Materialized View Selection and Maintenance Using Multi-Query Optimization[*]

Hoshi Mistry[1] Prasan Roy[2†] S. Sudarshan[1] Krithi Ramamritham[1,3]
[1]IIT-Bombay [2]Bell Labs [3]Univ. of Massachusetts-Amherst
hoshi_mistry@vsnl.com, prasan@research.bell-labs.com, {sudarsha,krithi}@cse.iitb.ernet.in

ABSTRACT
Materialized views have been found to be very effective at speeding up queries, and are increasingly being supported by commercial databases and data warehouse systems. However, whereas the amount of data entering a warehouse and the number of materialized views are rapidly increasing, the time window available for maintaining materialized views is shrinking. These trends necessitate efficient techniques for the maintenance of materialized views.

In this paper, we show how to find an efficient plan for the maintenance of a *set of materialized views*, by exploiting common subexpressions between different view maintenance expressions. In particular, we show how to efficiently select (a) expressions and indices that can be effectively shared, by *transient materialization*; (b) additional expressions and indices for *permanent materialization*; and (c) the best maintenance plan – *incremental* or *recomputation* – for each view. These three decisions are highly interdependent, and the choice of one affects the choice of the others. We develop a framework that cleanly integrates the various choices in a systematic and efficient manner. Our evaluations show that many-fold improvement in view maintenance time can be achieved using our techniques. Our algorithms can also be used to efficiently select materialized views to speed up workloads containing queries and updates.

1. INTRODUCTION
Materialized views have been found to be very effective in speeding up query, as well as update processing, and are increasingly being supported by commercial database systems. Materialized views are especially attractive in data warehousing environments because of the query intensive nature of data warehouses. However, when a warehouse is updated, the materialized views must also be updated. Typically, updates are accumulated and then applied to a data warehouse. While the need to provide up-to-date responses to an increasing query load is growing and the amount of data that gets added to data warehouses has been increasing, the time window available for making the warehouse up-to-date has been shrinking. These trends call for efficient techniques for maintaining the materialized views as and when the warehouse is updated.

The view maintenance problem can be seen as computing the expressions corresponding to the "delta" of the views, given the "delta"s of the base relations that are used to define the views. It is not difficult to motivate that query optimization techniques are important for choosing an efficient plan for maintaining a view, as shown in [15]. For example, consider the materialized view $V = (A \bowtie B \bowtie C)$. We assume, as in SQL, that relations A, B and C are multisets (i.e., relations with duplicates). Given that the multiset of tuples δ_C^+ is inserted into C, the change to the materialized view V consists of a set of tuples $(A \bowtie B) \bowtie \delta_C^+$ to be inserted into V. This expression can equivalently be computed as $(A \bowtie \delta_C^+) \bowtie B$ and by $(B \bowtie \delta_C^+) \bowtie A$, one of which may be substantially cheaper to compute. Further, in some cases the view may be best maintained by recomputing it, rather than by finding the differentials as above.

Our work addresses the problem of optimizing the maintenance of a *set* of materialized views. If there are multiple materialized views, as is common, significant opportunities exist for sharing computation between the maintenance of different views. Specifically, common subexpressions between the view maintenance expressions can reduce maintenance costs greatly.

Whether or not there are multiple materialized views, significant benefits can be had in many cases by materializing extra views or indices, whose presence can decrease maintenance costs significantly. The choice of what to materialize permanently depends on the choice of view maintenance plans, and vice versa. The choices of the two must therefore be closely coupled to get the best overall maintenance plans.

[*]Work partly supported by a Govt. of India, Department of Science and Technology Grant, and by an IBM University Partnership Program Grant. The work of Prasan Roy was supported by an IBM Research Fellowship. Ramamritham was also supported in part by NSF grant IRI-9619588.
[†]Work done while the author was at IIT-Bombay.

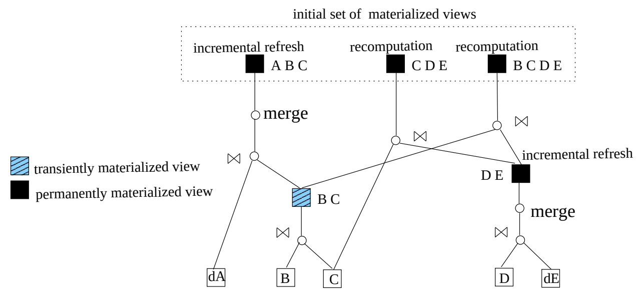


Figure 1: Example view maintenance plan. *Merge* refreshes a view given its "delta".

To motivate the techniques we propose, consider the following example.

EXAMPLE 1.1. Suppose we have three materialized views $V1 = (A \bowtie B \bowtie C)$, $V2 = (C \bowtie D \bowtie E)$ and $V3 = (B \bowtie C \bowtie D \bowtie E)$, and relations A and E are updated due to inserts. If the maintenance plans of the three views are chosen independently, the best view maintenance plan (incremental or recomputation) for each would be chosen, without any sharing of computation.

In contrast, as an illustration of the kind of plans our optimization methods are able to generate, Figure 1 shows a maintenance plan for the views that exploits sharing of computation. Here, $(A \bowtie B \bowtie C)$ is refreshed incrementally, while $(C \bowtie D \bowtie E)$ and $(B \bowtie C \bowtie D \bowtie E)$ are recomputed. Two extra views, $(B \bowtie C)$ and $(D \bowtie E)$ have been chosen to be materialized. Of these, $(B \bowtie C)$ is materialized *transiently*, and is disposed as soon as the views are refreshed; this could happen because there are also updates on B and C which make it expensive to maintain $(B \bowtie C)$ as a materialized view. The result $(D \bowtie E)$ has been chosen to be materialized *permanently*, and is itself refreshed incrementally given the updates to the relation E. Its full result is then used to recompute $(C \bowtie D \bowtie E)$ as well as $(B \bowtie C \bowtie D \bowtie E)$. □

Contributions. The contributions of this paper are as follows:

1. *We show how to exploit transient materialization of common subexpressions to reduce the cost of view maintenance plans.*

   Sharing of subexpressions occurs when multiple views are being maintained, since related views may share subexpressions, and as a result the maintenance expressions may also be shared. Furthermore, sharing can occur even within the plan for maintaining a single view if the view has common subexpressions within itself. The shared expressions could include differential expressions, as well as full expressions which are being recomputed.

   Here, *transient materialization* means that these results are materialized during the evaluation of the maintenance plan and disposed on its completion.

2. *We show how to efficiently choose additional expressions for permanent materialization to speed up maintenance of the given views.*

   Just as the presence of views allows queries to be evaluated more efficiently, the maintenance of the given permanently materialized views can be made more efficient by the presence of additional permanently materialized views [12, 11]. That is, given a set of materialized views to be maintained, we choose additional views to materialize in order to minimize the overall view maintenance costs.

   The expressions chosen for permanent materialization may be used in only one view maintenance plan, or may be shared between different views maintenance plans. We outline differences between our work and prior work in this area, in Section 2.

3. *We show how to determine the optimal maintenance plan for each individual view, given the choice of results for transient/permanent materialization*.

   Maintenance of a materialized view can either be done *incrementally* or *by recomputation*. Incremental view maintenance involves computing the differential ("delta"s) of a materialized view, given the "delta"s of the base relations that are used to define the views, and merging it with the old value of the view. However, incremental view maintenance may not always be the best way to maintain a materialized view; when the deltas are large the view may be best maintained by recomputing it from the updated base relations.

   Our techniques determine the maintenance policy, incremental or recomputation, for each view in the given set such that the overall combination has the minimum cost.

