Optimization algorithms for simultaneous multidimensional queries in OLAP environments

Panos Kalnis and Dimitris Papadias

Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
http://www.cs.ust.hk/{~kalnis, ~dimitris}

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 benchmarks, concluding that they do not scale well for realistic workloads. Motivated by this fact, we developed 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.

1 Introduction

Data warehouses have been successfully employed for assisting decision-making by offering a global view of the enterprise data and providing mechanisms for On-Line Analytical Processing (OLAP) [CCS93]. A common technique to accelerate OLAP operations is to store some redundant data, either statically [Gupt97, GM99, SDN98] or dynamically [KR99].

Most of OLAP literature assumes that queries are sent to the system one at a time. In multi-user environments, however, many queries can be submitted concurrently. In addition, the new API proposed by Microsoft [MS] for Multi-Dimensional Expressions (MDX), which becomes 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 [PS88, S88, SS94, RSSB00], but due to the restricted nature of the problem, better techniques can be developed.

Zhao et al. [ZDNS98] were the first ones 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, or 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 relational database systems. In the same paper they propose three greedy algorithms for creating the optimized execution plan for an MDX query, using the new join operators. Liang et. al [LOY00] also present approximation algorithms for the same problem.

In this paper we use the TPC-H [TPC] and APB [APB] benchmarks in addition to a 10-dimensional synthetic database, to test the performance of the above-mentioned algorithms under a realistic workload. Our experimental results suggest that the existing algorithms do not scale well when we relax the constraints for the view selection problem. We observed that in many cases when we allowed more space for materialized views, the execution cost of the plan derived by the optimization algorithms was higher than the case where no materialization was 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 trying to identify 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 [ZDNS98] and [LOY00]. 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

A multidimensional expression (MDX) [MS] provides a common interface for decision support applications to communicate with OLAP servers. Here we are interested on the feature of expressing several related OLAP queries with a single MDX expression. Therefore, an MDX expression can be decomposed into a set Q of group-by SQL queries. The intuition behind optimizing an MDX expression is to construct subsets of Q that share star joins, assuming a star schema for the warehouse. Usually, when the selectivity of the queries is low, hash-based star joins [Su96] are used; otherwise, the index-based star join method [OQ97] can be applied. [ZDNS98] 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:

$$t_{MV}^{hash}(v) = Size(v) \cdot t_{I/O} + t_{hash\_join}(v) \quad \textbf{(1)}$$

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:

$$t_{Q}^{hash}(q, mv(q)) = Size(mv(q)) \cdot t_{CPU}(q, mv(q)) \quad \textbf{(2)}$$

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:

$$t_{total}^{hash} = \sum_{v \in MV'} t_{MV}^{hash}(v) + \sum_{q \in Q, mv(q) \in MV'} t_{Q}^{hash}(q, mv(q)) \quad \textbf{(3)}$$

The problem of finding the optimal execution plan is equivalent to minimizing t_{total}^{hash} which is likely to be NP-hard. [LOY00] 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 Zelikovsky [Zeli97]. 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*. Similar to the previous case, a set of queries are answered by the same view v, but there are bitmap indexes that can be used to accelerate all queries. The read and execution cost are defined as before, the only difference being that they are scaled according to the selectivity of the indexes (see [KP00] for details). The aim is to minimize the total cost t_{total}^{index}. In addition to an exact exponential method, [LOY00] propose a polynomial approximation 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 star joins*. As the name implies, this is a combination of the previous two cases. [ZDNS98] propose three heuristic algorithms to construct an execution plan: i) the *Two Phase Local Optimal* algorithm (*TPLO*) which constructs the optimal plan for each query and then merges the individual plans, ii) the *Extended Two Phase Local Greedy* algorithm (*ETPLG*) which constructs the execution plan incrementally, adding one query at a time and starting from the most general queries, and iii) the *Global Greedy* algorithm (*GG*) which is similar to *ETPLG* but allows modifications to the already constructed part of the plan. [LOY00] propose another algorithm, named *GG-c*, which is similar to *ETPLG* but the order that the queries are inserted is defined dynamically.

