ACM SIGMOD Conference, June 1996, pp.447–458

Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu

Divesh Srivastava
AT&T Research
divesh@research.att.com

S. Sudarshan
Indian Institute of Technology, Bombay
sudarsha@cse.iitb.ernet.in

Abstract

We investigate the problem of incremental maintenance of an SQL view in the face of database updates, and show that it is possible to reduce the total time cost of view maintenance by materializing (and maintaining) additional views. We formulate the problem of determining the optimal set of additional views to materialize as an optimization problem over the space of possible view sets (which includes the empty set). The optimization problem is harder than query optimization since it has to deal with multiple view sets, updates of multiple relations, and multiple ways of maintaining each view set for each updated relation.

We develop a memoing solution for the problem; the solution can be implemented using the expression DAG representation used in rule-based optimizers such as Volcano. We demonstrate that global optimization cannot, in general, be achieved by locally optimizing each materialized subview, because common subexpressions between different materialized subviews can allow nonoptimal local plans to be combined into an optimal global plan. We identify conditions on materialized subviews in the expression DAG when local optimization is possible. Finally, we suggest heuristics that can be used to efficiently determine a useful set of additional views to materialize.

Our results are particularly important for the efficient checking of assertions (complex integrity constraints) in the SQL-92 standard, since the incremental checking of such integrity constraints is known to be essentially equivalent to the view maintenance problem.

1 Introduction

The problem of incremental view maintenance has seen renewed interest in the recent past (see, e.g., [4, 8, 11, 12]). Given a materialized view defined using database relations, the problem is to compute and perform the updates to this materialized view when the underlying database relations are updated.

In this paper we show that, given a materialized SQL view V to be maintained, it is possible to reduce the time cost of view maintenance by materializing (and maintaining) *additional* views. Obviously there is also a time cost for maintaining these additional views, but their use can often significantly reduce the cost of computing the updates to V, thereby reducing the total cost. This paper addresses the following question:

> Given a materialized view V, what additional views should be materialized (and maintained) for the *optimal* incremental maintenance of V?

The SQL-92 standard permits the specification of complex integrity constraints (called *assertions*) [16]. These integrity constraints have to be checked on updates to the underlying database relations; hence it is very important that they be checked efficiently. Integrity constraints can be modeled as materialized views whose results are required to be empty. Our results are particularly important for the efficient checking of SQL-92 assertions.

Example 1.1 (Additional Materialized Views) Consider a corporate database with two relations:

- `Dept (DName, MName, Budget)`, which gives the manager and budget for each department, and
- `Emp (EName, DName, Salary)`, which gives the department and the salary of each employee.

Consider the materialized view `ProblemDept`:

```
CREATE VIEW ProblemDept (DName) AS
SELECT      Dept.DName
FROM        Emp, Dept
WHERE       Dept.DName = Emp.DName
GROUPBY     Dept.DName, Budget
HAVING      SUM(Salary) > Budget
```

*The research of Kenneth Ross was supported by a grant from the AT&T Foundation, by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by a Sloan Foundation Fellowship, by NSF grants IRI-9209029, CDA-90-24735, and by an NSF Young Investigator award.

This view determines those departments whose expense (i.e., the sum of the salaries of the employees in the department) exceeds their budget. When the database relations `Emp` and `Dept` are updated, view maintenance of `ProblemDept`, even using incremental techniques (e.g., [2, 8, 12]), can be expensive. For example, when a new employee is added to a department that is not in `ProblemDept`, or the salary of an employee in such a department is raised, the sum of the salaries of *all* the employees in that department needs to be recomputed and compared with the department's budget; this can be expensive!

The view `ProblemDept` can also be used to specify the integrity constraint "a department's expense should not exceed it's budget", by requiring that `ProblemDept` be empty. This can be specified in SQL-92 as follows:

```
CREATE ASSERTION DeptConstraint CHECK
   (NOT EXISTS (SELECT * FROM ProblemDept))
```

The efficiency of incremental view maintenance of the view `ProblemDept` (and therefore also the efficiency of checking the integrity constraint `DeptConstraint`) can be considerably improved if the view `SumOfSals` below, is additionally kept materialized.

```
CREATE VIEW SumOfSals (DName, SalSum) AS
SELECT      DName, SUM(Salary)
FROM        Emp
GROUPBY     DName
```

When new employees are added, existing employees are removed, or salaries of existing employees are modified, efficient incremental view maintenance of `SumOfSals` is possible by adding to or subtracting from the previous aggregate values. View `ProblemDept` can also be efficiently maintained by performing a natural join of the changed tuples of view `SumOfSals` with the `Dept` relation (on `DName`), and checking whether the newly computed sum of salaries exceeds the department's budget. Similarly, when a department's budget is modified, the changed tuple of the `Dept` relation can be joined with the materialized view `SumOfSals` for efficient view maintenance of `ProblemDept`. This improved efficiency of view maintenance of `ProblemDept` comes at the expense of:

- additional space to represent `SumOfSals`, and
- additional time to maintain `SumOfSals`.

When the time cost of maintaining `SumOfSals` is less than the time benefit of using `SumOfSals` for maintaining `ProblemDept`, the overall time cost of view maintenance/integrity constraint checking is reduced. We present a detailed cost model and an analysis for this example in Section 3.6. On a sample dataset, we shall show how a *greater than threefold decrease* in (estimated) materialization cost can be achieved by maintaining the additional view `SumOfSals`. Thus maintaining a suitable set of additional materialized views can lead to a substantial reduction in maintenance cost. □

1.1 Contributions and Outline

Given a materialized view V, there are several possible views that can be additionally materialized and used for the incremental maintenance of V. We show how to formulate the problem of determining what additional views to materialize as an optimization problem over the space of possible view sets. We develop an exhaustive memoing algorithm to solve the optimization problem that works under any cost model; the algorithm can be implemented using the expression DAG representation used in rule-based optimizers such as Volcano [7] (Section 3).

We present a principle of local optimality that allows problem solutions for subviews to be combined to determine the solution for the top-level view. (In general, local optimization does not ensure global optimization.) We identify conditions, based on the expression DAG representation, when this principle can be used to restrict the search space explosion (Section 4).