4. *We show how to make the above three choices in an integrated manner to minimize the overall cost.*

   We propose a framework that cleanly integrates the choice of additional views to be transiently or permanently materialized, the choice of whether each of the given set of (user-specified) views must be maintained incrementally or by recomputation, and the choice of view maintenance plans.

5. We have implemented all our algorithms, and present a performance study, using queries from the TPC-D benchmark, showing very significant benefits due to our techniques.

Although the focus of our work is to speed up view maintenance, and we assume an initial set of views have been chosen to be materialized, our algorithms can also be used to choose extra materialized views to speed up a workload containing queries and updates.

Paper Organization. Related work is outlined in Section 2. Section 3 gives an overview of the techniques presented in this paper. Section 4 describes our system model, and how the search space of the maintenance plans is set up. Section 5 shows how to compute the optimal maintenance cost for a given set of permanently materialized views, and a given set of views to be transiently materialized during the maintenance. Section 6 describes a heuristic that uses this cost calculation to determine the set of views to be transiently or permanently materialized so as to minimize the overall maintenance cost. Section 7 outlines results of a performance study, and Section 8 concludes the paper.

2. RELATED WORK

In the past decade, there has been a large volume of research on view maintenance, transiently materialized view selection (also known as multi-query optimization) and also on permanently materialized view selection. This work is summarized below. However, each of these problems have been addressed independently since the concerns were considered to be orthogonal; no prior work, to the best of our knowledge, has looked at addressing all of these problems in an integrated manner.

View Maintenance. Amongst the early work on computing the differential results of operations/expressions was Blakeley et al. [2]. Gupta and Mumick [7] provide a survey of view maintenance techniques.

Vista [15] describes how to extend the Volcano query optimizer [5] to compute the best maintenance plan, but does not consider the materialization of expressions, whether transient or permanent. [10] and [15] propose optimizations that exploit knowledge of foreign key dependencies to detect that certain join results involving differentials will be empty. Such optimizations are orthogonal and complementary to our work.

Transiently Materialized View Selection (Multi-Query Optimization). Blakeley et al. [2] and Ross et al. [11] noted that the computation of the expression differentials has the potential for benefiting from multi-query optimization [14]. In the past, multi-query optimization was viewed as too expensive for practical use. As a result, [2] and [11] do not go beyond stating that multi-query optimization could be useful for view maintenance. Our recent work in [13] provides efficient heuristic algorithms for multi-query optimization, and demonstrates that multi-query optimization is feasible and effective.

However, none of the work on multi-query optimization considers updates or view maintenance, which is the focus of this paper. Using these techniques naively on differential maintenance expressions would be very expensive, since incremental maintenance expressions can be very large. We utilize the optimizations proposed by [13], but significant extensions are required to to take update costs into account, and to efficiently optimize view maintenance expressions.

Permanently Materialized View Selection. There has been much work on selection of views to be materialized. One notable early work in this area was by Roussopolous [12]. Ross et al. [11] considered the selection of extra materialized views to optimize maintenance of other materialized views/assertions, and mention some heuristics. The problem of materialized view selection for data cubes has seen much work, such as [9], who propose a greedy heuristic for the problem. Gupta [8] extends some of these ideas to a wider class of queries. Agrawal et al. [1] present heuristics for materialized view selection.

The major differences between our work and the above work on materialized view selection can be summarized as follows:

1. Earlier work in this area has not addressed optimization of view maintenance plans in the presence of other materialized views. Earlier work simply assumes that the cost of view maintenance for a given set of materialized views can be computed, without providing any details.

2. Earlier work does not consider how to exploit common subexpressions by temporarily materializing them because of their focus on permanent materialization. In particular, common subexpressions involving differential relations cannot be permanently materialized.

3. Earlier work does not cover efficient techniques for the implementation of materialized view selection algorithms, and their integration into state-of-the-art query optimizers. Showing how to do the above is amongst our important contributions.

3. OVERVIEW OF OUR APPROACH

We extend the Volcano query optimization framework [5] to generate optimal maintenance plans. This involves the following subproblems:

1. *Setting up the Search Space of Maintenance Plans*

   We extend the Query DAG representation of [5] and [13], which represents just the space of recomputation plans, to include the space of incremental plans as well. This new extension uses *propagation-based differential generation*, which propagates the effect of one delta relation at a time in a predefined order. Our approach has a lower space cost of optimization as compared to using incremental view maintenance expressions, and is easier to implement.

   Propagation-based differential generation is explained in Section 4.1, and the extended Query DAG generation is explained in Section 4.2.

2. *Choosing the Policy for Maintenance and Computing the Cost of Maintenance*

   We show how to compute the minimum overall maintenance cost of the given set of permanently materialized views, given a fixed set of additional views to be transiently materialized. In addition to computing the cost, the proposed technique generates the best consolidated maintenance plan for the given set of permanently materialized views. The maintenance plan chosen for each materialized view can be incremental or recomputation, based on costs.

   Maintenance cost computation is explained in Section 5.

3. *Transient/Permanent Materialized View Selection*

   Finally, we address the problem of determining the respective sets of transient and permanently materialized views that minimize the overall cost. Our technique uses, as a subroutine, the previously mentioned technique for computing the best maintenance policy given fixed sets of permanently and temporarily materialized views. The costs of materialization of transiently materialized views and maintenance of permanently materialized views are taken into account by this step.

We propose a greedy heuristic that iteratively picks up views in order of benefit – where benefit is defined as the decrease in the overall materialization cost if this view is transiently or permanently materialized in addition to the views already chosen. Then, depending upon whether transient or permanent materialization of the view produces the greater benefit, the view is categorized as such.

The greedy heuristic is presented in Section 6.1, and several optimizations of this heuristic that result in an efficient implementation are described in Section 6.2.

4. SETTING UP THE MAINTENANCE PLAN SPACE

In this section, we describe how the search space of maintenance plans is set up. As mentioned earlier, our approach to incremental maintenance is based on the compact propagation-based differential generation technique. This is described in Section 4.1. The Query DAG representation, used to represent the search space compactly, is described in Section 4.2.

In this paper, we assume that we are given an initial set of permanently materialized views. We may add more views to this set. We do not consider space limitations on storing materialized views in the main part of the paper, but address this issue in Section 6.3.

We assume that the updates (inserts/deletes) to relations are logged in corresponding *delta* relations, which are made available to the view refresh mechanism; for each relation R, there are two relations δ_R^+ and δ_R^- denoting, respectively, the (multiset of) tuples inserted into and deleted from the relation R. The maintenance expressions in our examples assume that the old value of the relation is available, but we can use maintenance expressions based on the new values of the relations in case the updates have already been performed on the base relations.

We assume that the given set of materialized views is refreshed at times chosen by users, which are typically at regular intervals. For optimization purposes, we need estimates of the sizes of these delta relations. In production environments, the rates of changes are usually stable across refresh periods, and these rates can be used to make decisions on what relations to materialize permanently. We will assume that the average insert and delete sizes for each relation are provided as percentages of the full relation size. The insert and delete percentages can be different for different relations. Other statistics, such as number of new distinct values for attributes (in each refresh interval), if available, can also be used to improve the cost estimates of the optimizer.

4.1 Propagation-Based Differential Generation for Incremental View Maintenance

We generate the differential of an expression by propagating differentials of the base relations up the *expression tree*, one relation at a time, and only one update type (insertions or deletions) at a time. The differential propagation technique we use is based on the techniques used in [12] and [11].

The differential of a node in the tree is computed using the differential (and if necessary, the old value) of its inputs. We start at the leaves of the tree (the base relations), and proceed upwards, computing the differential expressions corresponding to each node.

