A Dynamic View Materialization Scheme for Sequences of Query and Update Statements

Wugang Xu[1], Dimitri Theodoratos[1], Calisto Zuzarte[2], Xiaoying Wu[1], and Vincent Oria[1,⋆]

[1] New Jersey Institute of Technology
wx2@njit.edu, dth@cs.njit.edu, xw43@njit.edu, oria@njit.edu
[2] IBM Canada Ltd.
calisto@ca.ibm.com

Abstract. In a data warehouse design context, a set of views is selected for materialization in order to improve the overall performance of a given workload. Typically, the workload is a set of queries and updates. In many applications, the workload statements come in a fixed order. This scenario provides additional opportunities for optimization. Further, it modifies the view selection problem to one where views are materialized dynamically during the workload statement execution and dropped later to free space and prevent unnecessary maintenance overhead. We address the problem of dynamically selecting and dropping views when the input is a sequence of statements in order to minimize their overall execution cost under a space constraint. We model the problem as a shortest path problem in directed acyclic graphs. We then provide a heuristic algorithm that combines the process of finding the candidate set of views and the process of deciding when to create and drop materialized views during the execution of the statements in the workload. Our experimental results show that our approach performs better than previous static and dynamic approaches.

1 Introduction

Data warehousing applications materialize views to improve the performance of workloads of queries and updates. The queries are rewritten and answered using the materialized views [10]. A central issue in this context is the selection of views to materialize in order to optimize a cost function while satisfying a number of constraints [14,2]. The cost function usually reflects the execution cost of the workload statements that is, the cost of evaluating the workload queries using possibly the materialized views and the cost of applying the workload updates to the affected base relations and materialized views. The constraints usually express a restriction on the space available for view materialization [11], or a restriction on the maintenance cost of the materialized views [9], or both [13]. Usually the views are materialized before the execution of the first statement and remain materialized until the last statement is executed. This is the static view selection problem which has been studied extensively during the last decade [11,15,9]. Currently most of the commercial DBMSs (e.g. IBM DB2, MS SQL Server,

⋆ Research partially supported by a grant from the Army Research Laboratory.

Oracle) provide tools that recommend a set of views to materialize for a given workload of statements based on a static view materialization scheme [18,1,6].

If a materialized view can be created and dropped later during the execution of the workload, we face a dynamic version of the view selection problem. A dynamic view selection problem is more complex than its static counterpart. However, it is also more flexible and can bring more benefit since a materialized view can be dropped to free useful space and to prevent maintenance overhead. When there is no space constraint and there are only queries in the workload, the two view materialization schemes are the same: any view that can bring benefit to a query in the workload is materialized before the execution of the queries and never dropped.

Although in most view selection problems the workload is considered to be unknown or a set of statements without order, there are many applications where the workload forms a sequence of statements. This means that the statements in the workload are executed in a specific order. For example, in a typical data warehousing application, some routine queries are given during the day time of every weekday for daily reports; some analytical queries are given during the weekend for weekly reports; during the night the data warehouse is updated in response to update statements collected during the day. This is, for instance, a case where the workload is a sequence of queries and updates. Such a scenario is shown in Figure 1. The information on the order of the statements in the workload is important in selecting materialized views.

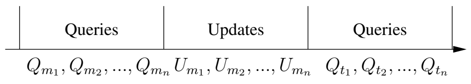
Queries
Updates
Queries
$Q_{m_1}, Q_{m_2}, ..., Q_{m_n}$
$U_{m_1}, U_{m_2}, ..., U_{m_n}$
$Q_{t_1}, Q_{t_2}, ..., Q_{t_n}$

Fig. 1. A workload as a sequence

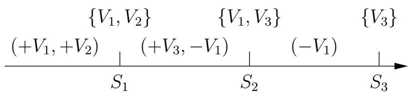

Fig. 2. Output of the problem

The solution to the view selection problem when the input is a sequence of statements can be described using a sequence of create view and drop view commands before the execution of every statement. An alternative representation determines the views that are materialized during the execution of every statement. Figure 2 shows these two representations. $+V_i$ $(-V_i)$ denotes the materialization (de-materialization) of view V_i.