None of the above algorithms scales well, when the number of materialized views increases. In the next section we present an example that highlights the scalability problem and verify experimentally that it affects severely the performance of the algorithms under realistic workloads.

3 Motivation

Figure 1 shows an instance of the multiple query optimization problem where {v_1, v_2, v_3} is the set of materialized views and {q_1, q_2} are the queries. 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-c* will search for the combination of queries and views that result in 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. Assume that v_3 is the fact table of the warehouse and v_1, v_2 are 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, the performance deteriorates instead of improving. Similar examples can be also constructed for the other algorithms.

In order to evaluate this situation under realistic conditions, we employed datasets from the TPC-H benchmark [TPC], the APB benchmark [APB] and a 10-dimensional synthetic database (SYNTH), where the size of the fact table was 6M, 1.3M and 20M tuples, respectively (see [KP00] for details). 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). 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. We varied the available space for the materialized views (S_{max}) from 0.01% to 10% of the size of the entire data cube (i.e. the case where all nodes in the lattice are materialized). We used the *GreedySelect* [HRU96] algorithm to select the set of materialized views.

We employed the shared operators and we compared the plans delivered by the optimization algorithms, against the optimal plan. All the queries used hash-based star join. We implemented the greedy algorithm of [ZDNS98] (*GG*) and the one of [LOY00] (*GG-c*). We also implemented the Steiner-tree-based approximation algorithm of [LOY00] for hash-based queries (*Steiner-1*). We set ε = 1, since for smaller values of ε the complexity of the algorithm increases while its performance doesn't change considerably, as the experiments of Liang et. al. 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.

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

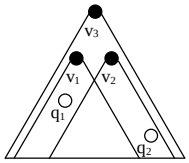
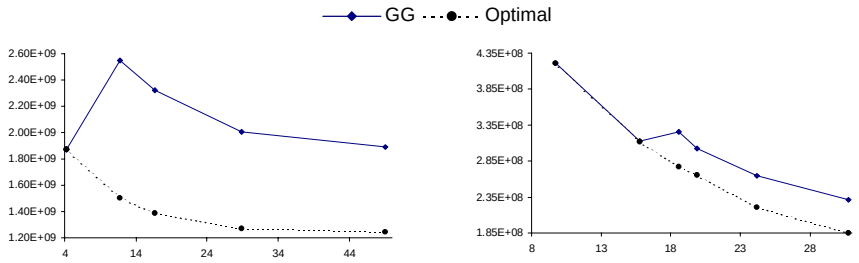


Fig. 1. Two instances of the multiple-query optimization problem.v_1|=100, |v_2|=150, |v_3|=200



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

Fig. 2. Total execution cost versus *AVQ*

(i.e. we could achieve lower cost if we had executed the queries against the base tables).

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: (i) 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. (ii) 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. (iii) 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 2 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 TPC-H, observe that for small values of *AVQ*, the solution of *GG* is identical to the optimal one. Therefore the high execution cost for the plan is due to the specific instance of the problem. 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 APB and for other query setsm, for all algortithms.

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 3 shows the pseudocode of *BVF*.

```
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≠∅
    Let best_view be the view with the highest savings value
    /* newSet is a set of queries that share an 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
```

Fig. 3. Best View First (BVF) greedy algorithm

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

$$s\_\cos t(v) = \sum_{q_i \in VQ} C_{\min}(q_i) \quad (4)$$

is the best cost of answering all queries in VQ individually (i.e. without using any shared operator). Let $cost(v)$ 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 $O(|Q|^2 \cdot |MV|)$.

Theorem 1: *BVF* delivers an execution plan whose cost decreases monotonically when the number of materialized views increases. The proof can be found in the full version of the paper [KP00]. It follows that:

Lemma 1: *BVF* delivers an execution plan P whose cost is less or equal to the cost of executing all queries against the base tables 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 1, 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 [KP00]): 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 $O(|Q|^2 \cdot |MV|^2)$.