We suggest heuristics that can be used to prune the search space, and reduce optimization cost (Section 5). Unlike purely heuristic techniques proposed in the past, these techniques are still cost-based, but are less expensive than the exhaustive algorithm since they do not explore some parts of the full search space.

Finally, we discuss possible extensions to our techniques in Section 6.

1.2 Related Work

View maintenance (and the closely related problem of integrity constraint checking) has been studied extensively in the literature (e.g., [2, 4, 8, 11, 12, 18, 22]) for various view definition languages, e.g., Select-Project-Join (or SPJ) views, views with multiset semantics, views with grouping/aggregation, and recursive views; for various types of updates, e.g., insertions, deletions, modifications, to the database relations; and for modifications to the view definition itself. For a recent survey of the view maintenance literature, see [10].

The problem of what additional views to materialize, in order to reduce the cost of view maintenance, has been studied in the context of rule-based systems based on the RETE, TREAT and A-TREAT models [23, 13]. These models are based on discrimination networks for each rule (view); the RETE model materializes selection and join nodes in the network, while the TREAT model materializes only the selection nodes. The A-TREAT model chooses (for a fixed discrimination network) what nodes to materialize using a selectivity based heuristic, while [6] actually chooses a discrimination network and nodes to maintain, using "profitability" heuristics. Similar issues have been studied earlier in the context of maintaining integrity constraints [3, 17]. However, none of the above have explored how to choose the best set of materialized views (the best choice of discrimination network and the nodes to be maintained in it) in a truly cost-based manner; a cost-based choice is particularly important in a large database environment. To our knowledge, ours is the first paper on the topic.

The supplementary relations used in the bottom-up evaluation of recursive queries [1] can be viewed as additional materialized views that are maintained (during query evaluation) in order to efficiently maintain the view relations defining the original query. However, supplementary relations are introduced as part of a query rewriting step and do not take cost into consideration. They may be introduced when they are not required for efficient view maintenance, or not be introduced even when they are useful for efficient view maintenance.

The maintenance of a collection of simple (Select-Project) views in a distributed system is discussed in [20], where a very simple form of multi-query optimization is used to screen updates that need to be sent to remote sites. The work is extended in [19], which considers using the updates to one view to maintain other views, rather than using database relation updates; the applicability conditions presented are however very restricted.

A related, but quite different, problem is to make use of available materialized views in order to efficiently evaluate a given query, and there has been considerable work in this area (e.g., [5, 9, 14]).

2 Background

2.1 Expression Trees and DAGs

Our algorithms for determining what views to additionally materialize (and maintain) use expression trees and expression DAGs developed for performing cost-based query optimization (although the problem of query optimization is quite different from our problem). We briefly describe the expression tree and expression DAG representations here, and in Section 3 we describe in more detail how we use the trees and the DAGs.

An *expression tree* for a query/view V is a binary tree; each leaf node corresponds to a database relation that is used to define V; each non-leaf node contains an operator, and either one or two children; the algebraic expression computed by the root node is equivalent to V. (While we have called them trees here, it is possible that they could have common subexpressions, and hence be directed acyclic graphs.) Expression trees are used in query optimizers to determine the cost of a particular way of evaluating the query. Our techniques are independent of the actual set of operators; in our examples we consider operators from an extended relational algebra, that includes duplicate elimination and grouping/aggregation, in addition to the usual relational operators.

Expression DAGs are used by rule-based optimizers such as Volcano [7, 15] to compactly represent the space of equivalent expression trees as a directed acyclic graph. An expression DAG is a bipartite directed acyclic graph with "equivalence" nodes and "operation" nodes. An equivalence node has edges to one or more operation nodes. An operation node contains an operator, either one or two children that are equivalence nodes, and only one parent equivalence node. An equivalence node is labeled by the algebraic expression it computes; operation nodes correspond to various expression trees that give a result that is algebraically equivalent to the label of the parent equivalence node. The leaves of an expression DAG are equivalence nodes corresponding to database relations.

An important aspect of an expression DAG is its similarity to an AND/OR tree. An equivalence node can be "computed" by computing *one* of its operation node children. An operation node can be computed only by computing *all* of its equivalence node children.

Given an expression tree for the query, rule-based query optimizers generate an expression DAG representation of the set of equivalent expression trees and subexpressions trees by using a set of equivalence rules, starting from the given query expression tree. Details of how this step is carried out may be found in [15]; the intuition is as follows. The initial DAG is generated from the query expression tree by adding an equivalence node between each operation node and its parent, adding an equivalence node above the root of the expression tree, and replacing each relation by an equivalence node. A new expression tree is incorporated into the DAG by making each operation node of that tree a child of an existing equivalence node whose label is equivalent to the expression tree (or of a new equivalence node if there is no such existing node), and replacing each of its operands by their equivalence nodes. The cost of generation is greatly reduced when generating new expression trees using equivalence rules since the rules operate locally on the DAG representation. For details on how query optimization uses expression DAGs, see [15].

Example 2.1 Two expression trees for the view `ProblemDept` of Example 1.1 are given in Figure 1.[1] The expression DAG representation of those trees is given in Figure 2. The bold nodes **Ni** are equivalence nodes, and the remaining nodes are operation nodes. In practice, an expression DAG would represent a much larger number of trees; we have used a small example for simplicity of presentation. □

[1]One can be generated from the other by using equivalence rules such as those proposed by Yan and Larson [24].

