Speeding Up Materialized-View Maintenance Using Cheap Filters at the Warehouse

Nam Huyn
Surromed, Inc.
phuyn@surromed.com

Abstract

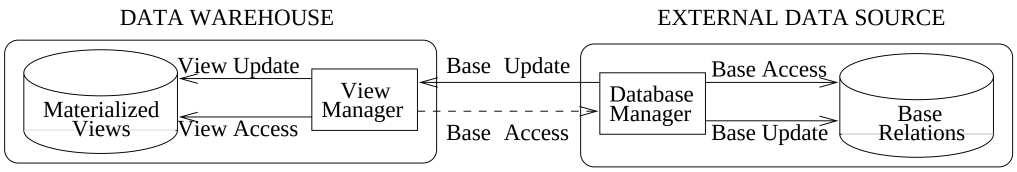
We consider the problem of speeding up the incremental maintenance of materialized views defined by conjunctive queries (CQ) over external base relations, when querying these base relations is *expensive*. Our approach consists of detecting, *without* using the base relations, situations where a view either is not affected by a base update (VDU) or can be maintained using only the views in the warehouse (VSM). We are doing runtime optimization of view maintenance, since the tests take the current state of the warehouse database into account. Testing VSM for CQ views in general is known to be co-NP-complete in the size of the views, and testing VDU is conjectured to be as hard. In this paper, we identify important subclasses of CQ views for which VDU and VSM can be tested *completely* and *efficiently*, using only a small constant number of view lookups. This result is significant because, by maintaining indexes on selected view attributes, we can speed up view maintenance practically without incurring any substantial overhead. For more general CQ views, we show sufficient tests for VDU and VSM that have comparable efficiency. To demonstrate the performance of our method, we implement a view manager in Oracle PL-SQL which maintains a simple view under various synthetic update streams and under various delays in querying the base relations. Our results clearly show situations where tremendous speedup can be achieved. While further performance studies remain to be done, we believe our approach has a great potential to significantly speed up incremental data warehouse maintenance.

1 Introduction

1.1 Motivation

A data warehouse is essentially a read-mostly database that provides views of integrated data to its users. However, it differs from a traditional database in many ways. First, the base data from which the views derive typically resides in data sources external to the warehouse. Thus, from the standpoint of the warehouse, querying these sources can be significantly more expensive than querying the warehouse. Second, the views provided by the warehouse are typically materialized, that is, precomputed and stored at the warehouse. Consequently, the views need to be refreshed periodically to reflect the changes made to the data sources.

In this paper, we assume that when a source changes, the base update is sent to the data warehouse, which has the burden of maintaining its views. The following depicts the relationship between a warehouse and a source, in which the view updates are evaluated at the warehouse, as opposed to the source from which the base update originates.



As data warehouses continue to grow in size, incremental techniques for maintaining them efficiently are becoming increasingly important. Incremental view maintenance has been well studied in the literature [BC79, QW91, CW91, SJ96, GLT97], but these work typically assume that the base relations can be accessed cheaply. For instance, to incrementally maintain a view that joins several external base relations when one of these relations is updated, the remaining nonupdated base relations are examined in order to determine the required view update. However, this process can be expensive in a data warehouse environment since it involves (1) querying the data sources which may be remotely located (2) computing a distributed join of the base relations which may reside across multiple sites.

Thus, we are led to wonder if we can speed up the maintenance process by saving on these external base accesses. A key observation is that since redundancy is often inherent in the views, we may be able to exploit this redundancy to maintain the views themselves. In many situations, a view is not affected by a base update and thus requires no update. In other situations, the view needs to be updated but the required view update can be computed using only information local to the warehouse, for instance, the contents of the view itself. A view in the latter situations is said to be *self-maintainable* ([TB88, Huy97a]). Thus, if we can detect these situations, based on the current state of the views but without using the base relations, and do so efficiently, we will have the potential to significantly speed up the maintenance process.

