Preprocessing for Fast Refreshing Materialized Views in DB2

Wugang Xu[1], Calisto Zuzarte[2], Dimitri Theodoratos[1], and Wenbin Ma[2]

[1] New Jersey Institute of Technology
wx2@njit.edu, dth@cs.njit.edu
[2] IBM Canada Ltd.
calisto, wenbinm@ca.ibm.com

Abstract. Materialized views (MVs) are used in databases and data warehouses to greatly improve query performance. In this context, a great challenge is to exploit commonalities among the views and to employ multi-query optimization techniques in order to derive an efficient global evaluation plan for refreshing the MVs concurrently. $IBM\ DB2^{®}\ Universal\ Database^{TM}$ (DB2 UDB) provides two query matching techniques, query stacking and query sharing, to exploit commonalities among the MVs, and to construct an efficient global evaluation plan. When the number of MVs is large, memory and time restrictions prevent us from using both query matching techniques in constructing efficient global plans. We suggest an approach that applies the query stacking and query sharing techniques in different steps. The query stacking technique is applied first, and the outcome is exploited to define groups of MVs. The number of MVs in each group is restricted. This allows the query sharing technique to be applied only within groups in a second step. Finally, the query stacking technique is used again to determine an efficient global evaluation plan. An experimental evaluation shows that the execution time of the plan generated by our approach is very close to that of the plan generated using both query matching techniques without restriction. This result is valid no matter how big the database is.

1 Introduction

The advent of data warehouses and of large databases for decision support has triggered interesting research in the database community. With decision support data warehouses getting larger and decision support queries getting more complex, the traditional query optimization techniques which compute answers from the base tables can not meet the stringent response time requirements. The most frequent solution used for this problem is to store a number of materialized views (MVs). Query answers are computed using these materialized views instead of using the base tables exclusively. Materialized views are manually or automatically selected based on the underlying schema and database statistics so that the frequent and long running queries can benefit from them. These queries are rewritten using the materialized views prior to their execution. Experience with the TPC-D benchmark and several customer applications has shown that MVs can often improve the response time of decision support queries by orders of magnitude [9]. This performance advantage is so big that TPC-D [1] had ceased to be an effective performance discriminator after the introduction of the systematic use of MVs [9].

Although this technique brings a great performance improvement, it also brings some new problems. The first one is the selection of a set of views to materialize in order to minimize the execution time of the frequent queries while satisfying a number of constraints. This is a typical data warehouse design problem. In fact, different versions of this problem have been addressed up to now. One can consider different optimization goals (e.g. minimizing the combination of the query evaluation and view maintenance cost) and different constraints (e.g. MV maintenance cost restrictions, MV space restrictions etc.). A general framework for addressing those problems is suggested in [7]. Nevertheless, polynomial time solutions are not expected for this type of problem. A heuristic algorithm to select both MVs and indexes in a unified way has been suggested in [10]. This algorithm has been implemented in the IBM DB2 design advisor [10].

The second problem is how to rewrite a query using a set of views. A good review of this issue is provided in [3]. Deciding whether a query can be answered using MVs is an NP-hard problem even for simple classes of queries. However, exact algorithms for special cases and heuristic approaches allow us to cope with this problem. A novel algorithm that rewrites a user query using one or more of the available MVs is presented in [9]. This algorithm exploits the graph representation for queries and views (*Query Graph Model - QGM*) used internally in DB2. It can deal with complex queries and views (e.g. queries involving grouping and aggregation and nesting) and has been implemented in the IBM DB2 design advisor.

The third problem is related to the maintenance of the MVs [5]. The MVs often have to be refreshed immediately after a bulk update of the underlying base tables, or periodically by the administrator, to synchronize the data. Depending on the requirements of the applications, it may not be necessary to have the data absolutely synchronized. The MVs can be refreshed incrementally or recomputed from scratch. In this paper we focus on the latter approach for simplicity. When one or more base tables are modified, several MVs may be affected. The technique of multi-query optimization [6] can be used to detect common subexpressions [8] among the definitions of the MVs and to rewrite the views using these common subexpressions. Using this technique one can avoid computing complex expressions more than once.