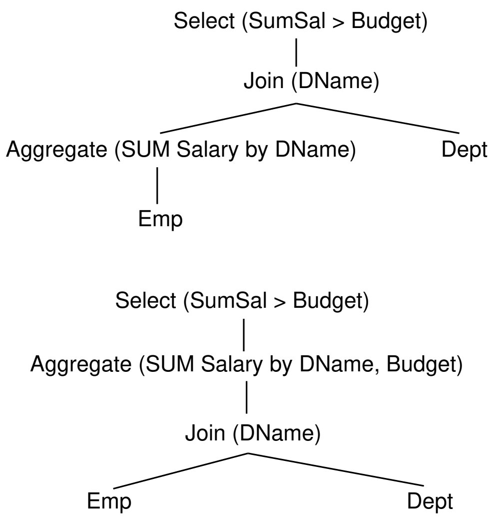

Figure 1: Two trees for the view `ProblemDept`

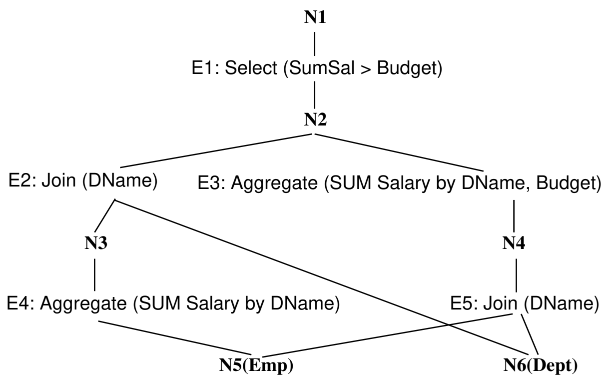

Figure 2: Expression DAG for trees of Figure 1

2.2 Incremental Updating of Expression Trees

Materialized views can be incrementally maintained when the underlying database relations are updated using the techniques of, e.g., [2, 8, 12]. The basic idea is to use differentials ΔR_i for each relation R_i that is updated, and compute the differential ΔV for a materialized view V as an expression involving the updates to the database relations (ΔR_i), the state of the database relations prior to the updates (R_i^{old}), and the state of the materialized view prior to the updates (V^{old}).

In this paper, we consider differentials that include inserted tuples, deleted tuples, and modified tuples. Our technique follows the approach of [2, 18].

To compute the Δ on the result of an operation, *queries* may have to be set up on the inputs to the operation. Consider, for example, a node N for the operation $E_1 \bowtie_\theta E_2$, and suppose an update ΔE_1 is propagated up to node N. When N is not materialized, in order to compute the update to the result of $E_1 \bowtie_\theta E_2$, a query has to be posed to E_2 asking for all tuples that match ΔE_1 on the join attributes; informally, this set of tuples can be defined as $E_2 \ltimes_\theta \Delta E_1$.

When E_2 is a database relation, or a materialized view, a lookup is sufficient; in general, the query must be evaluated. The case where both inputs to the join have been updated can be handled by a simple extension of the above scheme. Similar techniques apply for other operations, such as selection, projection, aggregation, duplicate elimination, union, intersection, difference, etc.

The exact way to generate the queries and to compute the updates can be rather subtle; see for example [8, 10, 12]. This issue will be addressed in more detail in the full version of this paper.

Consider now an expression tree. Given updates to the database relations at the leaves of the tree, the update to the result of an expression tree can be computed by starting from the updated database relations and *propagating* the updates all the way up to the root of the tree one node at a time. At each node, the update to the result of the node is computed from the updates to the children of the node. Given one or more database relations that are updated, the set of *affected* nodes in the tree are those that have an updated relation as a descendant.

At each operation node where an update is computed, there is a Δ on one or more of the inputs to the operation. We assume that the sizes of the Δ on the inputs are available. Given statistics about the inputs to an operation, we can then compute the size of the update to the result of the operation, for each of the above operations. Our techniques are independent of the exact formulae for computing the size of the Δ, although our examples use specific formulae.

3 Exhaustive Enumeration

Given a materialized view V, it may be worthwhile materializing and maintaining additional views, as illustrated in Example 1.1. In general, there are several possible views that can be additionally materialized and used for the incremental maintenance of V. For example, suppose we want to maintain the SPJ view $R_1 \bowtie R_2 \bowtie R_3$. There are several choices of sets of additional views to maintain, namely, $\{\ \}$, $\{R_1 \bowtie R_2\}$, $\{R_2 \bowtie R_3\}$, $\{R_1 \bowtie R_3\}$, $\{R_1 \bowtie R_2, R_2 \bowtie R_3\}$, $\{R_2 \bowtie R_3, R_1 \bowtie R_3\}$, $\{R_1 \bowtie R_2, R_1 \bowtie R_3\}$. Different choices may have different costs associated with them. In this section we present an exhaustive approach to the problem of determining the optimal set of additional views to materialize.

The following example illustrates some of the issues that arise when determining what views to additionally materialize.

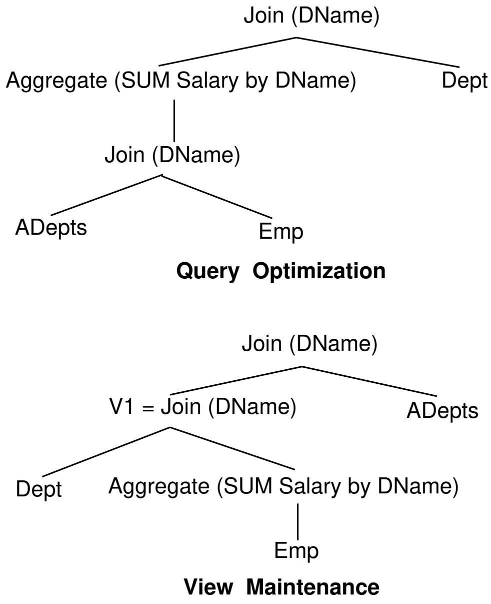

Figure 3: Query optimization versus view maintenance

Example 3.1 Consider a database with the relations `Dept` and `Emp` of Example 1.1, and an additional relation `ADepts (DName)`, which gives the departments of type `A`. Let `ADeptsStatus (DName, Budget, SumSal)` be the view defined by the following query:

```
SELECT     Dept.DName, Budget, SUM(Salary)
FROM       Emp, Dept, ADepts
WHERE      Dept.DName = Emp.DName AND
           Emp.DName = ADepts.DName
GROUPBY    Dept.DName, Budget
```

A likely plan for evaluating `ADeptsStatus`, when treated as a query, is the tree labeled "Query Optimization" in Figure 3, if the number of tuples in `ADepts` is small compared to the number of tuples in `Dept`.

When `ADeptsStatus` is a materialized view that has to be maintained under updates only to the relation `ADepts`, the cost of processing updates would be reduced significantly by materializing the view defined by `V1` in the tree labeled "View Maintenance" in Figure 3. This is because an update to `ADepts` only needs to look up the matching tuple in `V1` to incrementally maintain `ADeptsStatus`, whereas if `V1` were not maintained, a query would have to be invoked on `V1` to compute the matching tuples. Since there are no updates to the relations `Dept` and `Emp`, view `V1` does not need to be updated. In this example, {`V1`} is likely to be the optimal set of additional views to maintain.