In this paper we address the dynamic view selection problem when the workload is a sequence of query and update statements. This problem is more complex than the static one because we not only need to decide about which views to materialize, but also when to materialize and drop them with respect to the workload statements. Our main contributions are as follows:

1. We exploit the problem as a shortest path problem in a directed acyclic graph (DAG) [4]. Unlike that approach, our approach generate the DAG in a dynamic way. Therefore, it is autonomous in the sense that it does not rely on external modules for constructing the DAG.
2. In order to construct the nodes in the DAG, we extract "maximal" common subexpressions of queries and/or views. We also produce rewritings of the queries using

these common subexpressions thus avoiding the application of expensive processes that match queries to views.

3. We suggest a heuristic approach that controls the generation of nodes for the DAG based on nodes generated in previous steps.
4. We have implemented our approach and conducted an extensive experimental evaluation. Our results show that our approach performs better than previous static and dynamic approaches.

The next section reviews related work. In Section 3, we formally defined the problem. In Section 4, we present its modeling as a shortest path problem and provide an optimal solution. In Section 5, we show our heuristic approach for the case where no update statements are presented in the workload and in Section 6, we discuss how it can be extended to include also update statements. We present our experiment results in Section 7 and conclude in Section 8.

2 Related Work

In order to solve a view selection problem, one has to determine a search space of candidate views from which a solution view set is selected [16]. The most useful candidate views are the common subexpressions on queries since they can be used to answer more than one query. Common subexpression for pure group-by queries can be determined in a straightforward way [11]. In [8] this class of queries is extended to comprise, in addition, selection and join operations and nesting. In [5] the authors elaborate on how to common subexpressions of select-project-join queries without self-join can be found. This results are extended in [16] to consider also self-joins. Currently, most major commercial DBMSs provide utilities for constructing common subexpressions for queries [3,17]. More importantly, they provide utilities to estimate the cost of evaluating queries using materialized views. The What-If utility in Microsoft SQL Server [1] and the EXPLAIN utility in IBM DB2 [19] are two such examples.

One version of the dynamic view selection problem has been addressed in the past in [7] and [12]. Kotidis et al. [12] show that using a dynamic view management scheme, the solution outperforms the optimal solution of the static scheme. Both approaches focus on decomposing and storing the results of previous queries in order to increase the opportunity of answering subsequent queries partially or completely using these stored results. However, both approaches are different to ours since they assume that the workload of statements is unknown. Therefore, they focus on predicting what views to store in and what views to delete from the cache.

Agrawal et al. [4] consider a problem similar to ours. They model the problem as a shortest path problem for a DAG. However, their approach assumes that the candidate view set from which views are selected for materialization is given. This assumption is problematic in practice because there are too many views to consider. In contrast, our approach assumes that the input to the problem is a sequence of queries and update statements from which it constructs candidate views by identifying common subexpressions among the statements of the sequence.

3 Problem Specification

We assume a sequence $\mathcal{S} = (S_1, S_2, ..., S_n)$ of query and update statements is provided as input. The statements in the sequence are to be executed in the order they appear in the sequence. If no views are materialized, the total execution cost of $\mathcal{S}$ includes (a) the cost of evaluating all the queries in $\mathcal{S}$ over the base relations, and (b) the cost of updating the base relations in response to the updates in $\mathcal{S}$. Let's now assume that we create and materialize a view just before the execution of statement S_i and drop it before the execution of statement S_j. Then, the total execution cost includes (a) the cost of evaluating the queries in $(S_1, ..., S_{i-1})$ and in $(S_j, ..., S_n)$ over the base relations, (b) the cost of materializing view V (i.e., the cost of computing and storing V), (c) the cost of evaluating all queries in $(S_i,..., S_{j-1})$ *using the materialized view* V (V is used only if it provides some benefit), (d) the cost of updating view V in response to the changes of the base relations resulting by the update statements in $(S_i,..., S_{j-1})$, and (e) the cost of updating the base relations in response to the updates in $\mathcal{S}$. We ignore here for simplicity the cost of dropping the materialized view V. The total execution cost of $\mathcal{S}$ when multiple materialized views are created and dropped on different positions of sequence $\mathcal{S}$ is defined in a similar way. Since the cost of updating the base relations in response to updates in $\mathcal{S}$ is fixed, we consider it an overhead cost, and we ignore it in the following.