An algorithm for refreshing multiple MVs in IBM DB2 is suggested in [4]. This algorithm exploits a graph representation for multiple queries (called *global QGM*) constructed using two query matching techniques: query stacking and query sharing. Query stacking detects subsumption relationships between query or view definitions, while query sharing artificially creates common subexpressions which can be exploited by two or more queries or MVs. Oracle 10g also provides an algorithm for refreshing a set of MVs based on the dependencies among MVs [2]. This algorithm considers refreshing one MV using another one which has already been refreshed. This method is similar to the query stacking technique used in DB2. However, it does not consider using common subsumers for optimizing the refresh process (a technique that corresponds to query sharing used in DB2). This means they may miss the optimal evaluation plan.

When there are only few MVs to be refreshed, we can apply the method proposed in [4] to refresh all MVs together. This method considers both query stacking and query sharing techniques, and a globally optimized refresh plan is generated. However when the number of MVs gets larger, a number of problems prevent us from applying this method.

The first problem relates to the generation of a global plan. When there are many MVs to be refreshed, too much memory is required for constructing a global QGM using both query sharing and query stacking techniques. Further, it may take a lot of time to find an optimal global plan from the global QGM. The second problem relates to the execution of the refresh plan. There are several system issues here. The process of refreshing MVs usually takes a long time, since during this period, MVs are locked. User queries which use some MVs either have to wait for all MVs to be refreshed, or routed to the base tables. Either solution will increase the execution time. Another system issue relates to the limited size of the statement heap which is used to compile a given database statement. When a statement is too complex and involves a very large number of referenced base tables or MVs considered for matching, there may not be enough memory to compile and optimize the statement. One more system issue relates to transaction control. When many MVs are refreshed at the same time (with a single statement), usually a large transaction log is required. This is not always feasible. Further if something goes wrong during the refreshing, the whole process has to start over.

To deal with the problems above, we propose the following approach. When too many MVs need to be refreshed and the construction of a global QGM based on query stacking and query sharing together is not feasible, we partition the MV set into smaller groups based on query stacking alone. Then, we apply query sharing to each group independently. Consequently, we separate the execution plan into smaller ones, each involving fewer MVs. Intuitively, by partitioning MVs into smaller groups, we apply query sharing only within groups and query stacking between groups such that MVs from the lower groups are potentially exploited by the groups above. An implementation and experimental evaluation of our approach shows that our method has comparable performance to the approach that uses a globally optimized evaluation plan while at the same time avoiding the aforementioned problems.

In the next section, we present the QGM model and the two query matching techniques. Section 3 introduces our MV partition strategy. Section 4 presents our experimental setting and results. We conclude and suggest future work in Section 5.

2 Query Graph Model and Query Matching

In this section, we introduce the concept of QGM model which is used in DB2 to graphically represent queries. We first introduce the QGM model for a single query. Then we extend it to a global QGM model for multiple queries. This extension requires the concept of query matching using both query stacking and query sharing techniques.

2.1 Query Graph Model

The QGM model is the internal graph representation for queries in the DB2 database management system. It is used in all steps of query optimization in DB2: parsing and semantic checking, query rewriting transformation and plan optimization. Here, we show with an example how queries are represented in the QGM model. A query in the QGM model is represented by a set of boxes (called *Query Table Boxes – QTBs*) and arcs between them. A QTB represents a view or a base table. Typical QTBs are *select* QTBs and *group-by* QTBs. Other kinds of QTBs include the *union* and the *outer-join* QTBs.

Below, we give a query with *select* and *group-by* operations in SQL. Figure 1 shows a simplified QGM representation for this query.

```
select c.c3, d.d3, sum(f.f3) as sum
from c, d, fact f
where c.c1 =f.f1 and d.d1 = f.f2 and
      c.c2 = 'Mon' and d.d2 > 10
group by c.c3,d.d3
having sum > 100
```

2.2 Query Matching

To refresh multiple MVs concurrently, a global QGM for all of the MVs is generated using the definitions tied together loosely at the top. All QTBs are grouped into different levels with the base tables belonging to the bottom level. Then, from bottom to top, each QTB is compared with another QTB to examine whether one can be rewritten using the other. If this is the case, we say that the latter QTB *subsumes* the former QTB. The latter QTB is called the *subsumer* QTB while the former is called the *subsumee* QTB. A rewriting of the subsumee QTB using the subsumer QTB may also be generated at the time of matching, and this aditional work is called *compensation*. The comparison continues with the parent QTBs of both the sumsumer and subsumee QTBs. This process continues until no more matches can be made.