If there were updates on the relations `Dept` and `Emp`, the cost of maintaining view `V1` would have to be balanced against the benefit of using `V1` to process updates on `ADepts`. Based on the cost of propagating updates, an optimal set of additional views to maintain has to be chosen.

Note also that the expression tree used for processing updates on a view can be quite different from the expression tree used for evaluating the view (as a query), as illustrated in this example. Hence we cannot simply use the optimal expression tree for evaluating the view in order to propagate updates. □

3.1 Space of Views

We first define the space of views that we consider for possible additional materialization.

Let D_V denote the expression DAG obtained from view V by using a given set of equivalence rules, and a rule-based optimizer such as Volcano.[2] The first step in determining the additional views to materialize for efficient incremental maintenance of V is to generate D_V, and the algorithms in the rest of the paper assume the availability of D_V.

Definition 3.1 (View Set) Given a view V, let E_V denote the set of all equivalence nodes in D_V, other than the leaf nodes. A *view set* is a subset of E_V.

The *space of possible views to materialize* is the set of all subsets of E_V that include the equivalence node corresponding to V. □

We always materialize the root node V, and the leaf nodes correspond to database relations, which are already materialized.

Consider a materialized view V that needs to be maintained, and suppose the set of materialized views we decide to maintain is $\mathcal{V}$ (where $\mathcal{V} \subseteq E_V$, and $V \in \mathcal{V}$). The views in $\mathcal{V} \setminus \{V\}$ are the additional views that are maintained in order to reduce the total cost of view maintenance; each additional view in $\mathcal{V}$ is thus a subexpression of an expression algebraically equivalent to V.

Each materialized view (including V) corresponds to a distinct equivalence node in D_V; hence these equivalence nodes can be "marked" to indicate their materialization status. The equivalence nodes corresponding to the database relations are also considered "marked".

3.2 Update and Query Models

We assume a *set of transaction types* $T_1, T_2, \ldots, T_n$ that can update the database, where each transaction type defines the relations that are updated, the kinds of updates (insertions, deletions, modifications) to the relations, and the size of the update to each of the relations.[3] We also assume that each of the transaction types T_i has an associated *weight* f_i that could reflect

[2] Our results are independent of the actual set of equivalence rules used, though a larger set of rules would obviously allow us to explore a larger search space.

[3] The size information is needed for purposes of cost estimation.

the relative frequency of the transaction type, or the relative importance of efficiently maintaining view V when that transaction is executed.

Consider a view V, a transaction type T_i, and let $\mathcal{V}$ be the set of views to be maintained. It would be inefficient to compute the updates to each view in $\mathcal{V}$ independently, since they have a lot of commonality and updates to one can be used to compute updates to others. Our approach to maintaining a set of views $\mathcal{V}$ is based on the expression DAG D_V of V; this approach takes into account the possibility of shared computation between the update computations for the different materialized views in $\mathcal{V}$.

For each transaction type T_i we propagate database relation updates up the expression DAG. We examine the issue of how to propagate the updates more closely in Section 3.3.

In order to propagate database relation updates up an expression DAG, additional queries may need to be posed for many of the operations. When updates are propagated up the nodes of the expression DAG, the inputs to an operation node are equivalence nodes; queries are thus posed on equivalence nodes. A query on an equivalence node can be evaluated using any of the operation nodes that are the children of the equivalence node; they all generate the same result, but can have different costs.

Each transaction type defines the database relations that are updated. Given a transaction type, one can go up the expression DAG, starting from the updated relations, determining the queries that need to be posed at each equivalence node. We omit the straightforward details of this process. Since each query is generated by an operation node, the query can be identified by the operation node that generates it, the child on which it is generated, and the transaction type. In what follows we assume that we have augmented the expression DAG by attaching to each operation node the set of queries needed for each possible transaction type.

Example 3.2 Consider the expression DAG of Example 2.1, shown in Figure 2. The following are the queries that may need to be posed. In each case we label the query by the number of the operation node and "L" or "R", if the node has two operands, to denote whether the query is on the left operand, or the right operand. We consider two transactions, one which modifies the `Salary` of `Emp` tuples and one which modifies the `Budget` of `Dept` tuples. We further (redundantly) label the query with "e" or "d" to denote whether relation `Emp` or `Dept` was updated (i.e., to identify the transaction that generates the query).

Q2Ld: At E2, find the sum of salaries of the department(s) that have been updated.

Q2Re: At E2, find the matching `Dept` tuple of the department whose sum of salaries has changed.

Q3e, Q3d: At E3, find the sum of salaries of the department(s) of the updated join tuple(s).

Q4e: At E4, find the sum of salaries of the department(s) from which the updated `Emp` tuple(s) came.

Q5Ld: At E5, find the employees of the updated `Dept` tuple(s).

Q5Re: At E5, find the matching `Dept` tuple of the updated `Emp` tuple(s). ☐

3.3 Relevant Parts of the Expression DAG

Consider a materialized view V and a given set $\mathcal{T}$ of transaction types $T_1, T_2, \ldots, T_n$. We must be able to determine the cost of maintaining a given set of views $\mathcal{V}$ for a given transaction type T_i.

To maintain $\mathcal{V}$, updates must be propagated up the expression DAG from updated database relations to every materialized view in $\mathcal{V}$. However, it is not necessary to propagate an update along every path up the DAG, since each operation node below an equivalence node will generate the same result, and only one need be chosen to propagate the update. *Update tracks*, introduced below, make precise the different minimal ways of propagating updates up an expression DAG to maintain a set of materialized views $\mathcal{V}$, given a transaction type T_i.

First, we introduce the notion of a subdag of an expression DAG, which identifies the different ways of propagating updates up the expression DAG to the set of views $\mathcal{V}$, independent of the specific database relations updated by the transaction type.

Definition 3.2 (Subdags) Given an expression DAG D_V and a set $\mathcal{V}$ of equivalence nodes in D_V, a *subdag* of D_V containing $\mathcal{V}$ is any subset of the DAG D_V satisfying the following properties:

- each equivalence node in $\mathcal{V}$ is in the subdag.
- for each non-leaf equivalence node in the subdag, exactly one of its child operation nodes is in the subdag.
- for every operation node in the subdag, each of its child equivalence nodes is in the subdag.
- no other nodes are in the subdag.
- edges in D_V connecting the nodes in the subdag are edges in the subdag.