The problem we are addressing in this paper can be now formulated as follows. Given a sequence of n queries and updates $(S_1, S_2, ..., S_n)$ and a space constraint B, find a sequence $(O_1, O_2, ..., O_n)$ of sets of "create view" and "drop view" statements such that (a) the total execution cost of $\mathcal{S}$ is minimized and, (b) the space used to materialize views during the execution of $\mathcal{S}$ does not exceed B. Each create view statement contains also the definition of the view to be created. The set O_i of create view and drop view statements is executed before the execution of the statement S_i

The output of this dynamic view selection problem can also be described by a sequence of sets of views $\mathcal{C} = (C_1, C_2, ..., C_n)$. Each set C_i contains the views that have been created and not dropped before the evaluation of the statement S_i. We call $\mathcal{C}$ a solution to the problem. Further, if the views to be materialized can only be selected from a set of candidate views $\mathcal{V}$, we call C a solution to problem for the view set $\mathcal{V}$.

4 Modeling the Dynamic View Selection Problem

In this section, we show how the dynamic view selection problem for a sequence of query and update statements can be modeled as a shortest path problem on a directed acyclic graph. We follow the approach introduced by [4]. That approach assumes that the set of candidate views (i.e. the pool from which we can choose view to materialize) is given. Our approach is different. As we show later in this section, the candidate views are dynamically constructed from the workload statement by considering common subexpressions among them.

We show below how [4] models the problem assuming that a candidate set of views $\mathcal{V}$ is given and contains only one view V. Then, there are only two options for each statement S_i in the workload: either the view V is materialized or not. Those two options are represented as $C_i^0 = \{\}$ and $C_i^1 = \{V\}$ respectively in a solution where C_i is the set of views that are materialized before the execution of S_i. We can now construct a directed acyclic graph (DAG) as follows:

1. For each statement S_i in the workload, create two nodes N_i^0 and N_i^1 which represent the execution of statement S_i without and with the materialized view V respectively. Create also two virtual nodes N_0 and N_{n+1} which represent the state before and after the execution of the workload respectively. Label the node N_i^0 by the empty set $\{\}$ and the node N_i^1 by the set $\{V\}$.
2. Add an edge from each node of S_i to each node of S_{i+1} to represent the change in the set of materialized views. If both the source node and the target node are labeled by the same set, then label the edge by a empty sequence "()". If the source node is labeled by $\{\}$ and the target node is labeled by $\{V\}$, then label the edge by a sequence $(+V)$ to represent the operation "create materialized view V". If the source node is labeled by $\{V\}$ and the target node is labeled by $\{\}$, then label the edge by $(-V)$ to represent the operation "drop materialized view V".
3. Compute the cost of each edge from a node of S_i to a node of S_{i+1} as the sum of: (a) the cost of materializing V, if a $(+V)$ labels the edge, (b) the cost of dropping V, if $(-V)$ labels the edge, and (c) the cost of executing the statement S_{i+1} using the set of views that label the target node of the edge.

Figure 3 shows the DAG constructed the way described above. Each path from the node N_0 to the node N_{n+1} represents a possible execution option for the workload. The shortest path among them represents the optimal solution for the dynamic view selection problem. The labels of the edges in the path denote the solution represented as a sequence of "create view" and "drop view operations". The labels of nodes in the path denote the solution represented as a sequence of sets of materialized views.

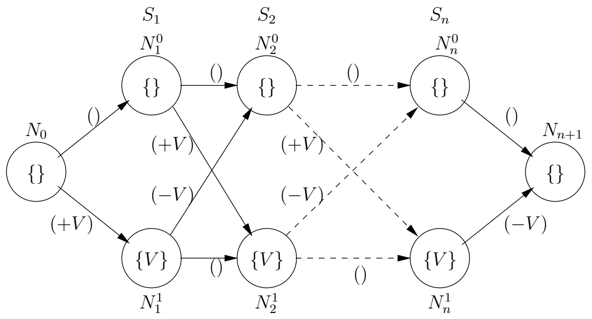

Fig. 3. Directed acyclic graph for candidate view set $\{V\}$

For a candidate set $\mathcal{V}$ of m views, we can construct a DAG in a similar way. Instead of having two nodes for each statement in the workload, we create 2^m nodes, one for each subset of $\mathcal{V}$. Again, the shortest path represents the optimal solution for the dynamic view selection problem.