For instance, the differential of a join $E_1 \bowtie E_2$, given inserts on relation R, is computed using the differentials of E_1 and E_2 and the old full results of E_1 and E_2. The differential result is empty if R is used in neither E_1 nor E_2. If R is used only in E_1, the differential is given by $(\delta_{E_1} \bowtie E_2)$; symmetrically if R is used only in E_2, the differential is given by $(E_1 \bowtie \delta_{E_2})$. If R is used in both, the differential consists of $(\delta_{E_1} \bowtie E_2) \cup (E_1 \bowtie \delta_{E_2}) \cup (\delta_{E_1} \bowtie \delta_{E_2})$.

The process of computing differentials starts at the bottom, and proceeds upwards, so when we compute the differential to $E_1 \bowtie E_2$, the differentials of the inputs have been computed already. The full results are computed when required, if they are not available already (materialized views and base relations are available already).

Extending the above technique to operations other than join is straightforward, using standard techniques for computing the differentials of operations, such as those in [2]; see [7] for a survey of view maintenance techniques.

Our search space includes differentials of all plans equivalent to $E_1 \bowtie E_2$. In the case of joins, in particular, the search space will include plans where every intermediate result includes the differential of R. To illustrate this point, consider the view $(A \bowtie B \bowtie C)$. If we wish to compute the differential of the view when tuples are inserted into A, then the plans $(B \bowtie (\delta_A^+ \bowtie C))$ and $(\delta_A^+ \bowtie (B \bowtie C))$ would both be among the plans considered, and the cheapest plan is selected. Similarly, if we wish to compute the differential of the view when tuples are inserted into B, then the plans $(A \bowtie (\delta_B^+ \bowtie C))$ and $(\delta_B^+ \bowtie (A \bowtie C))$ would be amongst the alternatives. Using the differentials of a single expression, such as $(A \bowtie (B \bowtie C))$ or $(B \bowtie (A \bowtie C))$, is not preferable for propagating all the base relation differentials.

Our optimizer's search space includes all of the alternatives for computing the differentials to $(A \bowtie B \bowtie C)$, including the above two, and the cheapest one is chosen for propagating the differential of each base relation.

Propagating differentials of only one type (inserts or deletes) to one relation at a time, simplifies choosing of a separate plan for each differential propagation. It is straightforward to extend the techniques to permit propagation of inserts and deletes to a single relation together, to reduce the number of different expressions computed.

We assume that the updates to the base relations are propagated one relation at a time. After each one is propagated, the base relation is itself updated, and the computed differentials are applied to all incrementally maintained materialized views.[1] We leave unspecified the order in which the base relations are considered. The order is not expected to have a significant effect when the deltas of all the relations are small percentages of the relation sizes: the relation statistics then do not change greatly due to the updates, and thus the costs of the plans should not be affected greatly by the order. For large deltas, our experimental results show that recomputation of the view is generally preferable to incremental maintenance, so the order of incremental propagation is not relevant.

An alternative approach for computing differentials is to generate the entire differential expression, and optimize it (see, e.g. [6]). However, the resultant expression can be very large – exponential in the size of the view expression. Optimizing such large expressions can be quite expensive, since query optimization is exponential in the size of the expression. Moreover, creating differential expressions is difficult with more complex expressions containing operations other than join (see, e.g. [6]). In contrast, the process of propagating differentials can be expressed purely in terms of how to compute the differentials for individual operations, given the differential of their inputs. As a result it is also easy to extend the technique to new operations.

4.2 The Query DAG Representation

In this section, we briefly describe the representation used in our algorithm to represent the space of recomputation and incremental maintenance plans for the given set of views.

A *Query DAG* is a directed acyclic graph whose nodes can be divided into *equivalence nodes* and *operation nodes*; the equivalence nodes have only operation nodes as children and operation nodes have only equivalence nodes as children. We first explain how the space of recomputation plans is represented as a Query DAG. This is followed by a description of how this Query DAG is refined to represent differential plans as well.

4.2.1 Query DAG Representation for Recomputation Plans

An operation node in the Query DAG corresponds to an algebraic operation, such as join ($\bowtie$), select (σ), etc. It represents the expression defined by the operation and its inputs. An equivalence node in the Query DAG represents the equivalence class of logical expressions that generate the same result set, each expression being defined by a child operation node of the equivalence node, and its inputs.

Figure 2 shows a Query DAG for the view A $\bowtie$ B $\bowtie$ C. Note that the DAG has exactly one equivalence node for every subset of $\{A, B, C\}$; the node represents all ways of computing the joins of the relations in that subset. Though the Query DAG in this example represents only a single view

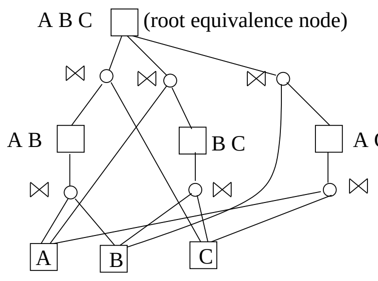


Figure 2: Query DAG for A $\bowtie$ B $\bowtie$ C. Commutativity not shown; every join node has another join node with inputs exchanged, below the same equivalence node.

A $\bowtie$ B $\bowtie$ C, in general, as indicated in Figure 1, a Query DAG can represent multiple views in a consolidated manner, with common subexpressions represented only once. Simple subsumption derivations, whereby a result such as $\sigma_{A<5}(E)$ can be computed from a result $\sigma_{A<10}(E)$, or ${}_{A}\mathcal{G}_{sum(C)}(E)$ can be generated from ${}_{A,B}\mathcal{G}_{sum(C)}(E)$, are also introduced when creating the consolidated Query DAG.

4.2.2 Incorporating Incremental Plans

Consider a database consisting of n relations: $R_1, \ldots, R_n$. Then, for each equivalence node e in the Query DAG described above, we introduce n additional equivalence nodes $\delta_e^1, \ldots, \delta_e^{2n}$, where δ_e^{2i-1} and δ_e^{2i} (for $i = 1, \ldots, n$) correspond to the differentials of e with respect to $\delta_{R_i}^+$ and $\delta_{R_i}^-$ respectively. For example, the equivalence node $e : (R_1 \bowtie R_2)$ is refined into four additional equivalence nodes $\delta_e^1 : (\delta_{R_1}^+ \bowtie R_2)$, $\delta_e^2 : (\delta_{R_1}^- \bowtie R_2)$, $\delta_e^3 : (R_1 \bowtie \delta_{R_2}^+)$ and $\delta_e^4 : (R_1 \bowtie \delta_{R_2}^-)$.

We now describe the structure of δ_e^k, $k = 1..2n$. For each child operation node o of e, there exists a child operation node o^k of δ_e^k, representing the differential of o with respect to the corresponding base relation update. In the example above, consider equivalence node e having a child operation node o which is a join operation; the children of o are the equivalents nodes representing R_1 and R_2. The node δ_e^1 has as its child an operation node o^1 which is a join operation, and the children of o^1 are the equivalence nodes for $\delta_{R_1}^+$ and R_2. The other nodes δ_e^k are similar in structure.[2] As can be seen from the above example, the children of o^k can be full results as well as differentials. The rationale of this construction was given in Section 4.1. As also mentioned in that section, the approach is easily extended to other operations.

The equivalence node e represents the full result; but this result varies as successive differentials $\delta_e^1, \ldots, \delta_e^{2n}$ are merged with it. For cost computation purposes, the system keeps an array $L[0..2n]$ with e, where $L[0]$ is the list of logical properties (such as schema and estimated statistics) of the old result and $L[i]$, for $i = 1..2n$, is the list of logical properties of the result after the result has been merged with the

[1]The differentials must be *logically* applied. The database system can give such a logical view, yet postpone physically applying the updates. By postponing physical application, multiple updates can be gathered and executed at once, reducing disk access costs.

[2]The structure is a little more complicated when a relation R is used in both children of a join node, requiring a union of several join operations. The details are straightforward and we omit them for simplicity.

differentials given by $\delta_e^1, \ldots, \delta_e^i$.