$SubDags(D_V, \mathcal{V})$ denotes the set of all subdags of D_V containing $\mathcal{V}$. ☐

The intuition behind subdags is that it suffices for each equivalence node to compute its update using one of its child operation nodes; computing updates using other child operation nodes at the same time would be redundant. So would computing updates to nonmaterialized equivalence nodes used only in operation nodes that are themselves not used.

The notion of subdags is a generalization of the notion of an expression tree, to handle multiple materialized views. If $\mathcal{V}$ contains only the view V, a subdag is merely any expression tree represented by the DAG, rooted at the equivalence node corresponding to the view V. Just as the set of all expression trees represented by an expression DAG for a *single query* is the set of *all ways* to compute the query, the set of all subdags of a DAG defines the set of *all ways* of computing a *set of queries* (or materialized views, in our case). Moreover, in our model, each way of computing the materialized views corresponds to a way of propagating updates to the materialized views. Hence the set of subdags $SubDags(D_V, \mathcal{V})$ also defines the set of all ways of *propagating updates* to the materialized views.

If the expression DAG D_V of V has nodes that do not have any descendant nodes updated by transactions of a given type, updates need not be propagated up to these nodes.

Definition 3.3 (Update Track) Consider a marked expression DAG D_V for view V with the set of marked nodes being $\mathcal{V}$, and a transaction of type T_i. Let U_V denote the subset of equivalence/operation nodes of D_V whose results are affected by transactions of type T_i. Given any subdag S_D of D_V including $\mathcal{V}$, the subset of S_D consisting of affected nodes and the edges between them in S_D is an *update track* of D_V for transactions of type T_i. □

Given an update track S_D as above, and a set of updates to database relations by a transaction of type T_i, the updates can be propagated up the nodes of the update track. At each node, the update to the result is computed based on the updates to its inputs using the incremental techniques described in, e.g., [2, 18].

3.4 Cost of Maintaining a Set of Views

Consider the time cost of maintaining a set of views $\mathcal{V}$ for a given transaction type T_i. There are multiple update tracks along which updates by transactions of type T_i can be propagated; these could have different costs. We now discuss the issue of computing the cost of propagating updates along a single update track. This cost can be divided into two costs: (a) computing the updates to the various nodes in the update track, and (b) performing the updates to the views in $\mathcal{V}$.

Cost of Computing Updates: The computation of updates to nodes in an update track poses queries that can make use of the other materialized views in $\mathcal{V}$ (which is the reason for maintaining additional views). Determining the cost of computing updates to a node in an update track in the presence of materialized views in $\mathcal{V}$ thus reduces to the problem of determining the cost of evaluating a query Q on an equivalence node in D_V, in the presence of the materialized views in $\mathcal{V}$. This is a standard query optimization problem, and the optimization techniques of Chaudhuri et al. [5], for example, can be easily adapted for this task.

When propagating updates along an update track, many queries may need to be posed. This set of queries can have common subexpressions, and multi-query optimization techniques (see, e.g., [21]) can be used for optimizing the evaluation of the collection of queries. Shared work between the queries could lead to locally nonoptimal plans being globally optimal. Note that the presence of common subexpressions in the expression DAG influence a solution to the problem of determining what additional views to materialize in two distinct ways:

- First, subexpressions can be shared between different views along a path for propagating updates. The notions of subdags and update tracks were used to deal with such subexpressions.
- Second, subexpressions can be shared between different queries generated along a single update track. Multi-query optimization is used to deal with such subexpressions.

Our technique and results are applicable for any cost model for evaluating the queries. Hence, we omit details of the exact way of determining the cheapest way of evaluating the set of queries generated, and computing their costs. The plan chosen for computing the updates will of course depend on the cost model used. In our examples, we describe and use a specific cost model for estimating the costs of the queries.

Cost of Performing Updates to $\mathcal{V}$: The cost of materializing a view also includes the cost of performing updates to the materialized view. The cost of performing updates depends on the physical storage model, including the availability of suitable indices (which must themselves be updated too). The cost also depends on the size of the incoming set of updates, that is, the size of the "delta" relations. The size can be estimated from the transaction type, and the definition of the deltas; the actual size is independent of the way the delta relations are computed.

For example, at node **N4** in Example 2.1 we might expect one update tuple for an update to the `Emp` relation, but 10 update tuples for an update to the Dept relation if the average department contains 10 employees. Another example is node **N3**, where the cost of materialization is zero for updates to the Dept relation. There are a number of reasonable cost models for calculating the cost of performing updates; our techniques apply no matter which is chosen.

Space Cost: In addition to the above costs, there is clearly a space cost associated with maintaining additional materialized views. There is no unique way of combining space and time costs to get a single cost; economic models based on assigning "monetary" costs to the space and time costs can be used to generate a combined cost for a view set. Although our techniques can work with such a combined cost, for simplicity we do not explicitly refer to the space cost in the rest of the paper.

3.5 Exhaustive Algorithm

Algorithm OptimalViewSet, in Figure 4, is an exhaustive algorithm for determining the optimal set of additional views to materialize for incremental maintenance of V.

The cost of maintaining a view set $\mathcal{V}$ for transaction type T_i is obtained as the cost of the *cheapest* update track that can be used for propagating updates to $\mathcal{V}$; let $C(\mathcal{V}, T_i)$ denote this cost. By weighting the cost $C(\mathcal{V}, T_i)$ with the weight f_i of transaction type T_i, the weighted average cost of maintaining materialized views $\mathcal{V}$ can be computed as:

$$C(\mathcal{V}) = \frac{\Sigma_i C(\mathcal{V}, T_i) * f_i}{\Sigma_i f_i}$$

The optimal view set $\mathcal{V}^{opt}$ to be materialized for maintaining V can be chosen by enumerating all possible markings of equivalence nodes of D_V, and choosing one that minimizes the weighted average cost. (Recall that the root equivalence node of D_V is always marked.)

