. The intuition behind optimizing the MDX expression is to construct subsets of *Q* that share star joins. Usually, when the selectivity of the queries is low, hash-based star joins [23] are used; otherwise, the index-based star join method [14] can be applied.

[26] introduced three shared join operators to perform the star joins. The first operator is the *shared scan for hash-based star join*. Let q_1 and q_2 be two queries which can be answered by the same materialized view *v*. Consequently they will share some (or all) of their dimensions. Assume that both queries are non-selective so hash-based join is used. To answer q_1 we construct hash tables for its dimensions and we probe each tuple of *v* against the hash tables. Observe that for q_2 we don't need to rebuild the hash tables for the common dimensions. Furthermore, only one scanning of *v* is necessary. Consider now that we have a set *Q* of queries all of which use hash-based star join and let *L* be the lattice of the data-cube and *MV* be the set of materialized views. We want to assign each $q \in Q$ to a view $v \in MV$ such that the total execution time is minimized. If *v* is used by at least one query, its contribution to the total execution cost is:

$$t_{MV}^{hash}(v) = Size(v) \cdot t_{I/O} + t_{hash\_join}(v)$$

where *Size(v)* is the number of tuples in *v*, $t_{I/O}$ is the time to fetch a tuple from the disk to the main memory, and $t_{hash\_join}(v)$ is the total time to generate the hash tables for the dimensions of *v* and to perform the hash join. Let *q* be a query that is answered by $v \equiv mv(q)$. Then the total execution cost is increased by:

$$t_{Q}^{hash}(q, mv(q)) = Size(mv(q)) \cdot t_{CPU}(q, mv(q))$$

where $t_{CPU}(q,v)$ is the time per tuple to process the selections in *q* and to evaluate the aggregate function. Let $MV' \subseteq MV$ be the set of materialized views which are selected to answer the queries in *Q*. The total cost of the execution plan is:

$$t_{total}^{hash} = \sum_{v \in MV'} t_{MV}^{hash}(v) + \sum_{q \in Q, mv(q) \in MV'} t_{Q}^{hash}(q, mv(q))$$

The problem of finding the optimal execution plan is equivalent to minimizing t_{total}^{hash} which is likely to be NP-hard. [12] provide an exhaustive algorithm which runs in exponential time. Since the algorithm is impractical for real life applications, they also describe an approximation algorithm. They reduce the problem to a directed Steiner tree problem and apply the algorithm of [25]. The solution is $O(|Q|^{\varepsilon})$ times worse than the optimal, where $0 < \varepsilon \leq 1$.

The second operator is the *shared scan index-based join*. Let q_1, $q_2 \in Q$ and let v be a materialized view which can answer both queries. Assume that each dimension table has bitmap join indices that map the join attributes to the relevant tuples of v, and the selectivity of both queries is high so the use of indices pays off. The evaluation of the join starts by OR-ing the bitmap vectors b_1 and b_2 which correspond to the predicates of q_1 and q_2 respectively. The resulting vector $b_{all} \equiv b_1 \vee b_2$ is used to find the set v' of matching tuples for both queries in v. The set v' is fetched in memory and each query uses its own bitmap to filter and aggregate the corresponding tuples.

The cost of evaluating a set Q of queries, where all queries are processed using index-based join, is defined as follows: Let $Q' \subseteq Q$ such as $\forall q_i \in Q'$, q_i can be answered by v. Let $R_i \subseteq v$ be the set of tuples that satisfies the predicates of q_i. The selectivity of q_i is $\sigma_i = |R_i|/Size(v)$. $R = \bigcup R_i$ is the set of tuples that satisfy the predicates of all queries in Q'. We define the selectivity of the set as $\sigma = |R|/Size(v)$. The cost of including v in the execution plan is:

$$t_{MV}^{index}(v) = \sigma \cdot Size(v) \cdot t_{I/O} + t_{index\_join}(v)$$

where $t_{index\_join}(v)$ is the total cost to build b_{all} and access it to select the appropriate tuples from v. Each query contributes to the total cost:

$$t_{Q}^{index}(q_i, mv(q_i)) = \sigma_i \cdot Size(mv(q_i)) \cdot t_{CPU}(q_i, mv(q_i))$$

Let $MV' \subseteq MV$ be the set of materialized views which are selected to answer the queries in Q. The total execution cost is:

$$t_{total}^{index} = \sum_{v \in MV'} t_{MV}^{index}(v) + \sum_{q \in Q, mv(q) \in MV'} t_{Q}^{index}(q, mv(q))$$

Again, we want to minimize t_{total}^{index}. In addition to an exact exponential method, [12] propose an approximate polynomial algorithm that delivers a plan whose execution cost is $O(|Q|)$ times the optimal.

The third operator is the *shared scan for hash-based and index-based start joins*. As the name implies, this is a combination of the previous two cases. Let $Q' \subseteq Q$ be a set of queries that can be answered by v. Q' is partitioned in two disjoint sets Q'_1 and Q'_2. The queries in Q'_1 share the hash-based star joins. For Q'_2 we use the combined bitmap to find the matching tuples for all the queries in the set, and afterwards the individual bitmaps to filter the appropriate tuples for each query. Observe that v is scanned only once. Its contribution to the total cost is:

$$t_{MV}^{comb}(v) = Size(v) \cdot t_{I/O} + t_{hash\_join}(v) + t_{index\_join}(v)$$

The contribution of $q_i \in Q'_1$ and $q_j \in Q'_2$ are given by $t_Q^{hash}(q_i, mv(q_i))$ and $t_Q^{index}(q_j, mv(q_j))$ respectively.

The combined case is the most interesting one in practice. Nevertheless, it is not possible to use directly the methods for hash-based-only or indexed-based-only star joins, because there is no obvious way to decide whether a query should belong to Q'_1 or Q'_2. In the next section we present the greedy algorithms that have been proposed for the combined case and analyze their performance under realistic workloads.

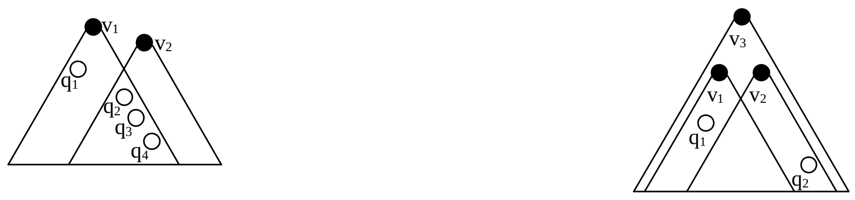
3. Performance of Existing Algorithms

[26] proposes three heuristic algorithms to construct an execution plan, namely *Two Phase Local Optimal* algorithm (*TPLO*), *Extended Two Phase Local Greedy* algorithm (*ETPLG*) and *Global Greedy* algorithm (*GG*). *TPLO* starts by selecting independently for each query q a materialized view v, such that the cost for q is minimized, and uses the SQL optimizer to generate the optimal plan for q. The second phase of the algorithm identifies the common subtasks among the individual plans and merges them using the three shared operators.