The shortest path problem for a DAG can be solved by well known algorithms in linear time on the number of edges of the DAG. Therefore the complexity of the process is $O(n \cdot 2^{2m})$ where m is the number of candidate views and n is the number of statements in the workload. To compute the cost of each edge, an optimizer can be used to assess the cost of executing the target statement using the corresponding set of materialized views. This is too expensive even for a small number of candidate views. In practice, the set of candidate views is expected to be of substantial size. The dynamic view selection problem is shown to be NP-hard [4]. Therefore, a heuristic approach must be employed to efficiently solve this problem.

In practice, we cannot assume that the set of candidate views is given. It has to be constructed. Tools that suggest candidate views for a static view selection problem [4] are not appropriate for determining candidate views for a dynamic view selection problem. Our approach constructs candidate views which are common subexpressions of queries in the workload and of views. More specifically, we consider subexpressions of two queries which represent the maximum commonalities of these two queries as these are defined in [16]. An additional advantage of this approach is that the rewritings of the queries using the common subexpressions are computed along with the common subexpressions. These rewritings can be fed to the query optimizer to compute the cost of executing the queries using the materialization of the common subexpressions. If a view V is defined as a common subexpression of a set of queries $\mathcal{Q}$, we call each query $Q \in \mathcal{Q}$ a parent of the V. Every $Q \in \mathcal{Q}$ can be answered using V. We consider only rewritings of a query Q using its common subexpression with other queries and views. A major advantage of this approach is that we do not have to check whether a query matches a view (i.e., check if there is a rewriting of the query using the view) which is in general an expensive process. In the following, we ignore a rewriting of a query using a common subexpression if the cost of evaluating this rewriting is not less than the cost of evaluating the query over base relations.

To solve the dynamic view selection problem, we can generate different views that are common subexpressions of subsets of the queries in the workload. Finding the solution for this problem using the shortest path algorithm is expensive because of the large number of candidate views and the large number of nodes in the DAG. In the next section, we introduce a heuristic approach that combines the process of generating candidate views with the process of selecting views for a solution of the view selection problem.

5 A Heuristic Approach

For the heuristic approach, we start by considering that there are only queries in the workload. In the next section, we discuss how our method can be extended to the case where there are also update statements in the workload.

Our heuristic approach uses two solution merging functions $Merge1$ and $Merge2$. Each function takes as input two solutions to the dynamic view selection problem, each one for a specific set of candidate views, and outputs a new solution.

Consider two solutions $l_1 = (C_1^1, C_2^1, ..., C_n^1)$ and $l_2 = (C_1^2, C_2^2, ..., C_n^2)$, for the candidate view sets $\mathcal{V}_1$ and $\mathcal{V}_2$ respectively. Function $Merge1$ is analogous to function $UnionPair$ [4]. It first creates a DAG as follows: for each statement S_i in the workload, it creates two nodes, one labeled by C_i^1, the other labeled by C_i^2. If the new view set

Function 1. $Merge2$

```
Input: solution $l_1$, solution $l_2$, space constraint $B$
Output: solution $l$
 1: Create a list of solutions $\mathcal{L}$ which initially contains $l_1$ and $l_2$
 2: for each view $V_1$ in $l_1$ and each view $V_2$ in $l_2$ do
 3:     Find the set of common subexpressions $\mathcal{V}_{12}$ of $V_1$ and $V_2$
 4:     for each view $V \in \mathcal{V}_{12}$ do
 5:         Find the solution for the candidate view set $\{V\}$ and add it into $\mathcal{L}$
 6:     end for
 7: end for
 8: Find the solution $l$ from $\mathcal{L}$ with the lowest execution cost and remove it from $\mathcal{L}$
 9: for all solutions in $\mathcal{L}$ do
10:     Find the solution $l'$ with the lowest execution cost and remove it from $\mathcal{L}$
11:     $l = Merge1(l, l')$
12: end for
13: Return the solution $l$
```

$C_i = C_i^1 \cup C_i^2$ satisfies the space constraint B, it also creates another node labeled by C_i. In addition it creates two virtual nodes representing the starting and ending states. Finally it creates edges as explained earlier from nodes of query S_i to nodes of query S_{i+1}. Once the DAG is constructed, it returns the solution corresponding to the shortest path in the DAG.