Theorem 3.1 *Given a view V and an expression DAG D_V for V, the view set $\mathcal{V}^{opt}$ selected by Algorithm* OptimalViewSet *has the* lowest *estimated maintenance cost among all view sets that are subsets of E_V and contain V.* □

The optimality of OptimalViewSet naturally depends on the assumed optimality of the underlying multi-query optimization subroutine. The complexity of OptimalViewSet depends on the set of equivalence rules used, and could be doubly exponential (or more) in the number of relations participating in the view definition; clearly the algorithm is not cheap. However, the cost can be acceptable if the view uses a small number of relations, or if the optimizations and heuristics discussed later are used.

The need for update tracks that are not trees in Algorithm OptimalViewSet results in cost calculations that are inherently nonlocal. We shall return to this issue in Section 4.

3.6 Motivating Example Revisited

Let us consider Example 1.1 once more. The expression trees and expression DAG are given in Figures 1 and 2. The subqueries generated are given in Example 3.2. For the sake of clarity, we shall use a simplified cost model described below. In practice, more realistic cost models would be used. We assume all indices are hash indices, that there are no overflowed hash buckets, and that there is no clustering of the tuples in the relation.

We count the number of page I/O operations. Looking up a materialized relation using an index involves reading one index page and as many relation pages as the number of tuples returned. Updating a materialized relation involves reading and writing (when required) one index page per index maintained on the materialized relation, one relation page read per tuple to read the old value, and one relation page write per tuple to write the new value.

Let us suppose that we have 1000 departments, 10000 employees, and a uniform distribution of employees to departments. Further, let us assume that none of the data is memory-resident initially. We have two types of transactions: ▷Emp that modifies the salary of a single employee, and ▷Dept that modifies the budget of a single department. We shall consider four possible additional view sets to materialize: (a) ∅, (b) {**N2**}, (c) {**N3**}, and (d) {**N4**}. (The exhaustive algorithm would consider all possible view sets.)

The total incremental maintenance costs of various materializations are summarized in the following table, assuming that each of the materializations has a single index on DName. (We do not count the cost of updating the database relations, or the top-level view ProblemDept.) For example, the cost of incrementally maintaining **N4** when a Dept tuple is modified involves reading, modifying and writing 10 tuples; this has a cost of 21 page I/Os (including one index page read; no index page write is necessary). Similarly, the cost of incrementally maintaining **N3** when an Emp tuple is modified involves reading, modifying and writing 1 tuple; this has a cost of 3 page I/Os (again, no index page write is necessary). The cost of maintaining **N4** in response to an update of an Emp tuple is 12, representing reading an index page for the given department, reading the 10 tuples of **N4** with that department, and writing back the modified tuple corresponding to the changed employee.

	∅	{**N2**}	{**N3**}	{**N4**}
▷Emp	0	3	3	12
▷Dept	0	3	0	21

```
Algorithm OptimalViewSet (V) {
  /* compute cost of performing updates */
  for each equivalence node N of D_V
    for each transaction type T_j
      calculate the update cost for N and store in M[N, j];

  C^opt = ∞;
  /* compute weighted total cost for each view set, and compare with best so far */
  for each possible view set 𝒱 {
    for each transaction type T_j {
      compute the total update cost m_j for all members of 𝒱;   /* m_j computation uses M[*, j] */
      find the update track from the marked nodes to the leaves with minimum total accumulated
        query cost q_j along the update track;   /* q_j computation utilizes multi-query optimization */
      associate with 𝒱 the cost q_j + m_j for updates to transactions of type T_j;
    }
    calculate the weighted cost c for 𝒱 according to the weights f_j for each T_j;
    if (c < C^opt) then { 𝒱^opt = 𝒱; C^opt = c; }
  }
}
```

Figure 4: Exhaustive algorithm for determining optimal view set

We consider the following four update tracks.[4]

Track	Transaction	Queries
N1,E1,**N2**,E2,**N3**,E4,**N5**	▷Emp	**Q4e**, **Q2Re**
N1,E1,**N2**,E3,**N4**,E5,**N5**	▷Emp	**Q5Re**, **Q3e**
N1,E1,**N2**,E2,**N6**	▷Dept	**Q2Ld**
N1,E1,**N2**,E3,**N4**,E5,**N6**	▷Dept	**Q5Ld**, **Q3d**

On each update track several queries are posed, as indicated in the table. We must perform multi-query optimization on each collection of queries. Further, the optimization should take advantage of the fact that certain index and main relation pages from the materialized views are available *for free* (given sufficient buffer space) since those pages are needed for view maintenance anyway. The following table gives the total query costs for each update track.

	$\emptyset$	{**N2**}	{**N3**}	{**N4**}
N1,E1,**N2**,E2,**N3**,E4,**N5**	13	0	2	0
N1,E1,**N2**,E3,**N4**,E5,**N5**	13	0	2	0
N1,E1,**N2**,E2,**N6**	11	0	2	0
N1,E1,**N2**,E3,**N4**,E5,**N6**	11	11	11	0

Let us examine more closely how some of these numbers were derived. Query **Q3d** can be evaluated very efficiently on the update track **N1**,E1,**N2**,E3,**N4**,E5,**N6**: Since Dname is a key for the Dept relation, the Δ result propagated up along E5 and **N4** contains *all* the tuples in the group. Thus no I/O is generated for **Q3d**. (The conditions under which keys can be used to reduce the set of needed queries will be presented in the full version of this paper.) On this same track, **Q5Ld** can be answered by looking up the Emp relation: since each department has an average of 10 employees, an indexed read of the Emp relation has a cost of 11 page I/Os (including one index page read). However, in the case that **N4** is materialized, we can reuse the pages of **N4** read for view maintenance (contributing to the maintenance cost of 21 above), leading to no additional query cost.

The update track **N1**,E1,**N2**,E2,**N3**,E4,**N5** involves queries **Q4e** and **Q2Re**. Both of these queries can be answered with no additional cost when either **N2** or **N4** is materialized, using the pages previously counted for maintenance. When **N3** is materialized, **Q4e** can be answered without additional cost, but **Q2Re** needs two page reads to get the required tuple from Dept. When nothing is materialized, we need 11 pages for **Q4e** and 2 for **Q2Re**, a total of 13.