If the top QTB of one MV subsumes some QTB of another MV, then the former MV subsumes the latter MV. This kind of matching is called *query stacking* because it ultimately determines that one MV can be rewritten using the other and the subsumee MV can be “stacked” on the subsumer MV.

In some cases, it is possible that we may not find a strict subsumption relationship between two MVs even if they are quite close to having one. For instance, a difference in the projected attributes of two otherwise equivalent MVs will make the matching fail. The matching technique of DB2 is extended in [4] to deal with this case. In some cases when there is no subsumption relationship between two MVs, an artificially built common subexpression (called *common subsumer*) can be constructed such that both

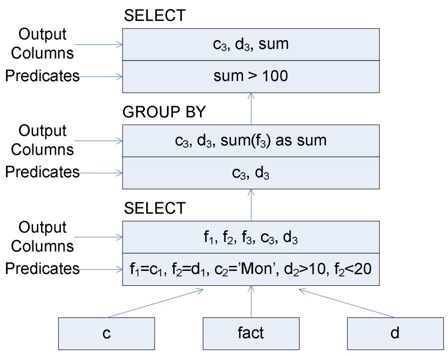
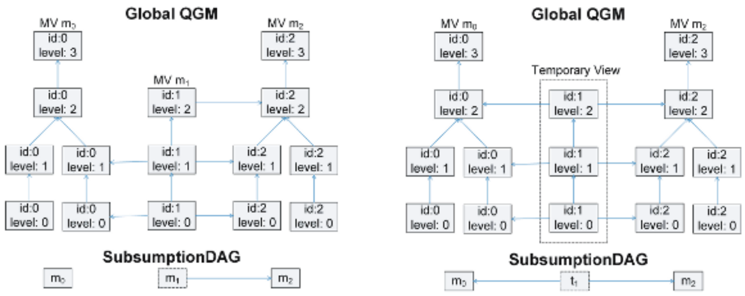


Fig. 1. QGM graph for query Q_1

MVs can be rewritten using this common subsumer. Because this common subsumer is “shared” by both MVs, this matching technique is called *query sharing*. With query sharing, matching techniques can be applied to a wider class of MVs.

In Figure 2, we show examples of query matching techniques. In Figure 2(a), we show the matching using query stacking only. In this example, we have three queries m_0, m_1, m_2. For each query pair, we match their QTBs from bottom to top until the top QTB of the subsumer query is reached. Since there is a successful matching of the top QTB of query m_1 with some QTB of query m_2, there is a subsumption relationship from m_1 to m_2 (m_1 subsumes m_2). This is not the case with queries m_0 and m_1. The matching process defines a query *subsumption DAG* shown in Figure 2(a). In this DAG, each node is a query. Since query m_1 subsumes query m_2, we draw a directed line from m_1 to m_2. In Figure 2(a), we show the subsumption DAG for queries m_0, m_1 and m_2. There is one subsumption edge from m_1 to m_2 while query m_0 is a disconnected component. This subsumption DAG can be used to optimize the concurrent execution of the three queries. For example, we can compute the results of m_0 and m_1 from base tables. Then, we can compute query m_2 using m_1 based on the rewriting of m_2 using m_1, instead of computing it using exclusively base tables. Query m_2 is “stacked” on m_1 since it has to be computed after m_1.

In this example, we also observe that although m_2 can be rewritten using m_1, we cannot rewrite m_0 using m_2 or vise versa. We cannot even find a successful match of the bottom QTBs of m_0 and m_2 based on query stacking. This is quite common in practice. When we try to answer m_0 and m_2 together, and we cannot find a subsumption relationship between them, we can try to create a new query, say t_1, which can be used to answer both queries m_0 and m_2. This newly constructed query is called *common subsumer* of the two queries m_0 and m_2 because it is constructed in a way so that both queries m_0 and m_2 can be rewritten using it. Although the common subsumer is not a user query to be answered, we can find its answer and then use it to compute the answers of both queries m_0 and m_2. As a “common part” between m_0 and m_2, t_1 is computed only once, and therefore, its computation might bring some benefit in the concurrent execution of m_0 and m_2. In the example of Figure 2(b), there is no subsumption edge between m_0 and m_2. However, after adding a common subsumer t_1 of m_0 and m_2, we have two subsumption edges: one from t_1 to m_0 and one from t_1 to m_2.