Function $Merge1$ does not add new views to the set of candidate views. Instead, for each statement, it might add one more alternative which is the union of two view sets. In contrast, function $Merge2$ shown in the previous page introduces new views into the set of candidate views.

Our heuristic approach for the dynamic view selection problem is implemented by Algorithm 2 shown above. Algorithm 2 first generates an initial set of candidate views which are common subexpressions of each pair of queries. For each view, it finds a solution. Then, it uses Function $Merge2$ to introduce new views into the set of candidate views.

Algorithm 2. A heuristic algorithm for the dynamic view selection problem

```
Input: list of queries $\mathcal{S}$, space constraint $B$
Output: solution $l$
 1: Create a set of solutions $\mathcal{L}$ which is initially empty
 2: for each pair of queries $S_i$ and $S_j \in \mathcal{S}$ do
 3:     Find the set of common subexpressions $\mathcal{V}_{ij}$ of $S_i$ and $S_j$
 4:     Find the solution for each view set $\mathcal{V} = \{V\}$ where $V \in \mathcal{V}_{ij}$ and add it into $\mathcal{L}$
 5: end for
 6: Find the solution $l$ from $\mathcal{L}$ with the lowest execution cost and remove it
 7: for all solutions in $\mathcal{L}$ do
 8:     Find the solution $l'$ lowest execution cost and remove it from $\mathcal{L}$
 9:     $l = Merge2(l, l')$
10: end for
11: Return the solution $l$
```

6 Considering Update Statements in the Sequence

In general, an update statement that involves updating more than one base relations can be modeled by a sequence of update statements each of which updates one base relation. For the updates we follow an incremental view maintenance strategy.

Let us assume that a view V contains an occurrence of the base relation R, and an update statement U in the workload updates R. Then, if V is materialized when U is executed, V has to be updated. This incurs a maintenance cost. An optimizer can assess the cost for maintaining U. Roughly speaking, we define the unaffected part of a view V with respect to an update U to be the set of subexpressions of V resulting by removing R from V. If an expression of the unaffected part of a view is materialized or if it can be rewritten using another materialized view, the maintenance cost of V can be greatly reduced. For example, the unaffected part of a view $V = R \bowtie \sigma_{A>10}(S) \bowtie \sigma_{C=0}(T)$ with respect to R is the view $V' = \sigma_{A>10}(S) \bowtie \sigma_{C=0}(T)$. View V can be maintained in response to changes in R either incrementally or through re-computation much more efficiently if V' is materialized.

If there are update statements in the sequence, we still start the heuristic approach with a set of candidate views which are common subexpressions of pairs of queries in the workload sequence. Then, for each view and each update statement, we add to the candidate set of views the unaffected parts of the view with respect to the updates. If two update statements update the same base relation, only one is used to generate the unaffected parts of a view. Finally we apply the heuristic algorithm using the new set of views as a candidate view set.

7 Experimental Evaluation

We implemented the heuristic algorithm for the dynamic view selection problem for a sequence of query and update statements. In order to examine the effectiveness of our algorithm, we also implemented a static view selection algorithm similar to the one presented in [18]. Further, we implemented the greedy heuristic algorithm $GREEDY-SEQ$ presented in [4]. We provide as input to $GREEDY-SEQ$ a set of candidate views recommended by the static view selection algorithm. We consider select-project-join queries. We compute "maximal" common subexpressions of two queries using the concept of closest common derivator as is defined in [16]. In order to deal with update statements, we take into account the unaffected parts of the views with respect to the update statements. We use a cost model that assesses the cost of a query (or an update statement) when this is rewritten completely or partially using one or more materialized views.

We measure the performance of each approach as a percentage using the percentage of the total execution cost of a workload using the set of materialized view suggest by the approach to the execution cost of the same workload without using any view. For each experiment, we consider three schemes, $Static$ (the static view selection approach), $Dynamic$ (the view selection approach presented in this paper), $GREEDY$-SEQ (the algorithm in [4] fed with the result of Static).