Example 1.1 Consider a view V defined by the following conjunctive query (aka select-project-join query with equality comparisons) in rule notation

$$v(X,Y,Z,U) :- r(X,Y,Z,T) \ \& \ p(X,W,U) \ \& \ q(U,Y,W)$$

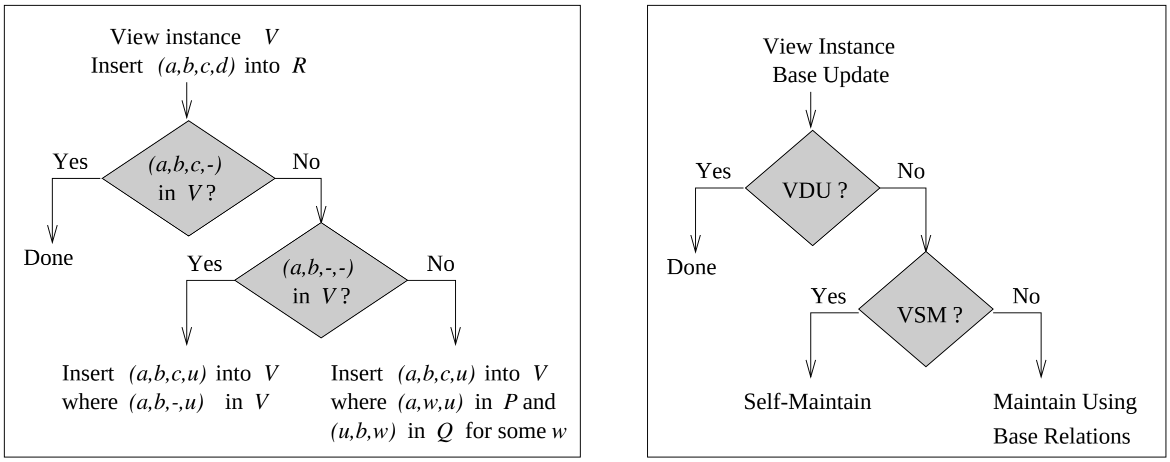
In this notation, $r(X,Y,Z,T)$, $p(X,W,U)$, and $q(U,Y,W)$ are called the *subgoals* and specify how R, P, and Q, the associated base relations, are to be joined. For instance, the sharing of variable U (variables are in upper case, constant objects in lower case) between the second and third subgoals indicates a join between P's third attribute and Q's first attribute. Furthermore, $v(X,Y,Z,U)$, called the *head*, indicates the attributes included in the projection. For instance, the fourth attribute of R, represented by variable T which does not appear in the head, is not retained in the projection. We assume view V and relations R, P, and Q are sets.

Now, consider the insertion of $r(a,b,c,d)$. To maintain V, an obvious solution would systematically try to determine the answer to the query

$$\mathcal{Q}_{remote} : \{U \mid (\exists W) \ p(a,W,U) \ \& \ q(U,b,W)\}$$

and would insert into V all tuples (a,b,c,U) whose U-component is in the answer to $\mathcal{Q}_{remote}$. If relations P and Q reside in two different remote sites, computing the answer to $\mathcal{Q}_{remote}$ is expensive because it involves a distributed join. A smarter way to maintain V would try to avoid evaluating $\mathcal{Q}_{remote}$. For example, consider the following view instance

X	Y	Z	U
a	b	c	1
a	f	e	2



(a) In Example 1.1

(b) Based on local filters in the general case

Figure 1: Plan for speeding up view maintenance.

From its contents alone, we can infer that V cannot be affected by the insertion, simply because: the database must contain tuple $R(a,b,c,x)$ for some constant x; if $\mathcal{Q}_{remote}$ produces any tuple (u), then the contributing tuples $P(a,w,u)$ and $Q(u,b,w)$ must have joined with $R(a,b,c,x)$ to produce the view tuple (a,b,c,u), and thus u must be identical to 1; after the insertion, the view gains at most tuple $(a,b,c,1)$, which is already in the view. Thus, a simple reasoning with the contents of this view instance saves us not only the work to obtain the answer to query $\mathcal{Q}_{remote}$, but also the work to update the view. Now, consider another view instance

X	Y	Z	U
a	b	e	1
a	f	e	2

From the view contents alone, we can infer that, after the insertion, V gains tuple $(a,b,c,1)$ exactly. The reasoning in this case goes as follows. First, analogous to the previous view instance, we can infer that $\mathcal{Q}_{remote}$ can only produce tuple (1), and thus, V cannot gain any tuple other than $(a,b,c,1)$. But does $\mathcal{Q}_{remote}$ produce tuple (1) at all? The answer must be affirmative in order to explain the presence of $(a,b,e,1)$ in V. Thus, V gains tuple $(a,b,c,1)$ exactly.