Space-Efficient Implementation. It might seem that by including all the differential expressions for each equivalence node, we have increased the size of the Query DAG by a factor of $2n$. However, our implementation reduces the cost by piggybacking the differential equivalence and operation nodes on the equivalence and operation nodes in the original Query DAG. These implementation details are explained next; however, for ease of explanation, in the rest of the paper, we stick to the above logical description.

For space efficiency, the equivalence nodes for each differential are not created separately in our implementation. Instead, each equivalence node e stores an array $D[1..2n]$, where $D[k]$ logically represents the differential equivalence node δ_e^k, and contains: (a) logical properties of the differential result δ_e^k, and (b) the best plan for computing δ_e^k.

If e does not depend on a relation R_i, or if there is no corresponding update, then the logical properties and best plan ((a) and (b) above) for $D[2i-1]$ and $D[2i]$ are set as null. In addition, as in the original representation, the equivalence node e stores the best plan for (and cost of) recomputing the entire result of the node after all updates have been made on the base relations.

Physical Properties. The Query DAG representation can be extended to incorporate *physical properties* [5], such as sort order, that do not form part of the logical data model. The extension results in a *Physical Query DAG*, in which an equivalence node in the Query Dag is refined to multiple physical equivalence nodes, one per required physical property. Our search algorithms handle physical properties, but to keep our description simple, we do not explicitly consider physical properties further.

However, it is important to note that, as suggested by [13], we model the presence of an index on a result as a physical property of the result. Our techniques thereby perform index selection as a special case of physical property selection. In fact, significant performance benefits are achieved by selecting appropriate indices for permanent materialization, especially when the number of inserts/deletes is a small percentage of the relation size.

5. MAINTENANCE COST COMPUTATION

In this section, we derive formulae for the total maintenance cost for a set $\mathcal{M}_p$ of views materialized permanently and a set $\mathcal{M}_t$ of views materialized temporarily. The optimizer basically traverses the Query DAG structure, applying these formulae, to find the overall cost.

The set $\mathcal{M}_t$ can have views corresponding to entire results (e.g. $A \bowtie B$), as well as views corresponding to differentials (e.g. $\delta_A^+ \bowtie B$). In contrast, the set $\mathcal{M}_p$ can only have views corresponding to entire results; this is because the differentials are only used during view maintenance.

The computation cost of the equivalence node e, denoted $c(e|\mathcal{M}_p, \mathcal{M}_t)$, is computed as follows, where $\mathcal{C}(e)$ is the set of children operation nodes of e.

$$c(e|\mathcal{M}_p, \mathcal{M}_t) = \begin{cases} \min_{o \in \mathcal{C}(e)} c(o|\mathcal{M}_p, \mathcal{M}_t) \\ \quad \text{if } \mathcal{C}(e) \neq \phi \\ 0 \quad \text{if } \mathcal{C}(e) = \phi \text{ (i.e. } e \text{ is a relation)} \end{cases}$$

In terms of forming the execution plan, the above equation represents the choice of the operation node with the minimum cost in order to compute the expression corresponding to the equivalence node e.

The computation cost of an operation node o, denoted $c(o|\mathcal{M}_p, \mathcal{M}_t)$, is:

$$c(o|\mathcal{M}_p, \mathcal{M}_t) = localc(o) + \sum_{e \in \mathcal{C}(o)} childc(e|\mathcal{M}_p, \mathcal{M}_t)$$

where $localc(o)$ is the "local" computation cost of the operation o, $\mathcal{C}(o)$ is the set of children equivalence nodes of o, and $childc(e|\mathcal{M}_p, \mathcal{M}_t)$ is the cost of computing the child equivalence node e, given by:

$$childc(e|\mathcal{M}_p, \mathcal{M}_t) = \begin{cases} reusec(e) \\ \quad \text{if } e \in \mathcal{M}_p \cup \mathcal{M}_t \\ c(e|\mathcal{M}_p, \mathcal{M}_t) \\ \quad \text{if } e \notin \mathcal{M}_p \cup \mathcal{M}_t \end{cases}$$

where $reusec(e)$ is the cost of reusing the result of the materialized view e.

During transient materialization, the view is computed and materialized on the disk for the duration of the maintenance processing. Thus, the cost of transiently materializing a view $e \in \mathcal{M}_t$, denoted by $trmatc(e|\mathcal{M}_p, \mathcal{M}_t)$, is:

$$trmatc(e|\mathcal{M}_p, \mathcal{M}_t) = c(e|\mathcal{M}_p, \mathcal{M}_t) + matc(e)$$

where $matc(e)$, is the cost of materializing the view (on disk, assuming materialized views do not fit in memory).

Further, for a given $e \in \mathcal{M}_p$, the cost of recomputing the result from the base relations is $c(e|\mathcal{M}_p, \mathcal{M}_t)$; and the cost of computing the differential δ_e^k, $k = 1..2n$, is $c(\delta_e^k|\mathcal{M}_p, \mathcal{M}_t)$. Let $mergec(\delta_e^k)$ denote the cost of merging the differential corresponding to δ_e^k with the view after the differentials corresponding to $\delta_e^1, \ldots, \delta_e^{k-1}$ have already been merged. Then, the cost of incrementally maintaining e, denoted by $imntc(e|\mathcal{M}_p, \mathcal{M}_t)$, is:

$$imntc(e|\mathcal{M}_p, \mathcal{M}_t) = \sum_{k=1}^{2n} (c(\delta_e^k|\mathcal{M}_p, \mathcal{M}_t) + mergec(\delta_e^k))$$

On the other hand, maintenance by recomputation involves computing the view and materializing it, replacing the old value of the view. The recomputation maintenance cost, denoted by $rmntc(e|\mathcal{M}_p, \mathcal{M}_t)$, is:

$$rmntc(e|\mathcal{M}_p, \mathcal{M}_t) = c(e|\mathcal{M}_p, \mathcal{M}_t) + matc(e)$$

where $matc(e)$, as before, is the cost of materializing the view. Notice that $rmntc(e|\mathcal{M}_p, \mathcal{M}_t)$ above is the same as $trmatc(e|\mathcal{M}_p, \mathcal{M}_t)$, the cost of transiently materializing e derived earlier. As such, we do not consider materializing a view permanently and maintaining using recomputation, unless it was already specified as permanently materialized.

For, if recomputation is the cheapest way of maintaining a view, we may as well materialize it transiently: keeping it permanently would not help the next round of view maintenance. Thus, the cost of maintaining the permanently materialized view $e \in \mathcal{M}_p$, denoted by $mntc(e|\mathcal{M}_p, \mathcal{M}_t)$, is as follows, where $\mathcal{M}$ is the set of views given as already materialized in the system.

$$mntc(e|\mathcal{M}_p, \mathcal{M}_t) = \begin{cases} \min(imntc(e|\mathcal{M}_p, \mathcal{M}_t), \\ \quad rmntc(e|\mathcal{M}_p, \mathcal{M}_t)) \\ \quad \text{if } e \in \mathcal{M} \\ imntc(e|\mathcal{M}_p, \mathcal{M}_t) \\ \quad \text{if } e \in \mathcal{M}_p - \mathcal{M} \end{cases}$$

For $e \in \mathcal{M}$, the choice corresponds to selecting the refresh mode – incremental refresh or recomputation – depending on whichever is cheaper.