First, we study the effect of the space constraint on the three algorithms. We consider two workloads W_1 and W_2 each of which consists of 25 queries (no updates). The

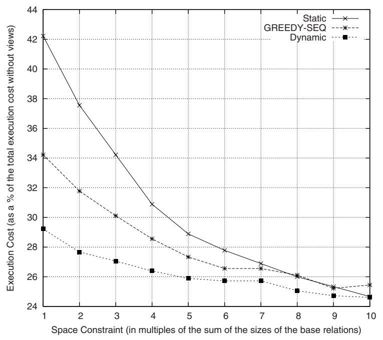

Fig. 4. Performance vs. space constraint, workload W_1

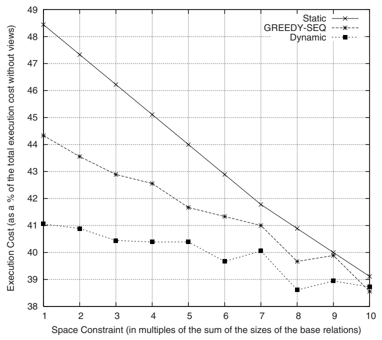

Fig. 5. Performance vs. space constraint, workload W_2

queries in W_1 have more overlapping than that of W_2. The space constraint varies from 1 to 10 times the total size of the base relations. The results are shown in Figures 4 and 5 for W_1 and W_2 respectively. When the space constraint is restrictive, the dynamic view selection schemes have better performance than the static one. This superiority is the result of the capacity of these approaches for creating and dropping materialized views dynamically. As expected, when the space constraint relaxes, all view selection schemes generate similar results. Among the two dynamic view selection approaches, $Dynamic$ performs better than the $GREEDY - SEQ$ algorithm. This shows that a statically selected view set is not appropriate for a dynamic view selection scheme. Our approach does not suffer from this shortcoming since its set of candidate views is constructed dynamically.

Then, we consider the effect of update statements on the three approaches. We consider two series of workloads WS_1 and WS_2, each workload contains the same 25 queries. However we vary the number of update statements in each workload from 1 to 10. Each update statement updates 30% tuples of base relation chosen randomly. In

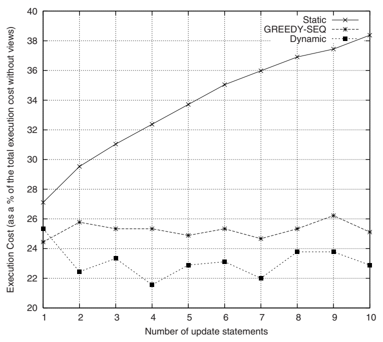

Fig. 6. Performance vs. number of update statements (workload Ws_1)

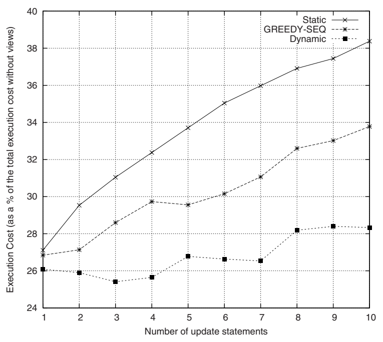

Fig. 7. Performance vs. space constraint (workload Ws_{1p})

the workloads of WS_1, all the update statements follow all the queries. In the workloads of WS_2, the update statements are interleaved randomly with the queries. The space constraint is fixed to 10 times the total size of the base relations. The results are shown in Figures 6 and 7 respectively. In all cases, when the number of update statements in the workload increases, the dynamic view selection approaches perform better compared to the static one. This is expected since the dynamic algorithms can drop materialized views before the evaluation of update statements and save the maintenance time of these views. The static view selection scheme does not depend on the order of query and update statements in the workload. Thus, for both workload series, the static view selection scheme performs the same. The dynamic view selection scheme depends on the order of query and update statements in the workload. When the update statements follow the queries, the dynamic view selection schemes perform better. The reason is that materialized views that are needed for queries are dropped after the queries are executed and therefore do not contribute to the maintenance cost. In any case, the $Dynamic$ outperforms the $GREEDY-SEQ$.