The view instances we just showed exemplified situations where view V can be maintained using only its own contents, totally obviating the need to compute the answer to query $\mathcal{Q}_{remote}$. But how expensive is it to detect such favorable situations? It turns out that for the view in our example, detecting these situations requires only a few lookups to the view relation, and by maintaining indexes on the X, Y, and Z view attributes, each lookup can be performed in sublinear and even constant time. Figure 1(a) shows a general plan for speeding up the incremental maintenance of view V under the insertion of $r(a,b,c,d)$. ∎

1.2 Problem Statement

We now formally define the situations described above, and we assume in the following definitions that the view is consistent, i.e., there is an underlying database that derives the view instance. Note that in the remainder of this paper, we use $\mathcal{D}$ to denote an instance of the underlying database, $\mathcal{U}$

an update to the underlying database, and $\mathcal{U}(\mathcal{D})$ the database instance that results from applying $\mathcal{U}$ to $\mathcal{D}$.

Definition 1.1 (View Unaffected by Update): Let V be a view and $\mathcal{Q}$ a query that defines the view in terms of some underlying database $\mathcal{D}$. We say that V is *definitely unaffected* by an update $\mathcal{U}$ to $\mathcal{D}$ if the view remains consistent with the updated database:

$$(\forall \mathcal{D})\ [[\mathcal{Q}(\mathcal{D}) = V] \Rightarrow [\mathcal{Q}(\mathcal{U}(\mathcal{D})) = V]]$$

The problem of determining whether or not a view is definitely unaffected by an update is called the VDU problem. Note that in the definition, VDU depends on V only (the state of the view database) but not on $\mathcal{D}$ (the state of the underlying database.) ∎

Definition 1.2 (View Self-Maintainability): Let V be a view and $\mathcal{Q}$ a query that defines the view in terms of some database $\mathcal{D}$. V is said to be *self-maintainable* under an update $\mathcal{U}$ to $\mathcal{D}$ if the new state of the view is uniquely defined, independently of the underlying database:

$$(\forall \mathcal{D}, \mathcal{D}')\ [[\mathcal{Q}(\mathcal{D}) = \mathcal{Q}(\mathcal{D}') = V] \Rightarrow [\mathcal{Q}(\mathcal{U}(\mathcal{D})) = \mathcal{Q}(\mathcal{U}(\mathcal{D}'))]]$$

The problem of determining whether or not a view is self-maintainable under an update is called the VSM problem. Also note that VSM does not depend on the state of the underlying database. It only depends on the state of the view. ∎

Definition 1.3 (View Self-Maintenance): Let V be a view defined by query $\mathcal{Q}$ in terms of some database $\mathcal{D}$ and let $\mathcal{U}$ be an update to $\mathcal{D}$. If V is self-maintainable under $\mathcal{U}$, a *self-maintenance function* $\mathcal{M}_{\mathcal{Q}}(V,\mathcal{U})$ is a function that computes the uniquely defined new state of the view from its current state and the update. In other words:

$$(\forall \mathcal{D})\ [[\mathcal{Q}(\mathcal{D}) = V] \Rightarrow [\mathcal{Q}(\mathcal{U}(\mathcal{D})) = \mathcal{M}_{\mathcal{Q}}(V,\mathcal{U})]]$$

In incremental view maintenance, $\mathcal{M}_{\mathcal{Q}}$ can be thought of as a pair of functions that compute the view increment and decrement respectively. ∎

Example 1.1 suggests a general approach to speeding up incremental view maintenance based on testing VDU and VSM as defined above, as outlined in Figure 1(b). Since the tests take the current state of the warehouse database into account, our approach can be thought of as doing *runtime* view maintenance optimization. Thus, the practicality of this approach is premised on being able to *efficiently* compute at runtime, based on the current state of the view: (1) test VDU, i.e., detect when a view is definitely not affected by a base update; (2) test VSM, i.e., detect when a view is self-maintainable under a base update; and (3) self-maintain the view, i.e., compute the required view update using only the view itself, when the view is self-maintainable. Both (1) and (2) are overhead which we desire to minimize, and (3), the work to determine the view update from the view itself, should be no more expensive than computing the view update using the base relations if the latter were local to the warehouse. The importance of efficiency cannot be overemphasized here: if trying to maintain a view using only the warehouse is not drastically cheaper than maintaining it using the base relations directly, the approach we propose to speed up view maintenance would lose some of its appeal.