Merging the local optimal plans, does not guarantee a global optimal execution plan. To overcome this problem, the improved algorithm *ETPLG* constructs the global plan incrementally by adding queries in a greedy manner. For each query *q*, the algorithm evaluates the cost of executing it individually, and the cost of sharing a view with a query that was previously selected. The plan with the lowest cost is chosen. Since the order of inserting queries into the plan can greatly affect the total cost, the algorithm processes the queries in ascending *GroupByLevel* order (i.e. the queries on the top of the lattice, are inserted first). The intuition is that the higher the query is in the lattice, more chances exist that the query can share its view with the subsequent ones.

GG is similar to *ETPLG*, the only difference being that *GG* allows the shared view of a group of queries to change in order to include a new query, if this leads to lower cost. The experimental evaluation indicates that *GG* outperforms the other two algorithms. [12] proposes another greedy algorithm based on the cost of inserting a new query to a global plan (*GG-c*). The algorithm is similar to *ETPLG* but in each step it checks all unassigned queries and adds to the global plan the one that its addition will result to the minimum increase in the total cost of the solution. Their experiments show that in general the performance of *GG-c* is similar to *GG*, except when there is a large number of materialized views and a small number of queries. In this case *GG-c* performs better.

None of the above algorithms scales well when the number of materialized views increases. Next we present two examples that highlight the scalability problem. We focus on GG and GG-c due to their superiority; similar examples can be also constructed for *TPLO* and *ETPLG*. Figure 3a shows an instance of the multiple query optimization problem where $\{v_1, v_2\}$ is the set of materialized views and $\{q_1,\ldots, q_4\}$ is the set of queries (the same example is presented in [12]). We assume for simplicity that all queries use hash-based star join. Let $t_{I/O} = 1$, $t_{hash\_join}(v) = Size(v)/10$ and $t_{CPU}(q,v) = 10^{-2}$, $\forall$ *q*, *v*. *GG* will start by selecting q_1, since it has the lower *GroupByLevel*, and will assign it to v_1. Then q_2 is considered. If q_2 is answered by v_1 the cost is increased by 10000/100 = 100. If v_2 is used the cost is increased by 222. Thus q_2 is assigned to v_1. In the same way q_3 and q_4 are also assigned to v_1 resulting to a total cost of 10000 + 10000/10 + 4·10000/100 = 11400. It is easy to verify that the optimal cost is 11326 and is achieved by assigning q_1 to v_1 and the rest queries to v_2.



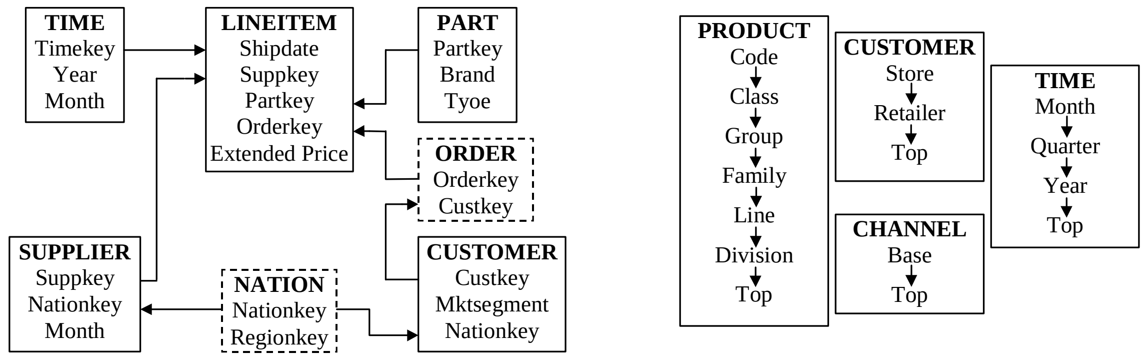
(a) $|v_1|$=10000, $|v_2|$=200 **(b)** $|v_1|$=100, $|v_2|$=150, $|v_3|$=200

Figure 3: Two instances of the multiple-query optimization problem.

Similar problems are also observed for *GG-c*. Assume the configuration of figure 3b. *GG-c* will search for the combination of queries and views that result to minimum increase of the total cost, so it will assign q_1 to v_1. At the next step q_2 will be assigned to v_2 resulting to a total cost of 277.5. Let v_3 be the fact table of the warehouse and v_1, v_2 be materialized views. If no materialization were allowed, *GG-c* would choose v_3 for both queries resulting to a cost of 224.

We observe that by materializing redundant views in the warehouse, we deteriorate, instead of improving, the performance of the system. Note that this is a drawback of the optimization algorithms and it is not due to the set of views that were chosen for materialization. To ensure this, assume that no shared join operator is available. Then, if v_1 and v_2 do not exist, the total cost is 2·222 = 444, but in the presence of v_1 and v_2 the cost drops to 277.5.

In order to evaluate this situation under realistic conditions, we employed datasets from the TPC-H benchmark [24], the APB benchmark [15] and a 10-dimensional synthetic database (SYNTH). We used a subset of the TPC-H database schema consisting of 14 attributes, as shown in figure 4a. The fact table contains 6M tuples. For the APB dataset, we used the full schema for the dimensions (figure 4b), but only one measure. The size of the fact table is 1.3M tuples. SYNTH dataset is a 10-dimensional database that models supermarket transactions, also used in [11]. The cardinality of each dimension is shown in table 1. The fact table contains 20M tuples. The sizes of the nodes in the lattice are calculated by the analytical algorithm of [21]. All experiments were run on an UltraSparc2 workstation (200MHz) with 256MB of main memory.