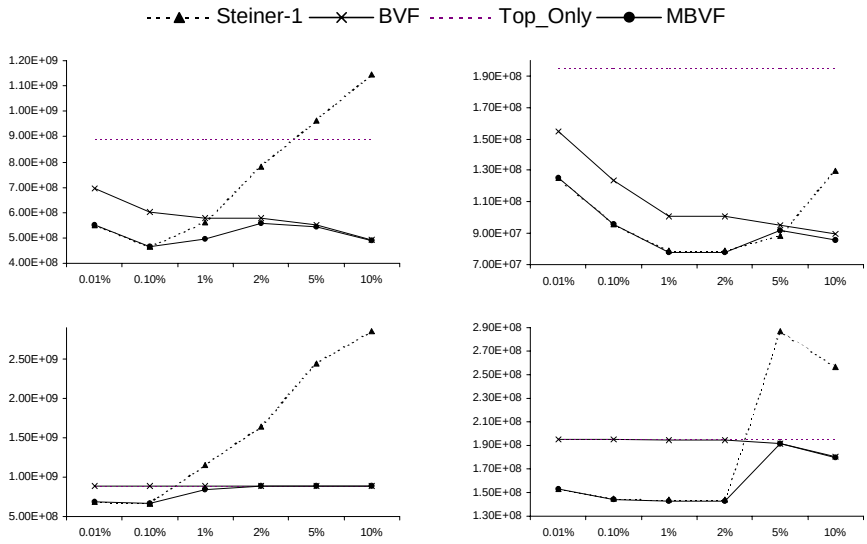
Lemma 2: The cost of the execution plan delivered by *MBVF* is in the worst case equal to the cost of the plan produced by *BVF*.

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 *S* sets of $|Q_{SET}|$ related SQL group-by queries. We generated the query sets as follows: For each MDX query we have randomly chosen *S* nodes q_1, q_2, ..., q_S in the corresponding lattice. Then, for each q_i, $1 \le i \le S$, we randomly selected $|Q_{SET}|$ nodes in the sub-lattice which is rooted in q_i. We denote each query set as $q|Q_{SET}|\_S$. For instance, the set *q50_1* captures the case where an MDX expression contains only related queries, while in *q1_50* the group-by queries are totally random.

In the first set of experiments, we assume that all queries use hash based star join. Figure 4 presents the cost of the plan versus S_{max} (the cost was calculated using the cost model from section 2). *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 SYNTH dataset is not shown due to lack of space; however the results 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 there are no materialized views). *BVF* on the other hand, doesn't



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

Fig. 4. Total execution cost versus S_{max}. All queries use hash based star join. The first row refers to the *q50_1* query set and the second to the *q1_50* query set

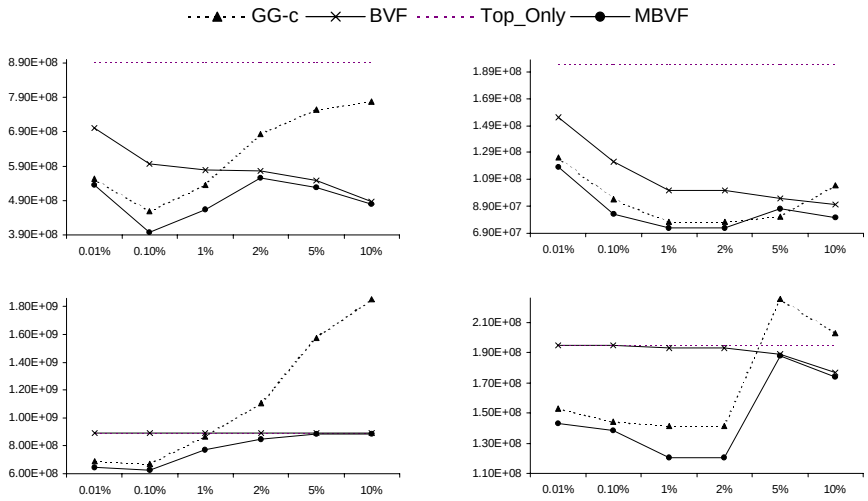
suffer from the hard-region problem, due to its monotonic property, so it is always better that the *Top_Only* case, and outperforms *Steiner-1* when S_{max} increases

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 dramatically.