In general, testing VDU and VSM efficiently is not a trivial problem, even if we restrict the view query to be conjunctive (a conjunctive query, abbreviated CQ, is a select-project-join query with only equality comparisons.) In fact, [AD97] recently showed that for CQ views in general, testing

VSM is co-NP complete (in the size of the view instance). Although it is still open whether or not testing VDU is co-NP-hard (we already know it is in co-NP), we conjecture that testing VDU is co-NP-complete as well.

This paper focuses on the following practically important question:

- assuming that we have efficient ways to implement only table lookups, are there cases where VDU and VSM can tested *efficiently* and where the required view update can be computed from the view contents efficiently?

1.3 Related Work

The runtime view maintenance optimization problem we consider in this paper was studied before. But to date, no efficient solutions are known in the literature. [TB88, GB95] studied the VSM problem for CQ views with arithmetic comparisons and the best result given there is a quadratic algorithm for testing VSM for CQ views with no self-joins. Our own work in [Huy96] gave only a very partial treatment of the range of problems considered in this paper. In particular, [Huy96] addressed the VSM problem only under insertions but not under deletions, and considered only CQ views with no self-joins. Furthermore, the VDU problem was not addressed at all. [Huy97a] gave a polynomial solution to testing VSM for CQ views with no projection in the form of SQL queries, but their tests have exponential query complexity.

The VSM problem considered in this paper is closely related to the problem of detecting *autonomously computable* updates studied in [BCL89]. The latter problem can be viewed as the *compile-time* analog of the VSM problem we study here, in which the actual contents of the view is *not* used in determining whether the view is self-maintainable under a given update. In this sense, compile-time VSM, being less aggressive than runtime VSM, is a notion more suitable for the problem of designing efficiently maintainable warehouses ([Q*96]) than the problem of speeding up maintenance of a warehouse whose design is given. Unfortunately, view maintenance methods based on compile-time VSM have limited applicability, since most views are *not* self-maintainable under insertions [BCL89]. Similarly, the VDU problem we study here has a compile-time analog, in which we are given a query $\mathcal{Q}$ and an update $\mathcal{U}$, and we want to determine whether or not $\mathcal{Q}(\mathcal{D}) = \mathcal{Q}(\mathcal{U}(\mathcal{D}))$ for every database $\mathcal{D}$, without using the current answer to $\mathcal{Q}$. [BCL89] called it the problem of detecting *irrelevant updates*, while [LS93] called it *query independent of updates*.

1.4 Contribution and Outline

In this paper, we establish the following results, for views and base relations that are sets and for updates to a single base relation:

- For CQ views with *no* self-joins, VDU and VSM under insertions and deletions can be tested and the view updates computed using only a small constant number of view lookups. The number of lookups is less than the number of subgoals in the view definition. For CQ views that allow a restricted form of self-joins, called *exposed* self-joins, the number of view lookups required is no more than the square of the number of subgoals in the view definition. Table 1 summarizes the solution complexity, where k is the number of subgoals in the view definition.

- For more general CQ views, we show a sufficient solution whose efficiency is comparable to the previous cases.

View lookups can be implemented efficiently if there are indexes on the appropriate view attributes, for instance using techniques for either indexing or hashing the search keys. Thus, the

	Views with no self-joins		Views with exposed self-joins	
	Insertion	**Deletion**	**Insertion**	**Deletion**
Testing VDU	1	1	$(k-2)^2+1$	1
Testing VSM	$(k-1)$	1	$(k-1)^2$	1
Computing View Updates	$(k-1)$	1	$(k-1)$	1

Table 1: Solution complexity (in number of view lookups) for CQ views.

practical significance of our results is that, by simply maintaining selected view indexes, we can effectively speed up the view maintenance process, incurring only a small overhead.