(a) The TPC-H database schema **(b)** The dimensions of APB

Figure 4: Details of the TPC-H and APB datasets

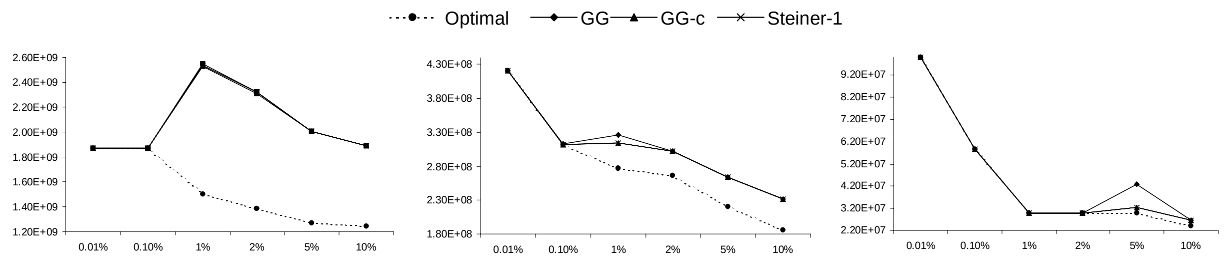
Dimension	1	2	3	4	5	6	7	8	9	10
Cardinality	100K	10K	800	365	54	5	200	1K	1.1K	70

Table 1: Cardinalities of the dimension tables for the SYNTH dataset

Since there is no standard benchmark for MDX, we constructed a set of 100 synthetic MDX queries. Each of them can be analyzed into 2 sets of 2 related SQL group-by queries (*q2_2* query set). Each dataset (i.e. SYNTH, TPC-H and APB) is represented by a different lattice, so we generated different query sets. We used this relatively small query set, in order to be able to run an exhaustive algorithm and compare the cost of the plans with the optimal one.

In our experiments we varied the available space for the materialized views (S_{max}) from 0.01% to 10% of the size of the full data cube (i.e. the case where all nodes in the lattice are materialized). For the SYNTH dataset, 1% of the data cube is around 186M tuples, while for the TPC-H and APB datasets, 1% of the data cube corresponds to 10M and 0.58M tuples respectively. We did not consider the maintenance time constraint for the materialized views, since it would not affect the trend of the optimization algorithms' performance. Without loss of generality, we used [8] *GreedySelect* algorithm to select the set of materialized views. We tested two cases: (i) every node in the lattice has the same probability to be queried and (ii) there is prior knowledge about the statistical properties of the queries. Although the output of *GreedySelect* is slightly different in the two cases, we found that the performance of the optimization algorithms is not affected considerably. In our experiments we used the second option.

We employed the shared operators and we compared the plans delivered by the optimization algorithms, against the optimal plan. All the queries use hash-based star join. We implemented the greedy algorithm of [26] (*GG*) and the one of [12] (*GG-c*). We also implemented the Steiner-tree-based approximation algorithm of [12] for hash-based queries (*Steiner-1*). We set $\varepsilon = 1$, since for smaller values of ε the complexity of the algorithm increases while its performance doesn't change considerably, as the experiments of [12] suggest. For obtaining the optimal plan, we used an exhaustive algorithm whose running time (for S_{max} = 10%) was 5300, 290 and 91 sec, for the SYNTH, the TPC-H and APB datasets respectively.



(a) SYNTH dataset **(b)** TPC-H dataset **(c)** APB dataset

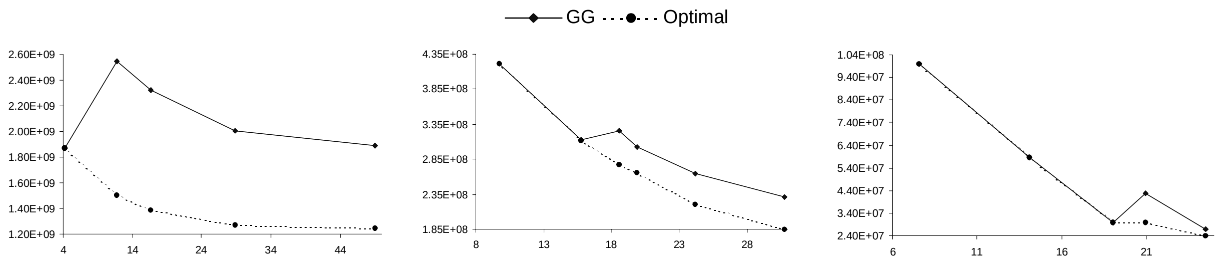
Figure 5: Total execution cost for *q2_2* query set

The behavior of *GG*, *GG-c* and *Steiner-1* is almost identical as shown in figure 5. Although the query set is too small to make safe conclusions, we can identify the instability problems. There is a point where the cost of the execution plan increases although more materialized views are available. Moreover, we observed that for the SYNTH dataset, when S_{max} varied from 1% to 5%, the execution cost of the plans delivered by *GG*, *GG-c* and *Steiner-1*, is higher in the presence of materialized views (i.e. we could achieve lower cost if we had executed the queries against the base tables). Although this case is highly undesirable, it does not contradict with the upper bound of the *Steiner-1* algorithm since in our experiments the cost of its plan was no more than 1.7 times worse than the optimal, which is within its theoretical bound. However, for the SYNTH dataset, if all queries are assigned to the top view, the cost is at most 1.66 times worse than the optimal.

The performance of the algorithms is affected by the *tightness* of the problem. Let *AVQ* be the average number of materialized views that can by used to answer each group-by query. We identify three regions:

1. The *high-tightness region* where the value of *AVQ* is small (i.e. very few views can answer each query). Since the search space is small, the algorithms can easily find a near optimal solution.
2. The *low-tightness region* where *AVQ* is large. Here, each query can be answered by many views, so there are numerous possible execution plans. Therefore there exist many near-optimal plans and there is a high probability for the algorithms to choose one of them.
3. The *hard-region*, which is between the high-tightness and the low-tightness regions. The problems in the hard region have a quite large number of solutions, but only few of them are close to the optimal, so it is difficult to locate one of these plans.

In figure 6 we draw the cost of the plan for *GG* and the optimal plan versus *AVQ*. For the SYNTH dataset the transition between the three regions is obvious. For the other two datasets, observe that for small values of *AVQ*, the solution of *GG* is identical to the optimal one. We can identify the hard region at the right part of the diagrams, when the trend for *GG* moves to the opposite direction of the optimal plan. Similar results were also observed for other query sets.



(a) SYNTH dataset **(b)** TPC-H dataset **(c)** APB dataset

Figure 6: Total execution cost versus *AVQ*

In summary, existing algorithms suffer from scalability problems, when the number of materialized views is increased. In the next section we will present two novel greedy algorithms, which have better behavior and outperform the existing ones in most cases.

4. Improved Algorithms

The intuition behind our first algorithm, named *Best View First* (BVF), is simple: Instead of constructing the global execution plan by adding the queries one by one (bottom-up approach), we use a top-down approach. At each iteration the most beneficial view *best_view* $\in$ *MV* is selected, based on a *savings* metric, and all the queries which are covered by *best_view* and have not been assigned to another view yet, are inserted in the global plan. The process continues until all queries are covered. Figure 7 shows the pseudocode of *BVF*.