Thus, the total cost incurred in maintaining the materialized views in $\mathcal{M}_p$ given that the views in $\mathcal{M}_t$ are transiently materialized, denoted $totalc(\mathcal{M}_p, \mathcal{M}_t)$, is:

$$totalc(\mathcal{M}_p, \mathcal{M}_t) = \sum_{e \in \mathcal{M}_p} mntc(e|\mathcal{M}_p, \mathcal{M}_t) + \sum_{e \in \mathcal{M}_t} trmatc(e|\mathcal{M}_p, \mathcal{M}_t)$$

Given the set $\mathcal{M}$ of views already materialized in the system, we need to determine the set $\mathcal{M}_p (\supseteq \mathcal{M})$ of views to be permanently materialized, as well as the set of views $\mathcal{M}_t$ to be transiently materialized, such that $totalc(\mathcal{M}_p, \mathcal{M}_t)$ is minimized. In the next section, we propose a heuristic greedy algorithm to determine $\mathcal{M}_p$ and $\mathcal{M}_t$.

As mentioned earlier, the optimizer performs a depth-first traversal of the Query DAG structure, applying these formulae at each node, to find the overall cost.

6. TRANSIENT/PERMANENT MATERIALIZED VIEW SELECTION

We now describe how to integrate the choice of extra materialized views with the choice of best plans for view maintenance. In Section 6.1, we present the basic algorithm for selecting the two sets of views for transient and permanent materialization respectively, followed by a discussion of some optimizations and extensions in Section 6.2.

6.1 The Basic Greedy Algorithm

Given a set of results $\mathcal{M}_p$ and $\mathcal{M}_t$ already chosen to be respectively permanently and transiently materialized, and a equivalence node x, the benefit of additionally materializing x, $benefit(x|\mathcal{M}_p, \mathcal{M}_t)$, is defined as:

$$benefit(x|\mathcal{M}_p, \mathcal{M}_t) = \begin{cases} totalc(\mathcal{M}_p, \mathcal{M}_t) - \min(totalc(\mathcal{M}_p \cup \{x\}, \mathcal{M}_t), \\ \quad totalc(\mathcal{M}_p, \mathcal{M}_t \cup \{x\})) \\ \quad \text{if } x \text{ is a full result} \\ totalc(\mathcal{M}_p, \mathcal{M}_t) - totalc(\mathcal{M}_p, \mathcal{M}_t \cup \{x\}) \\ \quad \text{if } x \text{ is a differential} \end{cases}$$

Using the expression for $totalc(\mathcal{M}_p, \mathcal{M}_t)$ derived in the previous section, along with the observations that (a) if x is a full result, then for all $e \in \mathcal{M}_p$, $mntc(e|\mathcal{M}_p, \mathcal{M}_t \cup \{x\}) = mntc(e|\mathcal{M}_p \cup \{x\}, \mathcal{M}_t)$, and also that (b) for all $e \in \mathcal{M}_t$, $trmatc(e|\mathcal{M}_p, \mathcal{M}_t \cup \{x\}) = trmatc(e|\mathcal{M}_p \cup \{x\}, \mathcal{M}_t)$, the above can be simplified to:

```
Procedure GREEDY
Input:    $\mathcal{M}$, the set of equivalence nodes
              for the initial materialized views
          $\mathcal{S}$, the set of candidate equivalence nodes
              for materialization
Output:   $\mathcal{M}_p$, the set of equivalence nodes
          to be materialized permanently
          $\mathcal{M}_t$, the set of equivalence nodes
          to be materialized transiently
Begin
    $\mathcal{M}_p$ = $\mathcal{M}$; $\mathcal{M}_t = \phi$
    while ($\mathcal{S} \neq \phi$)
L1:       Pick the node $x \in \mathcal{S}$
              with the highest $benefit(x|\mathcal{M}_p, \mathcal{M}_t)$
          if ($benefit(x|\mathcal{M}_p, \mathcal{M}_t) < 0$)
              break; /* No further benefits, stop */
          if ($x$ is a full result and
             $mntc(x|\mathcal{M}_p, \mathcal{M}_t) < trmatc(x|\mathcal{M}_p, \mathcal{M}_t)$)
              $\mathcal{M}_p = \mathcal{M}_p \cup \{x\}$
          else $\mathcal{M}_t = \mathcal{M}_t \cup \{x\}$
          $\mathcal{S} = \mathcal{S} - \{x\}$
    return ($\mathcal{M}_p, \mathcal{M}_t$)
End
```

Figure 3: The Greedy Algorithm for Selecting Views for Transient/Permanent Materialization

$$benefit(x|\mathcal{M}_p, \mathcal{M}_t) = gain(x|\mathcal{M}_p, \mathcal{M}_t) - inv(x|\mathcal{M}_p, \mathcal{M}_t)$$

where $gain(x|\mathcal{M}_p, \mathcal{M}_t)$, the gain due to additionally materializing x, is given by:

$$gain(x|\mathcal{M}_p, \mathcal{M}_t) = \sum_{e \in \mathcal{M}_p} (mntc(e|\mathcal{M}_p, \mathcal{M}_t) - mntc(e|\mathcal{M}_p, \mathcal{M}_t \cup \{x\})) + \sum_{e \in \mathcal{M}_t} (trmatc(e|\mathcal{M}_p, \mathcal{M}_t) - trmatc(e|\mathcal{M}_p, \mathcal{M}_t \cup \{x\}))$$

and $inv(x|\mathcal{M}_p, \mathcal{M}_t)$, the investment in additionally materializing x, is given by:

$$inv(x|\mathcal{M}_p, \mathcal{M}_t) = \begin{cases} \min(mntc(x|\mathcal{M}_p, \mathcal{M}_t), \\ \quad trmatc(x|\mathcal{M}_p, \mathcal{M}_t)) \\ \quad \text{if } x \text{ is a full result} \\ trmatc(x|\mathcal{M}_p, \mathcal{M}_t) \\ \quad \text{if } x \text{ is a differential} \end{cases}$$

Figure 3 outlines a greedy algorithm that iteratively picks nodes to be materialized. The procedure takes as input the set $\mathcal{S}$ of candidates (equivalence nodes, and their differentials) for materialization, and returns the sets $\mathcal{M}_p$ and $\mathcal{M}_t$ of equivalence nodes to be materialized permanently and transiently, respectively. $\mathcal{M}_p$ is initialized to $\mathcal{M}$, the set of equivalence nodes for the initial materialized views, while $\mathcal{M}_t$ is initialized as empty. At each iteration, the equivalence node $x \in \mathcal{S}$ with the maximum benefit is selected for materialization. If x is a full result, then it is added to either $\mathcal{M}_p$ or $\mathcal{M}_t$ based on whether maintaining it or transiently materializing it is cheaper; if x is a differential, then it is added to $\mathcal{M}_t$ since it cannot be permanently materialized.

Naively, the candidate set $\mathcal{S}$ can be the set of all equivalence nodes in the Query DAG (full results as well as differentials). In Section 6.2, we consider approaches to reduce the candidate set.

6.2 Optimizations

Three important optimizations to the greedy algorithm for multi-query optimization are presented in [13]. We extend these to handle differentials, as follows.

1. There are many calls to $benefit$ (and thereby to $mntc$ and $trmatc$) at line L1 of Figure 3, with different parameters. A simple option is to process each call to the above independent of other calls. However, observe that $\mathcal{M}_p$ and $\mathcal{M}_t$ change minimally in successive calls — successive calls take parameters of the form $\mathcal{M}_p \cup \{x\}$ or $\mathcal{M}_t \cup \{x\}$, where only x varies. That is, if one call considers materializing a set of the form $\mathcal{M}_p \cup \{x_1\}$ (or $\mathcal{M}_t \cup \{x_1\}$), the next call would consider materializing a different set of the form $\mathcal{M}_p \cup \{x_2\}$ (or $\mathcal{M}_t \cup \{x_2\}$). The best plans computed earlier does not change for nodes that are not ancestors of either x_1 or x_2. It makes sense for a call to leverage the work done by a previous call by recomputing best plans only for ancestors of x_1 and x_2.