In the second row of figure 4, we present the results for the *q1_50* query set. 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. 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 and assign it to a general view, or even to the base tables. Similar results were obtained for the *q25_2* query set.

In the next experiment, we tested the general case where some of the group-by queries of each MDX are processed by hash-based star join, while the rest use index-based hash join. We run experiments where the percentage of the group-by 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. We did not consider the quite unrealistic case where all the queries would benefit from the indices. Our results suggest that the trend in all the tested cases was the same. In figure 5 we present the results for the 25% case (only *GG-c* is presented in the diagrams, since it delivered the best plans).

The results are similar with the case where only hash-based star joins are allowed. Observe however, that the distance of the produced plans from the *Top_Only* plan has



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

Fig. 5. 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 and the second to the *q1_50* query set.

increased in most cases. This is because 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 base tables.

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.

Acknowledgements

This work was supported by grants HKUST 6070/00E and HKUST 6090/99E from Hong Kong RGC

References

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

[CCS93] Codd E.F., Codd S.B., Salley C.T., "Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate", Technical report, 1993.

[GM99] Gupta H., Mumick I. S., "Selection of Views to Materialize Under a Maintenance-Time Constraint", Proc. ICDT, 1999.

[Gupt97] Gupta H., "Selection of Views to Materialize in a Data Warehouse", Proc. ICDT, 1997.

[HRU96] Harinarayan V., Rajaraman A., Ullman J. D., "Implementing data cubes efficiently", Proc. ACM-SIGMOD, 1996.

[KP00] Kalnis P., Papadias D., "Optimization algorithms for simultaneous multidimensional queries in OLAP environments", Technical report, HKUST-CS00-04, 2000, available at http://www.cs.ust.hk/~kalnis/hkust-cs00-04.pdf.

[KR99] Kotidis Y., Roussopoulos N., "DynaMat: A Dynamic View Management System for Data Warehouses", Proc. ACM-SIGMOD, 1999.

[LOY00] Liang W., Orlowska M. E., Yu, J. X., "Optimizing multiple dimensional queries simultaneously in multidimensional databases", The VLDB Journal, 8, 2000.

[MS] Microsoft Corp., "OLE DB for OLAP Design Specification", http://www.microsoft.com

[OQ97] O'Neil P., Quass D., "Improved query performance with variant indexes", Proc. ACM-SIGMOD, 1997.

[PS88] Park J., Segev A., "Using common subexpressions to optimize multiple queries", Proc. ICDE, 1988.

[RSSB00] Roy P., Seshadri S., Sudarshan S., Bhobe S., "Efficient and Extensible Algorithms for Multi Query Optimization", Proc. ACM-SIGMOD, 2000.

[S88] Sellis T. K., "Multi-query optimization", ACM Trans. On Database Systems, 13(1), 1988.

[SDN98] Shukla A., Deshpande P., Naughton J. F., "Materialized View Selection for Multidimensional Datasets", Proc. VLDB, 1998.

[SS94] Shim K., Sellis T. K., "Improvements on heuristic algorithm for multi-query optimization", Data and Knowledge Engineering, 12(2), 1994.

[Su96] Sundaresan P.: "Data warehousing features in Informix Online XPS", Proc. 4th PDIS, 1996.

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

[Zeli97] Zelikovsky A., "A series of approximation algorithms for the acyclic directed Steiner tree problem", Algorithmica 18, 1997.

[ZDNS98] Zhao Y., Deshpande P. M., Naughton J. F., Shukla A., "Simultaneous optimization and evaluation of multiple dimension queries", Proc. ACM-SIGMOD, 1998.