The *savings* metric is defined as follows: Let $v \in MV$, and let $VQ \subseteq Q$ be the set of queries that can be answered by *v*. Let $C(q,u_i)$ be the cost of answering $q \in VQ$, by using $u_i \in MV$ and $C_{\min}(q) = \min_{1 \le i \le |MV|}(C(q,u_i))$ that of answering *q* by using the most beneficial materialized view. Then

$$s\_cost(v) = \sum_{q_i \in VQ} C_{\min}(q_i)$$

is the best cost of answering all queries in *VQ* individually (i.e. without using any shared operator). Let

$$cost(v) = \begin{cases} t_{total}^{hash}, if\ \forall q \in VQ, q\ is\ executed\ using\ v\ by\ hash\text{-}based\ star\ join \\ t_{total}^{index}, if\ \forall q \in VQ, q\ is\ executed\ using\ v\ by\ index\text{-}based\ star\ join \\ t_{total}^{comb}, if\ \exists q_i, q_j \in VQ\ where\ q_i\ uses\ hash\ and\ q_j\ index\ based\ star\ join \end{cases}$$

be the cost of executing all queries in *VQ* against *v*, by utilizing the shared operators; *savings(v)* equals to the difference between *s_cost(v)* and *cost(v)*.

The complexity of the algorithm is polynomial. To prove this, observe first that $C_{min}(q)$ can be calculated in constant time if we store the relevant information in the lattice during the process of materializing the set *MV*. Then *s_cost(v)* and *cost(v)* are calculated in $O(|VQ|) = O(|Q|)$ time in the worst case. The inner part of the for-loop is executed $O(|AMV|) = O(|MV|)$ times. The while-loop is executed $O(|Q|)$ times because in the worst case, only one query is extracted from *AQ* in each iteration. Therefore, the complexity of *BVF* is $O(|Q|^2 \cdot |MV|)$.

```
ALGORITHM BVF(MV, Q)
/* MV:={v1,v2, …,v|MV|} is the set of materialized views */
/* Q:={q1,q2, …,q|Q|} is the set of queries */

AMV:=MV /* set of unassigned materialized views */
AQ:=Q /* set of unassigned queries */
GlobalPlan:=∅
while AQ≠∅
   best_savings = -∞
   for every vx∈AMV do
      VQ:={q∈AQ: q is answered by vx}
      s_cost:=Single_Query_Cost(VQ) /* the cost to evaluate each query in VQ
                                       individually */
      cost:=Shared_Cost(vx,VQ) /* the cost to evaluate all queries in VQ by vx
                                using shared join operators */
      savings:=s_cost-cost;
      if best_savings < savings then
         best_savings:=savings
         best_view:=vx
      endif
   endfor

   create newSet /* set of queries to be executed by the same shared operator */
   newSet.answered_by_view:=best_view
   newSet.queries:= {q∈AQ: q is answered by best_view}
   GlobalPlan:=GlobalPlan ∪ newSet
   AMV:=AMV-best_view
   AQ:=AQ-{q∈AQ: q is answered by best_view}
endwhile

return GlobalPlan
```

Figure 7: Best View First (BVF) greedy algorithm

Let us now apply *BVF* to the example of figure 3a. If we don't use any shared operators, the most beneficial view to answer q_1 is v_1. Thus, $C_{min}(q_1)$=11100. For the other queries, the most beneficial view is v_2. The cost is $C_{min}(q_j)$=222, $2 \leq j \leq 4$. In the first iteration of the algorithm, the *savings* metric for both views is evaluated. For v_1, $VQ=\{q_1, q_2, q_3, q_4\}$, $s\_cost(v_1) = 11100 + 3 \cdot 222 = 11766$, $cost(v_1) = 10000 + 10000/10 + 4 \cdot 10000/100 = 11400$ and $savings(v_1) = 11766 - 11400 = 366$. For v_2, $VQ=\{q_2, q_3, q_4\}$, $s\_cost(v_2) = 3 \cdot 222 = 666$, $cost(v_2) = 200 + 200/10 + 3 \cdot 200/100 = 226$ and $savings(v_2) = 666 – 226 = 440$. v_2 is selected, and $\{q_2, q_3, q_4\}$ are assigned to it. In the next iteration, q_1 is assigned to v_1. Thus *BVF* produced the optimal execution plan. It is easy to check that *BVF* also delivers the optimal plan for the example of figure 3b.

Theorem 1: *BVF* delivers an execution plan whose cost decreases monotonically when the number of materialized views increases.

Proof: We will prove the theorem by induction. We will only present the case where all queries use hash based star join. The generalization for the other cases is straightforward.

Inductive hypothesis: Let M_i be the set of materialized views and P_i be the plan that *BVF* produces for M_i. Let $|M_i|=i$ and for every $i \leq j \Rightarrow M_i \subseteq M_j$. Then $cost(P_i) \geq cost(P_j)$.

Base Case: Let P_0 be the execution plan when no materialized view is available. Then all queries are answered by *top* (i.e. the most detailed view) and $cost(P_0) = cost(top)$. Assume that v_1 is materialized. *BVF* constructs a new plan P_1. There are two cases: (i) $v_1 \notin P_1$. Then $P_0 \equiv P_1$ and the theorem stands. (ii) $v_1 \in P_1$. Let $|Q|=n$. Then $k < n$ queries are assigned to *top* and the rest (n-k) queries are assigned to v_1. If $k = 0$ (i.e. all queries are assigned to v_2) the proof is trivial, since $size(v_1) \leq size(top)$. So let k >0 and assume that the theorem does not stand. Then

$$cost(P_0) < cost(P_1) \Rightarrow cost(P_0) < cost(v_1) + t_{MV}^{hash}(top) + k \cdot t_Q^{hash}(q_i, top) \tag{1}$$

Since the algorithm inserted v_1 in the plan, this must have happened before *top* view was inserted, because else *top* view would have covered all queries. So

$$\begin{aligned} &savings(v_1) > savings(top) \Rightarrow s\_cost(v_1) - cost(v_1) > s\_cost(top) - cost(P_o) \Rightarrow \\ &\Rightarrow cost(P_0) > s\_cost(top) - s\_cost(v_1) + cost(v_1) \end{aligned}$$

So (1) can be written as:

$$\begin{aligned} &s\_cost(top) - s\_cost(v_1) < t_{MV}^{hash}(top) + k \cdot t_Q^{hash}(q_i, top) \Rightarrow \\ &\Rightarrow n \cdot [t_{MV}^{hash}(top) + t_Q^{hash}(q_i, top)] - (n-k) \cdot [t_{MV}^{hash}(v_1) + t_Q^{hash}(q_j, v_1)] < t_{MV}^{hash}(top) + k \cdot t_Q^{hash}(q_i, top) \Rightarrow \\ &\Rightarrow (n-1) \cdot t_{MV}^{hash}(top) + (n-k) \cdot t_Q^{hash}(q_i, top) < (n-k) \cdot t_{MV}^{hash}(v_1) + (n-k) \cdot t_Q^{hash}(q_j, v_1) \end{aligned} \tag{2}$$