   The incremental cost update algorithm presented in [13] maintains the state of the Query DAG (which includes previously computed best plans for the equivalence nodes) across calls, and may even avoid visiting many of the ancestors of x_1 and x_2. We modify the incremental cost update algorithm to handle differentials as follows.

   (a) If the full result of a node is materialized, we update not only the cost of computing the full result of each ancestor node, but also the costs for the $2n$ differentials of each ancestor node since the full result may be used in any of the $2n$ differentials. Propagation up from an ancestor node can be stopped if there is no change in cost to computing the full result or any of the differentials.

   (b) If the differential of a node with respect to a given update is materialized, we update only the differentials of its ancestors with respect to the same update. Propagation can stop on ancestors whose differentials with respect to the given update do not change in cost.

2. With the greedy algorithm as presented above, in each iteration the benefit of every candidate node that is not yet materialized is recomputed since it may have changed. The *monotonicity optimization* is based on the assumption that the benefit of a node cannot increase as other nodes are chosen to be materialized – while this is not always true, it is often true in practice. The monotonicity optimization makes the above assumption, and does not recompute the benefit of a node x if the new benefit of some node y is higher than the previously computed benefit of x. It is clearly preferable to materialize y at this stage, rather than x — assuming monotonicity holds, the benefit of x could not have increased since it was last computed, and it cannot be the node with highest benefit now, hence its benefit need not be recomputed now. Thus, recomputations of benefit are greatly reduced.

3. It is wasteful to transiently materialize nodes unless they are used multiple times during the refresh. An algorithm for computing sharability of nodes is proposed in [13], which detects equivalence nodes that can potentially be used multiple times in a single plan. We consider differential results for transient materialization only if the corresponding full result is detected to be sharable.

   The sharability optimization cannot be applied to full results in our context, since a full result may be worth materializing permanently even if it is used in only one query. Thus all full results are candidates for optimization.

Due to lack of space, we omit details of all the above optimizations.

We tried out an optimization where all differentials of an expression are considered as a single unit of materialization. That is, the greedy algorithm either chooses all differentials for materialization (based on the benefit of materializing all of them), or none. (As a post-pass of greedy, any materialized result that is not used is discarded.) One benefit of the optimization is that the number of candidates considered by the greedy algorithm decreases, leading to a reduction in optimization time. Although the optimization could possibly result in somewhat worse plans, there may also be cases where it may give a better plan. This can occur, for example, when the best plan with no differentials materialized uses recomputation, and materializing any one differential will not change the best plan (and thus have no benefit), but materializing all differentials at once may change the best plan to incremental computation and thus have a positive benefit.

6.3 Extensions

The algorithms outlined above can be extended to deal with limited space for storing materialized results by modifying the greedy algorithm to prioritize results in order of benefit per unit space (computed by dividing the benefit by the size of the result). If the space available for permanent and transient materialized results are separate, we can modify the algorithm to continue considering results for permanent (resp. transient) materialization even after the space of transient (resp. permanent) materialization is exhausted.

7. PERFORMANCE STUDY

We implemented the algorithms described earlier for finding optimal plans for view maintenance. As mentioned earlier, the implementation performs index selection along with selection of results to materialize. The implementation was performed on top of an existing query optimizer.

7.1 Performance Model

We used a benchmark consisting of views representing the results of queries based on the TPC-D schema. In particular, we separately considered the following two workloads:

- *Set of Views Workload.* A set of 10 views, 5 with aggregates and 5 without, on a total of 8 distinct relations. There is some amount of overlap across these views, but most of the views have selections that are not present in other views, limiting the amount of overlap.

- *Single Views Workload.* The same views as above, but each optimized and executed separately, and we show the sum of the view maintenance times. Since the views are optimized separately, sharing between views cannot be exploited.

The purpose of choosing a simple workload in addition to the complex workload is to show that our methods are very effective not only for big sets of overlapping complex views, where one might argue that simple multi-query optimization may be as effective, but also for singleton views without common subexpressions, where a technique based exclusively on multi-query optimization would be useless.

The performance measure is *estimated maintenance cost*. The cost model used takes into account number of seeks, amount of data read, amount of data written, and CPU time for in-memory processing. Our cost model is fairly sophisticated and, as reported in our earlier work [13], we have verified its accuracy by comparing its estimates with numbers obtained by running queries on commercial database systems. We found close agreement (within around 10 percent) on most queries, which indicates that the numbers obtained in our performance study are fairly accurate.

We provide performance numbers for different percentages of updates to the database relations; we assume that all relations are updated by the same percentage. In our notation, a 10% update to a relation consists of inserting 10% as many tuples are currently in the relation.

We assume a TPC-D database at scale factor of 0.1, that is the relations occupy a total of 100 MB. The buffer size is set at 8000 blocks, each of size 4KB, for a total of 32 MB, although we also ran some tests at a much smaller buffer size of 1000 blocks. However, the numbers are not greatly affected by the buffer size, and in fact smaller buffer sizes can be expected to benefit more from sharing of common subexpressions. The tests were run on an Ultrasparc 10, with 256 MB of memory.

7.2 Performance Results

The purpose of the experiments reported in this section is to:

1. Verify the efficacy of transient and permanent materialization of additional views (Section 7.2.1),
2. Verify the efficacy of adaptive determination of maintenance policy for each permanently materialized view (Section 7.2.2), and
3. Establish that our methods are indeed practical by showing that the overheads of our optimization-based techniques are reasonable, and that our methods scale with respect to increasing number of views (Section 7.2.3).

7.2.1 Effect of Transient and Permanent Materialization

We executed the following variations of our algorithm:

- *No Materialization.* Neither transient nor permanent materialization of additional views is allowed. That is, only the given set of initial views is permanently materialized and maintained without any sharing. This corresponds to the current state of the art.
- *Only Transient.* Transient materialization is allowed, but permanent materialization of additional views is disallowed. This corresponds to using multi-query optimization in view maintenance.
- *Transient and Permanent.* Both transient and permanent materialization of additional results is allowed. This corresponds to the techniques proposed in this paper.

In all the cases, the maintenance policy of each of the views is decided based on whether recomputation and incremental computation is cheaper, given the constraints in each case as above. All the numbers reported incorporate the optimization of treating all differentials of an expression as a single unit of materialization; we consider the effect of not using the optimization later. The results for the single view workload and the set of views workload are reported in Figure 4.

For the single-view workload, transient materialization is not useful if the view maintenance plan used is recomputation, but when incremental computation is used, full results can potentially be shared between differentials corresponding to updates of different base relations. Indeed, we found several such instances at low update percentages, although they did not have a large impact on the cost. At higher update percentages we found fewer such occurrences, and using only transient materialization did not offer much benefit. However, permanent materialization of intermediate results reduces the overall materialization cost by nearly 50% for smaller update percentages (the smallest update percentage we considered was 1%). These results clearly illustrate the efficacy of the methods proposed in this paper over and above multi-query optimization.

The set of views workload has a significant amount of overlap among the constituent views. Thus, the reduction, as high as 50%, in the overall maintenance cost due to only transient materialization is as expected. Permanent materialization has an even more significant impact in this case, and further reduces the resulting in a total reduction of up to 75% at 1% update. The gains decrease with update percentage, but remain substantial.

Recall from our discussion in Section 5 that all additional permanently materialized nodes are always maintained incrementally, since if recomputation-based maintenance of these views is cheaper than incremental maintenance, then they would be chosen for transient materialization instead of permanent materialization. Now, the cost of incremental maintenance increases with the size of the updates; for larger updates, recomputation of a permanently materialized view is a better alternative than incremental maintenance, so a smaller fraction of views are permanently materialized. These two facts together account for the slightly decreasing advantage of transient cum permanent materialization over only transient materialization as update percentages increase, as is clear from the convergence of the respective plots in Figure 4 for either workload.