Combining the materialization and query costs, and minimizing the costs for each updated relation we get the following table. For example, the cheapest way of maintaining the view set {**N3**} for transaction type ▷Emp involves using the update track **N1**,E1,**N2**,E2,**N3**,E4,**N5** (for a cost of 2 page I/Os); the cost of maintaining **N3** itself for transaction type ▷Emp is 3 page I/Os; the total is 5 page I/Os.

	$\emptyset$	{**N2**}	{**N3**}	{**N4**}
▷Emp	13	3	5	12
▷Dept	11	3	2	21

Independent of the weighting for each transaction type, materializing {**N2**} or {**N3**} wins over not maintaining any additional views, as well as over maintaining the view set {**N4**}; {**N3**} corresponds to maintaining the view SumOfSals from Example 1.1. Maintaining {**N4**} is usually (depending on the transaction weights)

[4] There are additional update tracks that are not simple paths, such as **N1**,E1,**N2**,E3,**N4**,E5,**N5** $\cup$ **N3**,E4,**N5** when **N3** is materialized.

worse than not materializing any additional views; by making a wrong choice of additional views to materialize the cost of view maintenance may be worse than materializing no additional views. The choice between materializing $\{\mathbf{N2}\}$ and $\{\mathbf{N3}\}$ can be made *quantitatively* on the basis of the transaction weights. If we assume that the top-level view changes rarely, because the integrity constraint is rarely violated, then by materializing $\{\mathbf{N2}\}$ we use an average of 3 page I/Os per transaction for maintenance compared with 12 when materializing no additional views, assuming an equal weight for the two transactions. *That's a reduction to 25% of the cost incurred when no additional views are maintained.*

4 Impact of Common Subexpressions

A close examination of Algorithm OptimalViewSet reveals that it computes the minimum cost query plan for an operation node once for every update track being considered. It would certainly be more efficient to compute the minimum cost plans only once for a given set of marked nodes, and not repeat the computation for each update track. Unfortunately, such a computation would not be correct in that it may ignore a globally optimal plan for propagating updates that is composed of plans that are not locally optimal. Locally suboptimal plans may be part of a globally optimal plan due to the costs of common subexpressions being amortized by multiple uses.

Common subexpressions between two views in the view set $\mathcal{V}$ can arise because the view V itself had common subexpressions. Even when V itself does not have common subexpressions, it is possible for the expression DAG D_V to have common subexpressions. In estimating the cost of maintaining an arbitrary view set (i.e., a set of equivalence nodes) $\mathcal{V}$, a similar argument shows that suboptimal plans for maintaining individual views in $\mathcal{V}$ can be combined to obtain an optimal plan for maintaining $\mathcal{V}$.

As a consequence, it appears that there is inherently little scope for reuse of lower-level cost computations in deriving globally optimal costs. We next look at a special case where there is a higher degree of locality.

Definition 4.1 (Articulation Node) An *articulation node* of a connected undirected graph is a node whose removal disconnects the graph. □

When the expression DAG D_V (viewed as an undirected graph) of a view V has equivalence nodes that are articulation nodes, it turns out that only optimal cost expression trees for the materialized subviews corresponding to articulation nodes are used in determining the optimal choice for V.

The intuition is as follows: Consider an equivalence node N that is an articulation node of D_V; let $N1$ denote any descendant equivalence node of N; and $N2$ denote any ancestor equivalence node of $N1$ that is not also a descendant of N. Then $N1$ can be part of an expression tree with root $N2$ only if N is also part of that expression tree. Consequently, the subview corresponding to node N can be optimized locally. This intuitive notion is formalized in the following principle of local optimality. Recall that E_V denotes the set of all subviews of view V. We let $Opt(V)$ denote the optimal subset of E_V chosen for maintenance of V.

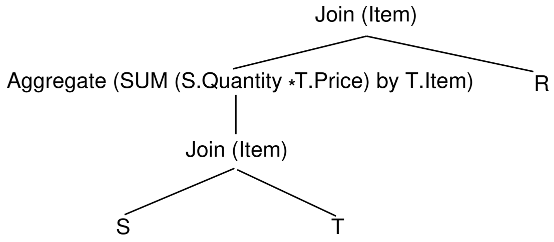

Figure 5: Articulation node

Theorem 4.1 (Local Optimality Principle) : If $V1 \in Opt(V)$, and the equivalence node corresponding to $V1$ is an articulation node of D_V, then $Opt(V1) = Opt(V) \cap E_{V1}$. □

A similar argument establishes that query costs can be minimized at articulation nodes.

Given a materialized view V, the algorithm OptimalViewSet presented previously operated on the expression DAG D_V of view V, and computed update costs and query costs separately for each update track in D_V. When D_V has equivalence nodes that are articulation nodes, the Local Optimality Principle can be used to considerably optimize this algorithm. For each equivalence node N in D_V that is an articulation node, let D_N denote the subset of D_V consisting of N, its descendant nodes, and edges between them. Then, for each D_N, update costs and query costs can be computed for update tracks *independently* of (update and query) cost computations for other D_N's. These costs can be directly combined to compute the update and query costs for update tracks in D_V. This can considerably restrict the search space that needs to be explored to obtain an optimal solution!

Articulation nodes often arise in expression DAGs of complex views, especially when the view is defined in terms of other views with grouping/aggregation. Consider, for example, the expression tree in Figure 5. The aggregation cannot be pushed down the expression tree because it needs both `S.Quantity` and `T.Price`. If `Item` is not a key for relation `R`, then the aggregation cannot be pushed up the expression tree because the multiplicities would change. Hence, the equivalence node that is the parent of the grouping/aggregation node in the expression DAG is a natural articulation point.

5 Heuristic Pruning of the Search Space

The exhaustive approach, even with the optimizations described above, can be expensive for complex SQL views since the search space is inherently large. By the very nature of the problem, optimization does not have to be performed very often, and hence the exhaustive approach may be feasible even for complex views. If, however, the view is too complex for an exhaustive search strategy to be feasible, or the optimization time is required to be small, heuristics can be used to drastically reduce the search space. The heuristics described below reduce, but (intentionally) do not eliminate the search space entirely, so that at least several different view sets are considered and the best one chosen.

Using a Single Expression Tree: Using a *single* expression tree equivalent to V for determining a view set can dramatically reduce the search space: the number of equivalence nodes that need to be considered is smaller than in the expression DAG D_V, and each equivalence node has only one child operation node whose costs need to be computed. One possibility is to simply use the expression tree that has the lowest cost for evaluating V when *treated as a query*, ignoring the potential suboptimality outlined in Example 3.1.