But $t_{MV}^{hash}(top) > t_{MV}^{hash}(v_1) \Rightarrow (n-1) \cdot t_{MV}^{hash}(top) > (n-1) \cdot t_{MV}^{hash}(v_1) \geq (n-k) \cdot t_{MV}^{hash}(v_1)$ (3)

And $t_Q^{hash}(q_i, top) > t_Q^{hash}(q_j, v_1) \Rightarrow (n-k) \cdot t_Q^{hash}(q_i, top) > (n-k) \cdot t_Q^{hash}(q_j, v_1)$ (4)

By adding (3) and (4) we contradict (2), so the theorem stands.

Inductive case: Without lost of generality, we assume that only one new view v_{j+1} is materialized in M_{j+1}. Then *BVF* either does not consider v_{j+1}, in which case $cost(P_{j+1}) = cost(P_j)$, or v_{j+1} is included to

P_{j+1}. Then k queries from the P_j plan will be assigned to the new view. The proof continues in the same way as the base step.

Lemma 1: From theorem 1, it follows that *BVF* delivers an execution plan P whose cost is less or equal to the cost of executing all queries against the most detailed view of the warehouse by using shared star join.

Theorem 1 together with lemma 1, guarantee that *BVF* avoids the pitfalls of the previous algorithms. Note that there is no assurance for the performance of *BVF* compared to the optimal one, since the cost of answering all the queries from the base tables can be arbitrary far from the cost of the optimal plan. Consider again the example of figure 3b, except that there are 100 queries that are answered by $\{v_1, v_3\}$ and 100 queries that are answered by $\{v_2, v_3\}$. *savings* for v_1 and v_2 is zero, while $savings(v_3)$ = 11100 + 16650 – 620 = 27130, so all queries are assigned to v_3. The cost for the plan is 620. However, if we assign to v_1 all the queries that are bellow it and do the same for v_2, the cost of the plan is 525. We can make this example arbitrarily bad, by adding more queries bellow v_1 and v_2.

In general, *BVF* tends to construct a small number of sets, where each set contains many queries that share the same star join. This behavior usually results to high cost plans when there are a lot of queries and a small number of materialized views. To overcome this problem, we developed a multilevel version of *BVF*, called *MBVF*. The idea is that we can recursively explore the plan delivered by *BVF* by assigning some of the queries to views that are lower in the lattice (i.e. less general views) in order to lower the cost. *MBVF* works as follows (see figure 8): First it calls *BVF* to produce an initial plan, called *LowerPlan*. Then, it selects from *LowerPlan* the view v which is higher in the lattice (i.e. the more general view). It assigns to v the queries that cannot be answered by any other view and calls *BVF* again for the remaining views and queries to produce *newPlan*. v and its assigned queries plus the *newPlan* compose the complete plan. If its cost is lower that the original plan, the process continues for *newPlan*, else the algorithm terminates. In the worst case, the algorithm will terminate after examining all the views. Therefore, the complexity is $O(|Q|^2 \cdot |MV|^2)$.

```
ALGORITHM MBVF(MV, Q)
/* MV:={v1,v2, …,v|MV|} is the set of materialized views */
/* Q:={q1,q2, …,q|Q|} is the set of queries */

UpperPlan:=∅
LowerPlan:=BVF(MV,Q)

exit_cond:=false
do
    v:=HigherView(LowerPlan) /* the most general view in LowerPlan */

    VQ:={q∈Q: q is answered by v, AND ¬∃u∈MV,u≠v: q is answered by u}
    if VQ≠∅ then
        if VQ≠Q then
            create newSet /* a set of queries that will share the same view */
            newSet.answered_by_view:=v
            newSet.queries:= VQ
            tempPlan:=UpperPlan ∪ newSet
        else /* all queries in Q can be answered by views other than v */
            tempPlan:=UpperPlan
        endif

        newPlan:=BVF(MV-{v},Q-VQ)

        if Cost(tempPlan ∪ newPlan) < Cost(UpperPlan ∪ LowerPlan) then
            MV:=MV-{v}
            Q:=Q-VQ
            UpperPlan:=tempPlan
            LowerPlan:=newPlan
        else exit_cond:=true /* newPlan didn't reduce the cost */
        endif
    else exit_cond:=true /* only v can answer the queries */
    endif
until exit_cond

return UpperPlan ∪ LowerPlan
```

Figure 8: Multilevel Best View First (*MBVF*) greedy algorithm

The following lemma can be easily derived from the pseudocode of *MBVF*:

Lemma 2: The cost of the execution plan delivered by *MBVF* is in the worst case equal to the cost of the plan produced by *BVF*.

Note that lemma 2 does not imply that the behavior of *MBVF* is monotonic. It is possible that the cost of the plan derived by *MBVF* increases when more materialized views are available, but still it will be less or equal to the cost of *BVF's* plan.

5. Experimental Evaluation

In order to test the behavior of our algorithms under realistic conditions, we constructed three families of synthetic query sets larger than *q2_2*. Each query set contains 100 MDX queries. An MDX query can be analyzed into k sets of $|Q_{SET}|$ related SQL group-by queries. We generated the query sets as follows: For each MDX query we randomly chose k nodes $q_1, q_2, ..., q_k$ in the corresponding lattice. Then, for each q_i, $1 \leq i \leq k$, we randomly selected $|Q_{SET}|$ nodes in the sub-lattice which is rooted in q_i. Table 2 contains details about the query sets. *q50_1* captures the case where an MDX expression contains only related queries, while in *q1_50* the group-by queries are totally random; this is a tricky input for the optimization algorithms.