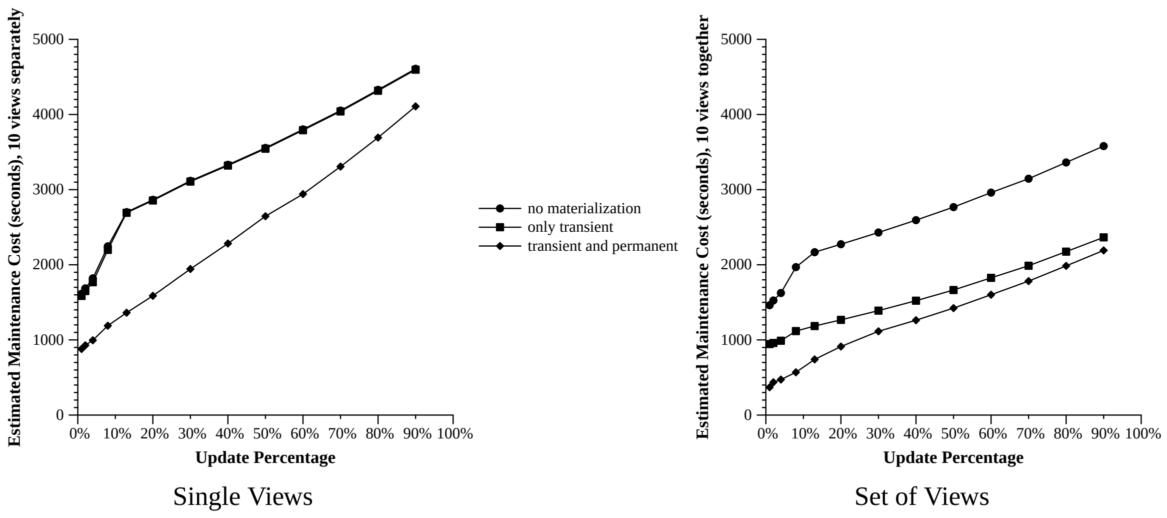


Figure 4: Effect of Transient and Permanent Materialization

Comparing across the two workloads reveals an interesting result: the cost of maintenance without selecting additional materialized view is less for the set of views than for the single view workload, even though they have the same set of queries. The reason is that in the case of set of views, the maintenance of a view can exploit the presence of *existing* materialized views, even without selecting additional materialized views. Our optimizer indeed takes such plans into consideration even when it does not select additional materialized views.

We also executed tests on a variant of our algorithm, which we call *no differential*, where materialization of differential results is turned off. Full results are still permitted to be materialized, and may be maintained either incrementally or by recomputation. For the single view benchmark, there is no sharing of differential results, so there was no difference. For the set of views benchmark, we found that there were significant benefits at low update percentages (where incremental computation is more likely to be used). For instance at 1% updates, the cost went up to 460 seconds from 370 seconds, or roughly 25%, when materialization of differential results was turned off. Even at 8% updates, there was an increase of about 15%, but by 13% the difference became small, since recomputation is used more often. (We omitted the plots for the "no differential" case from our graphs to avoid clutter.) These results clearly indicate the importance of materializing and sharing both differential and full results.

We also tested the effect of treating the differentials of an expression as separate units of materialization instead of considering them as a single unit. We found that this roughly doubled the time taken for greedy optimization, across the whole range of update percentage, yet yet made no significant difference to the plans generated. Thus the optimization of considering all differentials as a single unit has a significant benefit, at no cost, on all the examples we considered.

To summarize this section, to the best of our knowledge ours is the first study that demonstrates quantitatively the benefits of materializing extra views (transiently or permanently) to speed up view maintenance in a general setting. Earlier work on selection of materialized views, as far as we are aware, has not presented any performance results except in the limited context of data cubes or star schemas [4].

7.2.2 Effect of Adaptive Maintenance Policy Selection

In the current database systems, the user needs to specify the maintenance policy (incremental or recomputation) for a materialized view during its definition [3]. In this section, we show that an *a priori* fixed specification as above may not be the a good idea, and make a case for adaptively choosing the maintenance policy for a view in an adaptive manner.

We explored the following variants of our algorithm:

- *Forced Incremental.* All the permanent materialized views, including the views given initially as well as the views picked additionally by greedy, are forced to be maintained incrementally.
- *Forced Recomputation.* Incremental maintenance is disallowed and all the permanent materialized views are forced to be recomputed.
- *Adaptive.* The maintenance policy, incremental or recomputation, for each permanently materialized view is chosen based on the goal of minimizing the overall maintenance cost; one or the other may be chosen for a given view at different update percentages. This corresponds to the techniques proposed in the paper.

In all the cases, additional transient and materialized views were chosen by executing greedy as described earlier in the paper. The results of executing the above variants on each of our workloads are plotted in Figure 5.

The graphs show that incremental maintenance may be much more expensive than recomputation; the incremental maintenance cost increases sharply for medium to large

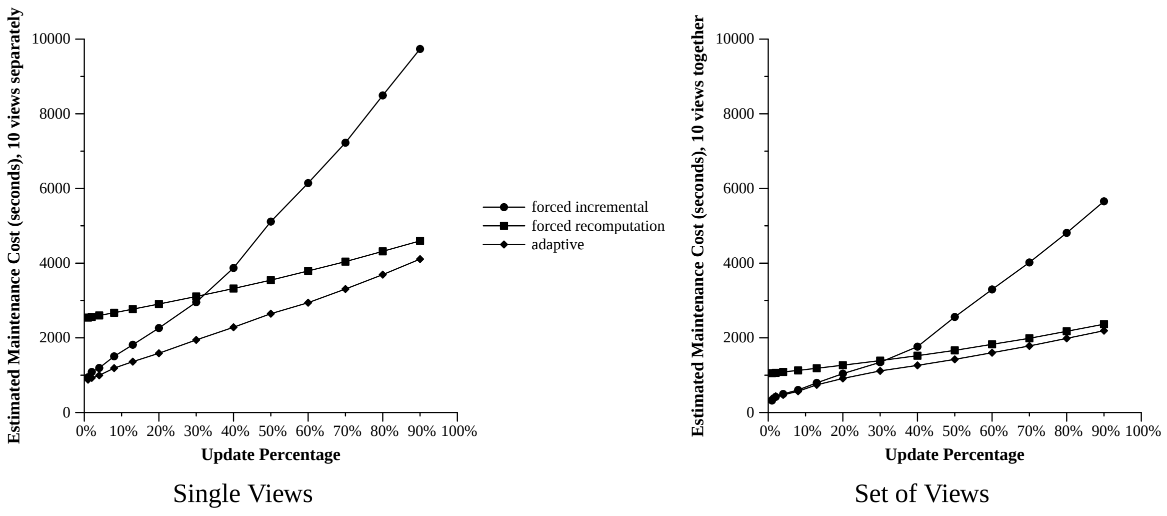


Figure 5: Effect of Adaptive Maintenance Policy Selection

update percentages – by even around 100% for very high update percentages. In both the workloads, the adaptive technique performs better than both forced incremental and forced recomputation; this extra improvement, by up to 30% for the single-view workload and 20% for the set of views workload, is due to its ability to adaptively choose incremental maintenance or recomputation on a per-view basis for the initial as well as additionally materialized views. However, the difference between adaptive and forced recomputation for either workload decreases slightly with increasing update percentage. This is because for large update percentages, incremental maintenance is expensive, and hence every view is recomputed.

These observations clearly show that blindly favoring incremental maintenance over recomputation may not be a good idea (this conclusion is similar to the findings of Vista [15]); and make a case for adaptively choosing the maintenance policy for each view, as done by our algorithms. The ability to mix different maintenance policies for different subparts of the maintenance plan, even for a single view, is novel to our techniques, and not supported by [15].