Choosing a Single View Set: Given an expression tree, a marking of its nodes can be chosen heuristically. A simple heuristic is to mark each equivalence node of the tree that is the (unique) parent of operations that are expensive to compute incrementally if their result is not materialized. Examples of such operations are grouping/aggregation and duplicate elimination. Finally, the set of marked nodes can be chosen as the additional views, provided that the cost of this option is cheaper than the cost of not materializing any additional views. This approach is similar to the rationale used in TREAT, but unlike TREAT, the set of all expressions trees can still be explored to find the best.

Approximate Costing: An alternative to using the heuristics described above is based on *approximate costing*. One possibility is to use a greedy approach to costing, and only maintain a single update cost with each equivalence node, and a single cost with each query at an equivalence node during the exhaustive procedure, even when the equivalence node is not an articulation node of expression DAG D_V (see Section 4). The result of the greedy approach of associating a single cost with each query, on Algorithm OptimalViewSet, is to move the query cost computation out of the inner-most loop, which reduces the complexity of the algorithm.

6 Discussion

The problem of efficient view maintenance is especially important for implementing SQL-92 assertions. These assertions are simply queries that return true/false, and the database is required to be such that the assertions are always true. When the database is updated, it is important to check if the update will make the assertion false. An assertion can be modeled as a materialized view, and the problem then becomes one of computing the incremental update to the materialized view. While the materialized view corresponding to the root of the assertion expression must always be empty, the remaining nodes can correspond to non-empty views. Standard integrity constraint checking techniques can be used for simple assertions. However, if assertions are complex, incrementally checking them may be quite costly unless additional views are materialized, as illustrated in the motivating example (Example 1.1).

Our results can be applied in a straightforward fashion to the problem of determining what views to additionally materialize for efficiently maintaining a *set* of materialized views. The key to this is the fact that the expression DAG representation can also be used to compactly represent the expression trees for a set of queries, not just a single query. The only change will be that the expression DAG will have to include multiple view definitions, and may therefore have multiple roots, and every view that must be materialized will be marked in the expression DAG. Other details of our algorithms remain unchanged.

Materializing additional views has benefits beyond those addressed here. In particular, additional views could be used to speed up ad-hoc queries. Algorithm OptimalViewSet can be extended to take query benefits into account by performing an additional optimization step (given a set of queries with weights) for each possible view set, and incorporating this query cost into our total cost.

7 Conclusions and Future Work

We have examined the problem of reducing the cost of maintaining a materialized view by materializing (and maintaining) additional views, and presented an exhaustive algorithm for finding the best set of views to materialize. We have presented optimizations for the algorithm, as well as some heuristics. Our techniques are important for maintaining complex SQL views and for checking complex SQL-92 assertions. Our techniques are *cost-based*, but independent of the cost model used; any cost model can be "plugged in." We showed that the problem is inherently nonlocal and that the cost must be globally optimized. Our work should stand as a foundation for future work in the area.

The view sets chosen using our (non-heuristic) techniques are optimal under the update propagation model we have used. The model is a complete and powerful one, but there are other models for propagating updates based on defining update expressions, such as the one described in [8]. It would be interesting to consider how to find the optimal way to maintain a view (or set of views) under such a model.

Another direction for future work is to use a more general abstraction of database relation updates than the insert/delete/modify abstraction used in this paper. Abstracting other types of updates, such as increment/decrement, and recognizing the type of a given update, can lead to better view maintenance techniques. The issue of space cost also needs further study.

Acknowledgements

We thank Zoltan Somogyi for pointing out how our techniques could be extended to incorporate user-query weights. We also thank the anonymous referees for their many insightful comments.

References

[1] C. Beeri and R. Ramakrishnan. On the power of Magic. *Journal of Logic Programming*, 10(3&4):255–300, 1991.

[2] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 61–71, Washington D.C., May 1986.

[3] B. Blaustein. *Enforcing database assertions: Techniques and applications.* PhD thesis, Harvard University, 1981.

[4] S. Ceri and J. Widom. Production rules for incremental view maintenance. In *Proceedings of the International Conference on Very Large Databases*, Barcelona, Spain, 1991.

[5] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the IEEE International Conference on Data Engineering*, 1995.

[6] F. Fabret, M. Regnier, and E. Simon. An adaptive algorithm for incremental evaluation of production rules in databases. In *Proceedings of the International Conference on Very Large Databases*, Aug. 1993.

[7] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the IEEE International Conference on Data Engineering*, Vienna, Austria, 1993.

[8] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1995.

[9] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of the International Conference on Very Large Databases*, 1995.

[10] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Data Engineering Bulletin*, 18(2), June 1995. Special Issue on Materialized Views and Data Warehousing.

[11] A. Gupta, I. S. Mumick, and K. Ross. Adapting materialized views after redefinitions. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, San Jose, CA, May 1995.

[12] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 157–166, 1993.

[13] E. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, June 1992.

[14] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the ACM Symposium on Principles of Database Systems*, San Jose, CA, May 1995.

[15] W. J. McKenna. *Efficient search in extensible database query optimization: The Volcano optimizer generator.* PhD thesis, University of Colorado, 1993.

[16] J. Melton and A. R. Simon. *Understanding the new SQL: A complete guide.* Morgan Kaufmann, San Francisco, CA, 1993.

[17] R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In *Advances in Database Theory.* Plenum, 1984.

[18] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[19] A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 512–520, 1990.

[20] A. Segev and J. Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.

[21] T. Sellis. Multiple query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, Mar. 1988.

[22] L. Vieille, P. Bayer, and V. Küchenhoff. Integrity checking and materialized views handling by update propagation in the EKS-V1 system. Technical report, CERMICS - Ecole Nationale Des Ponts et Chaussees, France, June 1991. Rapport de Recherche, CERMICS 91.1.

[23] Y.-W. Wang and E. Hanson. A performance comparison of the Rete and TREAT algorithms for testing database rule conditions. In *Proceedings of the IEEE International Conference on Data Engineering*, 1992.

[24] W. P. Yan and P.-A. Larson. Performing group-by before join. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 89–100, 1994.