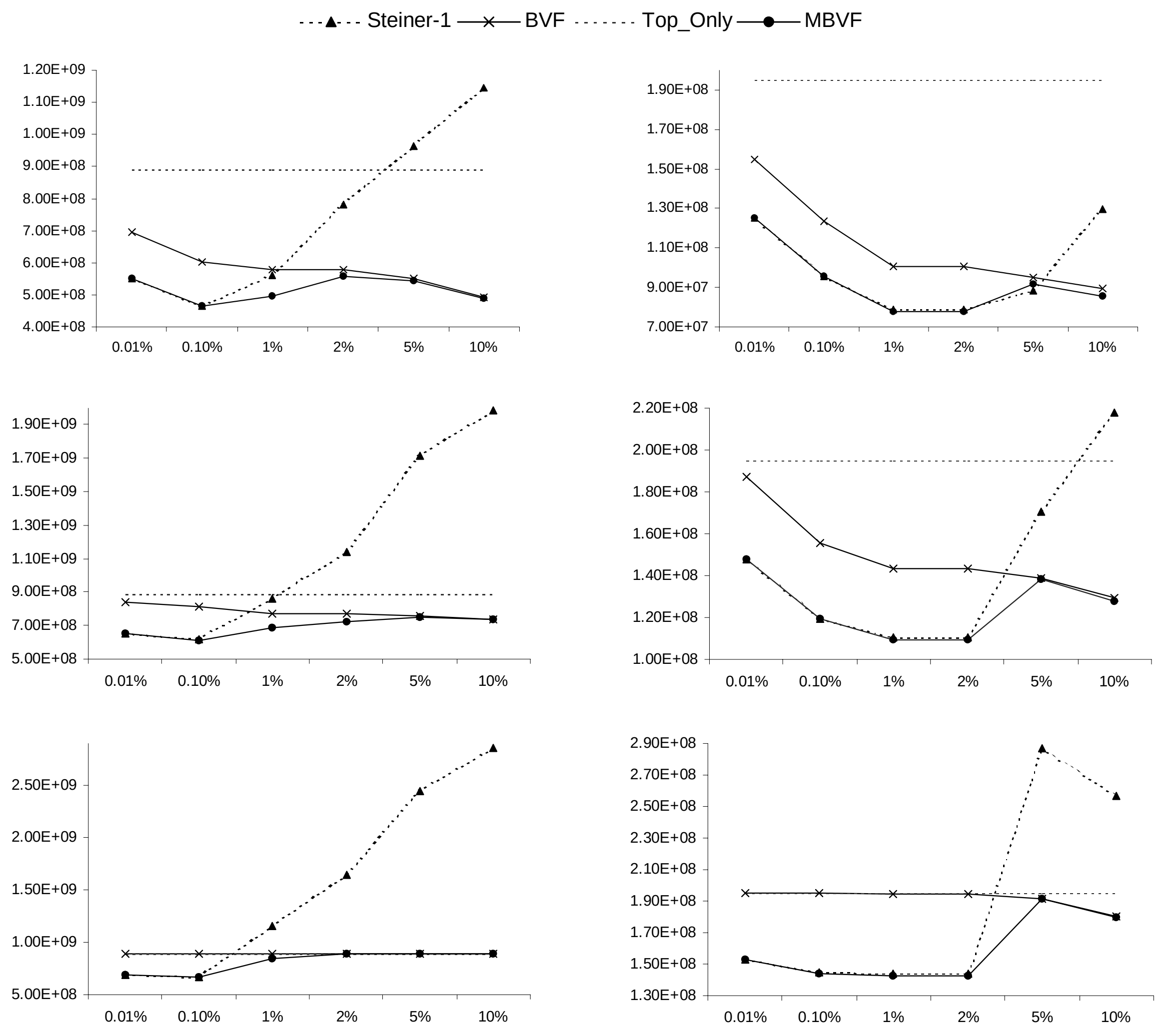
	Number of sets k in each MDX expression	**Number of related group-by queries $\|Q_{SET}\|$ in each set**	**Total number of group-by queries in an MDX expression**	**Total number of MDX expressions**
q2_2	2	2	4	100
q50_1	1	50	50	100
q25_2	2	25	50	100
q1_50	50	1	50	100

Table 2: Details about the query sets

In the first set of experiments, we assume that all queries use hash based star join. Figure 9 presents the cost of the plan versus S_{max}. *GG* and *GG-c* produced similar results and *Steiner-1* outperformed them in most cases, so we only include the later algorithm in our figures. The results from the SYNTH dataset are not presented since they were similar. The first row refers to the *q50_1* query set which is very skewed. Therefore it is easy to identify sets of queries that share their star joins. *BVF* is worse than *Steiner-1* for small values of S_{max} (i.e. small number of materialized views), but when S_{max} increases *Steiner-1* goes into the hard-region and its performance deteriorates. There are cases where the cost of its solution is higher that the *Top_Only* case (i.e. when only the most detailed view is materialized). *BVF* on the other hand, doesn't suffer from the hard-region problem, due to its monotonic property, so it is always better that the *Top_Only* case, and outperforms *Steiner-1* when S_{max} beyond the point that it enters the hard region.

MBVF was found to be better in all cases. For small values of S_{max} the algorithm is almost identical to *Steiner-1*, but when the later goes into the hard-region, *MBVF* follows the trend of *BVF*. Observe that

MBVF is not monotonic. However, since it is bounded by *BVF*, it exits the hard region fast, and even inside the hard region, the cost of the plans does not increase considerably.



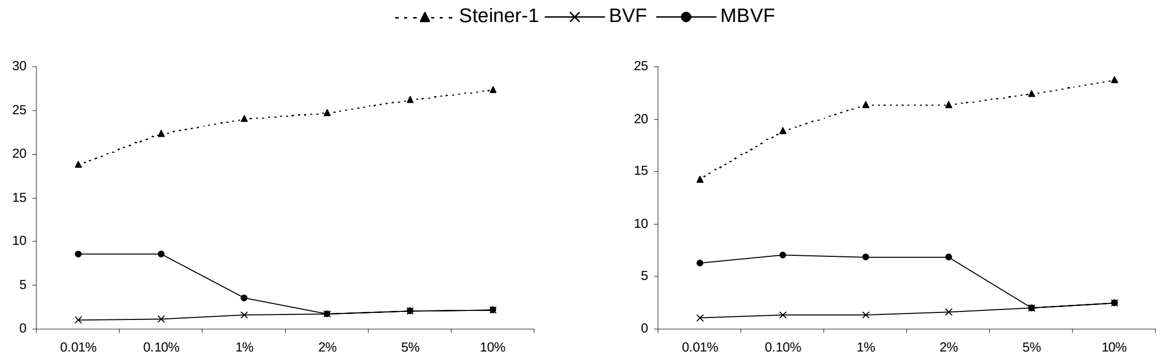
(a) TPC-H dataset **(b)** APB dataset

Figure 9: Total execution cost versus S_{max}. All queries use hash based star join. The first row refers to the *q50_1* query set, the second to the *q25_2* and the third to the *q1_50* query set

In the second and the third row of figure 9, we present the results for the *q25_2* and *q1_50* query sets respectively. Although the trend is the same, observe that the cost of the plans of both *BVF* and *MBVF* approach the cost of the *Top_Only* plan. This is more obvious for the *q1_50* query set. The reason is that the group-by queries inside *q1_50* are random, so there is a small probability that there exist many sets of related queries. Therefore, BVF and MBVF tend to construct plans with one or two sets of queries and assign them to very detailed views.