7.2.3 Overheads and Scalability Analysis

To see how well our algorithms scale up with increasing numbers of views and relations, we used the following benchmark. The benchmark uses 22 relations, R_1 to R_{22}, with an identical schema (P, SP, NUM) denoting part id, subpart id and number. Over these relations, we defined a set of 10 views V_1 to V_{10}: the view V_i was a star query on four relations R_1, R_{2i}, R_{2i+1} and R_{2i+2}, with $R_1.SP$ joined with $R_{2i}.P$, $R_{2i+1}.P$ and $R_{2i+2}.P$. We then grouped these views into 10 sets, where the k^{th} set SV_k consisted of the k views $V_1, \dots, V_k$, which together access $2k+2$ relations. For each SV_k we measured (a) the memory requirements of our algorithm and (b) the time taken by our algorithm, and report the same in Figure 6.

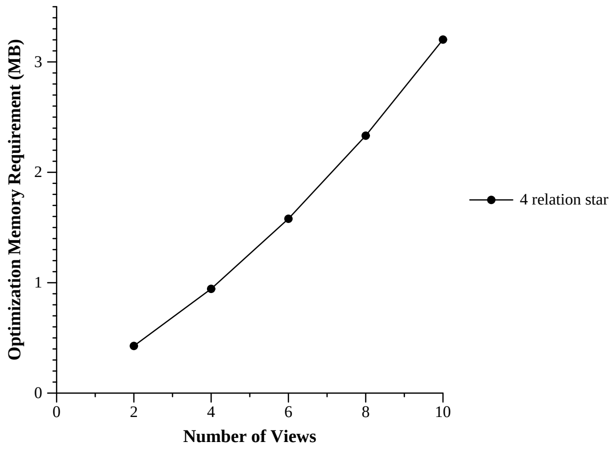
The figure shows that the memory consumption of our algorithm increases practically linearly with the number of views in the set. The reason for this is that the memory usage is basically in maintaining the Query DAG, and for our view set, the increase in the size of the Query DAG is constant per additional view added to the DAG (with a fixed number of base relations). The memory requirement for the view set SV_{10}, containing 10 views on a total of 22 relations, is only about 3.2 MB.

Further, addition of a new view from our view set to the Query DAG increases the breadth of the DAG, not its height (we think this is the expected case in reality – most views are expected to be of similar size and with only partial mutual overlap). Since the height remains constant, the time taken per incremental cost update (ref. Section 6.2) remains constant [13]. However, the number of these incremental cost updates grows quadratically with increasing number of views. This accounts for the quadratic growth in the time spent by our algorithm with increasing number of views, as shown in Figure 6. However, despite the quadratic growth, the time spent on the 22-relation 10-view set SV_{10} was less than a couple of minutes. This is very reasonable for an algorithm that needs to be executed only occasionally, and which provides savings of the order of 100's of minutes on each view refresh. For the set of 10 views in the set of views benchmark, the optimization time was around 20 seconds across the range of update percentages, which is quite acceptable.

Thus, we conclude that the memory requirements of our algorithm are reasonable and scale well with increasing number of views. The time taken shows quadratic growth, but this growth is slow enough to make the algorithm practical for reasonably large sets of views.

8. CONCLUSIONS AND FUTURE WORK

The problem of finding the best way to maintain a given set of materialized views is an important practical problem, especially in data warehouses and data marts, where the main-



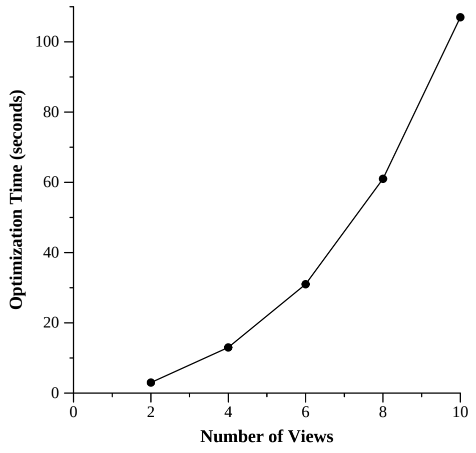


Figure 6: Scalability analysis on increasing number of views

tenance windows are shrinking. We have presented solutions that exploit commonality between different tasks in view maintenance, to minimize the cost of maintenance. Our techniques have been implemented on an existing optimizer, and we have conducted a performance study of their benefits. As shown by the results in Section 7, our techniques can generate significant speedup in view maintenance cost, and the increase in cost of optimization is acceptable.

Future work includes implementing the extensions stated in Section 6.3 to handle limited space. Another direction of extension would be to select materialized views in order to speed up a workload of queries. The greedy algorithm can be modified for this task as follows: candidates would be final/intermediate results of queries, and benefits to queries would be included when computing benefits. Longer term future work would include dealing with large sets of queries efficiently. We also plan to consider extensions of our work to a dynamic query result caching environment.

9. REFERENCES

[1] AGRAWAL, S., CHAUDHURI, S., AND NARASAYYA, V. R. Automated selection of materialized views and indexes in SQL databases. In *Intl. Conf. Very Large Databases* (2000), pp. 496–505.

[2] BLAKELEY, J. A., LARSON, P.-Å., AND TOMPA, F. W. Efficiently updating materialized views. In *ACM SIGMOD Intl. Conf. on Management of Data* (1986).

[3] BOBROWSKI, S. Using materialized views to speed up queries. *Oracle Magazine* (Sept. 1999).

[4] COLBY, L., COLE, R. L., HASLAM, E., JAZAYERI, N., JOHNSON, G., MCKENNA, W. J., SCHUMACHER, L., AND WILHITE, D. Redbrick Vista: Aggregate computation and management. In *Intl. Conf. on Data Engineering* (1998).

[5] GRAEFE, G., AND MCKENNA, W. J. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Intl. Conf. on Data Engineering* (1993).

[6] GRIFFIN, T., AND LIBKIN, L. Incremental maintenance of views with duplicates. In *ACM SIGMOD Intl. Conf. on Management of Data* (1995).

[7] GUPTA, A., AND MUMICK, I. S. Maintenance of materialized views : Problems, techniques, and applications. *IEEE Data Engineering Bulletin 18*, 2 (June 1995).

[8] GUPTA, H. Selection of views to materialize in a data warehouse. In *Intl. Conf. on Database Theory* (1997).

[9] HARINARAYAN, V., RAJARAMAN, A., AND ULLMAN, J. Implementing data cubes efficiently. In *ACM SIGMOD Intl. Conf. on Management of Data* (Montreal, Canada, June 1996).

[10] QUASS, D., GUPTA, A., MUMICK, I., AND WIDOM, J. Making views self-maintainable for data warehousing. In *Intl. Conf. on Parallel and Distributed Information Systems* (1996).

[11] ROSS, K., SRIVASTAVA, D., AND SUDARSHAN, S. Materialized view maintenance and integrity constraint checking: Trading space for time. In *ACM SIGMOD Intl. Conf. on Management of Data* (1996).

[12] ROUSSOPOLOUS, N. View indexing in relational databases. *ACM Trans. on Database Systems 7*, 2 (1982), 258–290.

[13] ROY, P., SESHADRI, S., SUDARSHAN, S., AND BHOBHE, S. Efficient and extensible algorithms for multi-query optimization. In *ACM SIGMOD Intl. Conf. on Management of Data* (2000).

[14] SELLIS, T. K. Multiple query optimization. *ACM Transactions on Database Systems 13*, 1 (1988).

[15] VISTA, D. Integration of incremental view maintenance into query optimizers. In *Intl. Conf. on Extending Database Technology (EDBT)* (1998).