(a) Query stacking (b) Query sharing

Fig. 2. Query matching

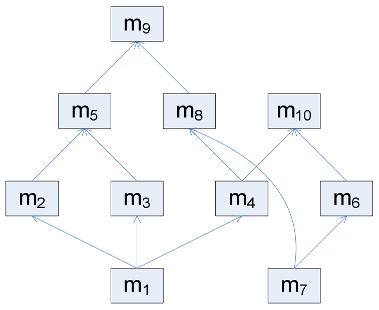
The subsumption relationship graph is a DAG because there is no cycle in it. In most cases, if one MV subsumes another one, the latter one cannot subsume the former one. Nevertheless in some cases, two or more MVs may subsume each other, thus generating a subsumption cycle. The DB2 matching techniques will ignore one subsumption relationship randomly, when this happens, to break any cycles. This will guarantee the result subsumption graph to be a real DAG. In drawing a subsumption DAG, if $m_1 \rightarrow m_2$, and $m_2 \rightarrow m_3$, we don't show in the DAG the transitive subsumption edge $m_1 \rightarrow m_3$. However, this subsumption relationship can be directly derived from the DAG, and it is of the same importance as the other subsumption relationships in optimizing the computation of the queries.

3 Group Partition Strategy

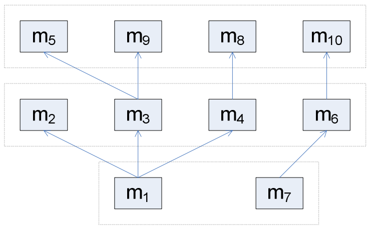
If there are too many MVs to be refreshed then, as we described above, we may not be able to construct the global QGM using both query stacking and query sharing techniques. Our goal is to partition the given MV set into groups that are small enough so that both query matching techniques can be applied, and we do not face the problems mentioned in the introduction. Our approach first creates a subsumption DAG using query stacking only which is a much less memory and time consuming process. This subsumption DAG is used for generating an optimal global evaluation plan. The different levels of this plan determine groups of materialized views on which query sharing is applied.

3.1 Building an Optimal Plan Using Query Stacking

Given a set of MVs to be refreshed, we construct a global QGM using only query stacking and then create a subsumption DAG as described in Section 2.2. Then, we have the query optimizer choose an optimal plan for computing each MV using either exclusively base relations or using other MVs in addition to base tables as appropriate. The compensations stored in the global QGM of a MV using other MVs are used to support this task. The optimizer decides whether using a MV to compute another MV



(a) Subsumption DAG



(b) Optimal global logical plan

Fig. 3. Query stacking based refreshing

is beneficial when compared to computing it from the base relations. These optimal "local" plans define an optimal global plan for refreshing all the queries. Figure 3(a) shows an example of a subsumption DAG for ten MVs. Transitive edges are ignored for clarity of presentation. Figure 3(b) shows an optimal global evaluation plan.

Groups are defined by the views in the optimal plan. If one group is still too big for the query sharing technique to be applied, we can divide it into suitable subgroups heuristically based on some common objects and operations within the group or possibly randomly.

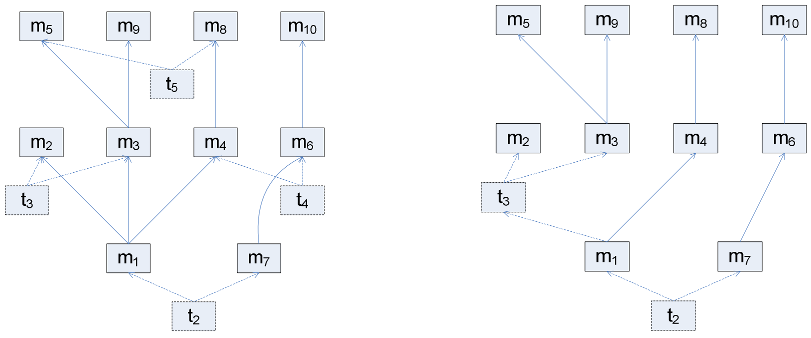
3.2 Adding also Query Sharing

By considering the query stacking technique only, we may miss some commonalities between queries which can be beneficial to the refreshing process. Therefore, we enable both query matching techniques within each group to capture most of those commonalities. We outline this process below.