We mentioned above that *BVF* has in general the trend to deliver plans with only a few shared operators. This is obvious in figure 10, which presents the average number of shared operators per MDX expression as a function of S_{max}. *Steiner-1*, on the other hand, analyses each MDX expression into many related sets. The poor performance of *BVF* for small values of S_{max} is due to the fact that the interrelation among the group-by queries is not exploited enough. By attempting to break the initial plan into smaller ones, *MBVF* constructs plans with more shared operations and outperforms *BVF*.

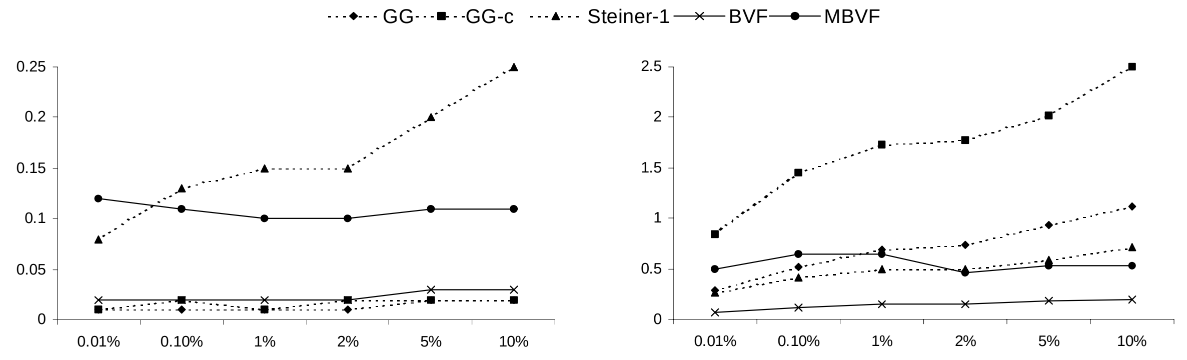
Observe that the number of shared operators for *MBVF* decreases after some point, and the algorithm converges to *BVF*. This is due to the fact that some beneficial general view has been materialized, which can effectively replace two or more of its descendants. The reason the other algorithms enter the hard region is exactly that they fail to recognize such cases.



(a) TPC-H dataset **(b)** APB dataset

Figure 10: Average number of shared operators in each MDX execution plan, versus S_{max}

In figure 11 we present the running time of the algorithms in seconds versus S_{max} for the TPC-H dataset. When the number of queries is small (*q2_2*), the running time for *BVF* is almost the same as for *GG* and *GG-c*, while *MBVF* is one order of magnitude slower. However it is still faster than *Steiner-1*. For a large number of queries, the absolute running time for all algorithms increases. *BVF* is the fastest, while the gap from *MBVF* decreases. *GG* and *Steiner-1* have similar behavior with *MBVF* and *GG-c* is the slowest. The results for the other datasets were similar.



(a) *q2_2* query set (4 group-by queries per MDX) **(b)** *q50_1* query set (50 group-by queries per MDX)

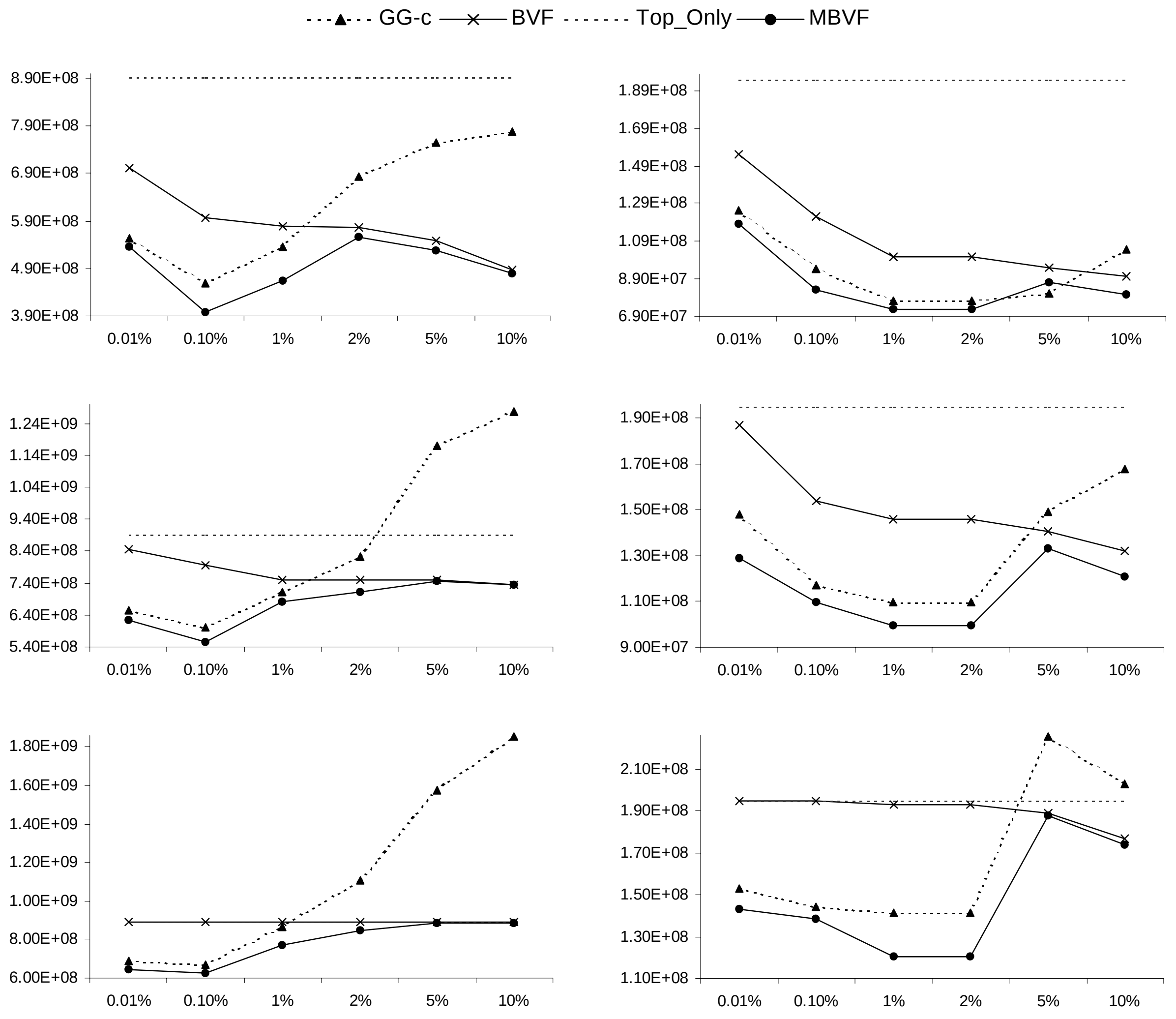
Figure 11: Total running time (in sec) to generate the plan for 100 MDX queries versus S_{max}, for the TPC-H dataset

In our last set of experiments, we tested the general case where some of the queries are processed by hash-based star join, while the rest use index-based hash join. We run experiments where the percentage of the queries that could use index-based star join was set to 50%, 25% and 10%. The subset of queries that could use the indices was randomly selected from our previous query sets. The trend in all the tested cases was the same. In figure 12 we present the cost of the plan versus S_{max} for the 25% case (only *GG-c* is presented in the diagrams, since it delivered the best plans).