8 Conclusion

We addressed the problem of dynamically creating and dropping materialized views when the workload is a sequence of query and update statements. We modeled it as a shortest path problem in DAGs where the nodes of the DAG are dynamically constructed by exploiting common subexpressions among the query and update statements in the workload. We designed a heuristic algorithm that combines the process of finding the candidate set of views and the process of deciding when to create and drop materialized views during the execution of the statements in the workload. An experimental evaluation of our approach showed that it performs better than previous static and dynamic ones. We are currently working towards studying alternative algorithms and we are also addressing a similar problem where the input query and update statements form a partial order.

References

1. Agrawal, S., Chaudhuri, S., Kollár, L., Marathe, A., Narasayya, V., Syamala, M.: Database Tuning Advisor for Microsoft SQL Server 2005. In: Proc. of 30th Int. Conf. on VLDB (2004)
2. Yu, S., Atluri, V., Adam, N.R.: Selective View Materialization in a Spatial Data Warehouse. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2005. LNCS, vol. 3589, Springer, Heidelberg (2005)
3. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated Selection of Materialized Views and Indexes in SQL Databases. In: Proc. of 26th VLDB (2000)
4. Agrawal, S., Chu, E., Narasayya, V.R.: Automatic physical design tuning: workload as a sequence. In: Proc. ACM SIGMOD Int. Conf. on Management of Data (2006)
5. Chen, F.-C.F., Dunham, M.H.: Common Subexpression Processing in Multiple-Query Processing. IEEE Trans. Knowl. Data Eng. 10(3) (1998)
6. Dageville, B., Das, D., Dias, K., Yagoub, K., Zaït, M., Ziauddin, M.: Automatic SQL Tuning in Oracle 10g. In: Proc. of VLDB (2004)

7. Deshpande, P., Ramasamy, K., Shukla, A., Naughton, J.F.: Caching Multidimensional Queries Using Chunks. In: Proc. ACM SIGMOD (1998)
8. Golfarelli, M., Rizzi, S.: View materialization for nested GPSJ queries. In: Proc. Int. Workshop on Design and Management of Data Warehouses (2000)
9. Gupta, H., Mumick, I.S.: Selection of Views to Materialize Under a Maintenance Cost Constraint. In: Proc. 7th Int. Conf. on Database Theory (1999)
10. Halevy, A.Y.: Answering Queries Using Views: A survey. The International Journal on Very Large Data Bases 10(4), 270–294 (2001)
11. Harinarayan, V., Rajaraman, A., Ullman, J.D.: Implementing Data Cubes Efficiently. In: Proc. ACM SIGMOD Int. Conf. on Management of Data (1996)
12. Kotidis, Y., Roussopoulos, N.: DynaMat: A Dynamic View Management System for Data Warehouses. In: Proc. ACM SIGMOD Int. Conf. on Management of Data (1999)
13. Mistry, H., Roy, P., Sudarshan, S., Ramamritham, K.: Materialized View Selection and Maintenance Using Multi-Query Optimization. In: Proc. ACM SIGMOD (2001)
14. Theodoratos, D., Ligoudistianos, S., Sellis, T.K.: View selection for designing the global data warehouse. Data Knowl. Eng. 39(3) (2001)
15. Theodoratos, D., Sellis, T.K.: Data Warehouse Configuration. In: Proc. 23rd Int. Conf. on Very Large Data Bases (1997)
16. Theodoratos, D., Xu, W.: Constructing Search Spaces for Materialized View Selection. In: Proc. ACM 7th Int. Workshop on Data Warehousing and OLAP (2004)
17. Zaharioudakis, M., Cochrane, R., Lapis, G., Pirahesh, H., Urata, M.: Answering Complex SQL Queries Using Automatic Summary Tables. In: Proc. ACM SIGMOD, ACM Press, New York (2000)
18. Zilio, D., Rao, J., Lightstone, S., Lohman, G., Storm, A., Garcia-Arellano, C., Fadden, S.: DB2 Design Advisor: Integrated Automatic Physical Database Design. In: Proc. VLDB (2004)
19. Zilio, D., Zuzarte, C., Lightstone, S., Ma, W., Lohman, G., Cochrane, R., Pirahesh, H., Colby, L., Gryz, J., Alton, E., Liang, D., Valentin, G.: Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In: Proc. Int. Conf. on Autonomic Computing (2004)