1. We apply query stacking and query sharing techniques to the MVs of each group. Even though no subsumption edges will be added between MVs in the group, some common subsumers may be identified and new subsumption edges will be added from those common subsumers to MVs in the group.
2. We apply the query stacking technique to the common subsumers of one group and the MVs of lower groups. Lower groups are those that comprise MVs from lower levels of the optimal global plan. This step might add some new subsumption edges from MVs to common subsumers in the subsumption DAG.
3. Using a common subsumer induces additional view materialization cost. However, if this cost is lower than the gain we obtained in computing the MVs that use this common subsumer, it is beneficial to materialize this common subsumer. We call such a common subsumer candidate common subsumer. The use of a candidate common subsumer may prevent the use of other candidate common subsumers. We heuristically retain those candidate common subsumers such that no one of them prevents the use of the others and together yield the highest benefit. This process is applied from the bottom level to the top level in the subsumption DAG.
4. We have the optimizer create a new optimal global plan using the retained candidate common subsumers. Compared to the optimal global plan constructed using only query stacking, this optimal global plan contains also some new MVs, in the form of the retained candidate common subsumers.

During the refreshing of the MVs, a common subsumer is first materialized when it is used for refreshing another MV and it is discarded when the last MV that uses it has been refreshed.

In Figure 4, we show the construction of an optimal global plan taking also query sharing into account. Figure 4(a) shows the subsumption DAG of Figure 4(b) along with some common subsumers. Dotted directed edges indicate subsumption edges involving common subsumers. Among the candidate common subsumers, some of them are retained in the optimal global plan. Such an optimal global plan is shown in Figure 4(b). This optimal global plan will have a better performance than the one of Figure 3(b).



(a) Subsumption DAG (b) Optimal global logical plan

Fig. 4. Query sharing based refreshing

4 Performance Test

For the experimental evaluation we consider a database with a star schema. We also consider a number of MVs to be refreshed (16 in our test). We keep the number of MVs small enough so that, in finding an optimal global evaluation plan, both the query stacking and query sharing techniques can be applied without restrictions. The performance comparison test is not feasible when we have too many MVs. The goal is to compare the performance of different approaches. We compare the performance of four kinds of MV refreshing approaches for different sizes of databases. These approaches are as follows:

1. *Naive Refreshing(NR):* Refresh each MV one by one by computing its new state using the base tables referrenced in the MV definition exclusively. This approach disallows any multi-query optimization technique or other already refreshed MV exploitation.
2. *Stacking-Based Refreshing(STR):* Refresh each MV one by one in the topological order induced by the optimal global plan constructed using the query stacking technique only. (for example, the optimal global plan of Figure 3(b)) in our example. This approach disallows query sharing. With this approach some MVs are refreshed using the base relations exclusively. Some other MVs are refreshed using other MVs if they have a rewriting using those MVs that are in lower groups in the optimal global plan.
3. *Group-Sharing-Based Refreshing(SHR):* Refresh each MV in the topological order induced by the optimal global evaluation plan constructed using query stacking first and then query sharing only within groups (for example, the optimal global plan of Figure 4(b)).
4. *Unrestricted-Sharing-Based Refreshing(USR):* Refresh all MVs based on an optimal global plan constructed using, without restrictions, both query matching techniques.

Our test schema consists of one fact table and three dimension tables. Each dimension table has 10,000 tuples, while the number of tuples of the fact table varies from 100,000 to 10,000,000. We refresh the set of MVs with each one of the four refreshing approaches mentioned above, and we measure the overall refreshing time. We run our performance test on a machine with the following configuration.

$Model$	OS	$Memory$	$CPUs$	$rPerf$	$Database$
P640-B80	AIX 5.2 ML06	8 GB	4	3.59	DB2 V91