The results are similar to the case where only hash-based star joins are allowed. Observe however, that the distance of the produced plans from the *Top_Only* case has increased in most cases. This is due to the fact that the algorithms deliver plans that include shared index-based star joins so they can achieve, in general, lower execution cost.

6. Conclusions

In this paper we conducted an extensive experimental study on the existing algorithms for optimizing multiple dimensional queries simultaneously in multidimensional databases, using realistic datasets. We concluded that the existing algorithms do not scale well if a set of views is materialized to accelerate the OLAP operations. Specifically, we identified the existence of a hard-region in the process of constructing an optimized execution plan, which appears when the number of materialized views increases. Inside the hard region the behavior of the algorithms is unstable, and the delivered plans that use materialized views can be worse than executing all queries from the most detailed view.



(a) TPC-H dataset **(b)** APB dataset

Figure 12: Total execution cost versus S_{max}. 25% of the queries can use index based star join. The first row refers to the *q50_1* query set, the second to the *q25_2* and the third to the *q1_50* query set.

Motivated by this fact, we developed a novel greedy algorithm (*BVF*), which is monotonic and its worst-case performance is bounded by the case where no materialized views are available. Our algorithm outperforms the existing ones beyond the point that they enter the hard-region. However, *BVF* tends to deliver poor plans when the number of materialized views is small. As a solution, we developed a multilevel variation of *BVF*. *MBVF* is bounded by *BVF*, although it does not have the monotonic property. Our experiments indicate that for realistic workloads *MBVF* outperforms its competitors in most cases.

Currently our research focuses on distributed OLAP systems. We are planning to extend our methods for distributed environments, where there may exist multiple replicas of a view. The problem becomes more complicated since we don't only need to decide which view will answer a query, but also the site that will execute the query. Another direction of future work is the efficient cooperation of multi-query optimization techniques with cache control algorithms. The intuition is that we can advise the replacement algorithm to evict cached results based not only on the frequency of the queries but also on the combinations that are posed simultaneously.

References

[1] Baralis E., Paraboschi S., Teniente E., "Materialized view selection in a multidimensional database", Proc. VLDB, 1997.

[2] Codd E.F., Codd S.B., Salley C.T., "Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate", Technical report, 1993.

[3] Deshpande P., Ramasamy K., Shukla A., Naughton J.F., "Caching Multidimensional Queries Using Chunks", Proc. ACM-SIGMOD, 1998.

[4] Gray J., Bosworth A., Layman A., Pirahesh H., "Data cube: a relational aggregation operator generalizing group-by, cross-tabs and subtotals", Proc. ICDE, 1996.

[5] Gupta H., "Selection of Views to Materialize in a Data Warehouse", Proc. ICDT, 1997.

[6] Gupta H., Harinarayan V., Rajaraman A., Ullman J. D., "Index selection for OLAP", Proc. ICDE, 1997.

[7] Gupta H., Mumick I. S., "Selection of Views to Materialize Under a Maintenance-Time Constraint", Proc. ICDT, 1999.

[8] Harinarayan V., Rajaraman A., Ullman J. D., “Implementing data cubes efficiently”, Proc. ACM-SIGMOD, 1996.

[9] Kalnis P., Papadias D., “Optimization Algorithms for Simultaneous Multidimensional Queries in OLAP Environments”, Proc. DaWaK, 2001.

[10] Kimball R., “The Data Warehouse Toolkit”, John Wiley, 1996.

[11] Kotidis Y., Roussopoulos N., “DynaMat: A Dynamic View Management System for Data Warehouses”, Proc. ACM-SIGMOD, 1999.

[12] Liang W., Orlowska M. E., Yu, J. X., “Optimizing multiple dimensional queries simultaneously in multidimensional databases”, The VLDB Journal, 8, 2000.

[13] Microsoft Corp., “OLE DB for OLAP Design Specification”, http://www.microsoft.com

[14] O’Neil P., Quass D., “Improved query performance with variant indexes”, Proc. ACM-SIGMOD, 1997.

[15] OLAP Council, “OLAP Council APB-1 OLAP Benchmark RII”, http://www.olapcouncil.org

[16] Park J., Segev A., “Using common subexpressions to optimize multiple queries”, Proc. ICDE, 1988.

[17] Roy P., Seshadri S., Sudarshan S., Bhobe S., “Efficient and Extensible Algorithms for Multi Query Optimization”, Proc. ACM-SIGMOD, 2000.

[18] Scheuermann P., Shim J., Vingralek R., “WATCHMAN: A Data Warehouse Intelligent Cache Manager”, Proc. VLDB, 1996.

[19] Sellis T. K., “Multi-query optimization”, ACM Trans. On Database Systems, 13(1), 1988.

[20] Shim K., Sellis T. K., “Improvements on heuristic algorithm for multi-query optimization”, Data and Knowledge Engineering, 12(2), 1994.

[21] Shukla A., Deshpande P. M., Naughton J. F., Ramasamy K., "Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies", Proc. VLDB, 1996.

[22] Shukla A., Deshpande P., Naughton J. F., “Materialized View Selection for Multidimensional Datasets”, Proc. VLDB, 1998.

[23] Sundaresan P.: “Data warehousing features in Informix Online XPS”, Proc. 4th PDIS, 1996.

[24] Transaction Processing Performance Council, “TPC-H Benchmark Specification”, v. 1.2.1, http://www.tpc.org

[25] Zelikovsky A., “A series of approximation algorithms for the acyclic directed Steiner tree problem”, Algorithmica 18, 1997.

[26] Zhao Y., Deshpande P. M., Naughton J. F., Shukla A., “Simultaneous optimization and evaluation of multiple dimension queries”, Proc. ACM-SIGMOD, 1998.