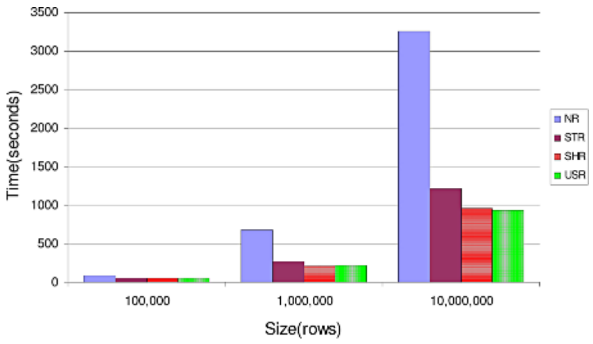


Fig. 5. Performance test result for different refreshing method

Figure 5 shows our experimental results. The unrestricted-sharing-based approach always has the best performance since it allows unrestricted application of both query stacking and query sharing techniques. The group-sharing-based approach has the second best performance because, even though it exploits both query matching techniques, they are considered separately in different steps and query sharing is restricted only within groups. The stacking-based approach is the next in performance because it cannot take advantage of the query sharing technique. Finally, far behind in performance is the naive approach which does not profit of any query matching technique. As we can see in Figure 5 the group-sharing based approach is very close to the unrestricted sharing approach. This remark is valid for all database sizes and the difference in those two approaches remains insignificant. In contrast, the difference between the naive and the pure stacked approach compared to other two grows significantly as the size of the database increases. In a real data warehouse, it is often the case that MVs have indexes defined on them. The group-sharing-based refresh may outdo the unrestricted approach if there is occasion to exploit the indexes of MVs when used by the refreshing of the higher group MVs.

5 Conclusion and Future Work

We have addressed the problem of refreshing concurrently multiple MVs. In this context, two query matching techniques, query stacking and query sharing, are used in DB2 to exploit commonalities among the MVs, and to construct an efficient global evaluation plan. When the number of MVs is large, memory and time restrictions prevent us from using both query matching techniques in constructing efficient global plans. We have suggested an approach that applies the two techniques in different steps. The query stacking technique is applied first, and the generated subsumption DAG is used to define groups of MVs. The number of MVs in each group is smaller than the total number of MVs. This will allow the query sharing technique to be applied only within groups in a second step. Finally, the query stacking technique is used again to determine an efficient global evaluation plan. An experimental evaluation shows that the execution time of the optimal global plan generated by our approach is very close to that of the optimal global plan generated using, without restriction, both query matching techniques. This result is valid no matter how big the database is.

Our approach can be further fine-tuned to deal with the case where the groups of MVs turn out to be too small. In this case, merging smaller groups into bigger ones may further enhance the potential for applying the query sharing technique. Although we assume complete repopulation of all MVs in our approach for simplicity, we can actually apply our approach to incremental refreshing of MVs. In a typical data warehouse application, there usually exist indexes on MVs. Our approach can be extended to adapt to this scenario. Actually, because the existence of indexes increases the complexity of the global QGM, our approach may achieve better performance.

References

1. *TPC (Transaction Processing Performance Council) Web Site: http://www.tpc.org*.
2. Nathan Folkert, Abhinav Gupta, Andrew Witkowski, Sankar Subramanian, Srikanth Bellamkonda, Shrikanth Shankar, Tolga Bozkaya, and Lei Sheng. Optimizing Refresh of a Set of Materialized Views. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1043–1054, 2005.
3. Alon Y. Halevy. Answering Queries Using Views: A survey. *VLDB J.*, 10(4):270–294, 2001.
4. Wolfgang Lehner, Roberta Cochrane, Hamid Pirahesh, and Markos Zaharioudakis. fAST Refresh Using Mass Query Optimization. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 391–398, 2001.
5. Wolfgang Lehner, Richard Sidle, Hamid Pirahesh, and Roberta Cochrane. Maintenance of automatic summary tables. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 512–513, 2000.
6. Timos K. Sellis. Multiple-Query Optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.
7. Dimitri Theodoratos and Mokrane Bouzeghoub. A General Framework for the View Selection Problem for Data Warehouse Design and Evolution. In *DOLAP 2000, ACM Seventh International Workshop on Data Warehousing and OLAP, Washington, DC, USA, November 10, 2000, Proceedings*, pages 1–8, 2000.
8. Dimitri Theodoratos and Wugang Xu. Constructing Search Spaces for Materialized View Selection. In *DOLAP 2004, ACM Seventh International Workshop on Data Warehousing and OLAP, Washington, DC, USA, November 12-13, 2004, Proceedings*, pages 112–121, 2004.
9. Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex sql queries using automatic summary tables. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 105–116, 2000.
10. Daniel C. Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M. Lohman, Roberta Cochrane, Hamid Pirahesh, Latha S. Colby, Jarek Gryz, Eric Alton, Dongming Liang, and Gary Valentin. Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In *1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA*, pages 180–188, 2004.

Trademarks