The rest of this paper is organized as follows. In Section 2, we show that efficiency in testing VDU and VSM and in view self-maintenance, under insertions and deletions, can be achieved for CQ views with no self-joins. Section 3 extends this result to CQ views that allow a restricted form of self-joins called exposed self-joins. Section 4 considers a larger class of views but gives solutions that are only sufficient, trading completeness for efficiency. Section 5 discusses performance of the method we propose in this paper for maintaining views. We summarize the results and discuss future work in Section 6. Throughout this paper, results are presented without their proof, which can be found in [Huy97b].

2 Efficient Solution for CQ Views with no Self-Joins

This section considers the class of views defined by CQ's with no self-joins, and only insertions or deletions to a single base relation. For this class, it turns out that testing VSM under insertions has a surprisingly simple solution that can be described entirely by how variables are shared among the subgoals in the body of the view definition. For further details, we refer the reader to [Huy97b]. To help analyze how these variables are used, we will introduce the notion of *subgoal partitioning*, and then present the solution in terms of this notion. But first, let us define some notation and terminology used in this section and later sections.

Without loss of generality, the definition of a CQ view V with no self-joins can be defined as

$$v(\bar{X}', \bar{Y}', \bar{Z}') :- r(\bar{X}, \bar{Y}) \ \& \ S(\bar{Y}, \bar{Z}) \tag{1}$$

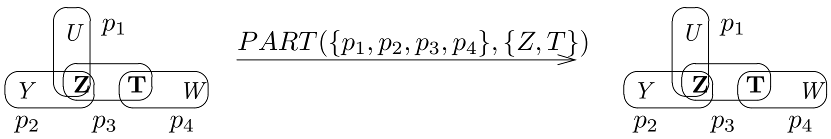
where $\bar{X}$, $\bar{Y}$, and $\bar{Z}$ are disjoint sets of variables, $\bar{X}' \subseteq \bar{X}$, $\bar{Y}' \subseteq \bar{Y}$, $\bar{Z}' \subseteq \bar{Z}$, r is the updated predicate, and S is a conjunction of subgoals with nonupdated predicates. Assume no predicate is repeated in S. The variables in $\bar{Y}$ are called *join variables.* Note that the notion of join variable is relative to which updated base relation we are considering. A variable in the body of the view definition that also appears in the head is said to be *exposed.* It is *hidden* otherwise. In our representation, even though constants and repeated variables within each subgoal are not represented explicitly, they are allowed in the view definition. We do not need to represent them explicitly because they play no role in the solution we are going to present shortly, except this: a tuple that is either inserted into or deleted from a base relation cannot affect the view if it does not match with the subgoal with that relation; thus, only tuples that match with the corresponding subgoal need to be considered; the matching of a tuple with the corresponding subgoal produces a substitution for the variables in the subgoal with constants, which is represented in our notation by $\bar{X} \rightarrow \bar{a}$ and $\bar{Y} \rightarrow \bar{b}$ for some vectors of constants $\bar{a}$ and $\bar{b}$.

2.1 Subgoal partitioning

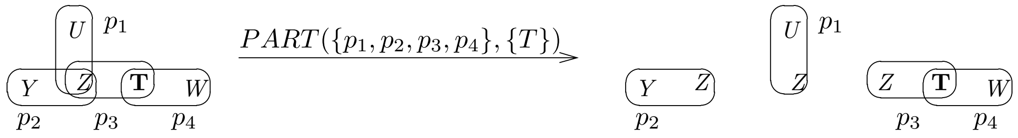
Definition 2.1 (Subgoal Partition): Let $S(\bar{U})$ be a set of subgoals with distinct predicates, where $\bar{U}$ represents the set of variables used in the subgoals. Let $\bar{V}$ be a subset of the variables in $\bar{U}$. We define $PART(S(\bar{U}), \bar{V})$ to be the finest partition of $S(\bar{U})$ into groups $S_1(\bar{U}_1)$, $S_2(\bar{U}_2)$, … , $S_n(\bar{U}_n)$, such that no two groups share some $\bar{V}$-variables. ∎

We can equivalently define $PART(S(\bar{U}), \bar{V})$ as follows. Consider a graph whose nodes correspond to the subgoals in $S(\bar{U})$ and where two nodes are connected if the corresponding subgoals share some $\bar{V}$-variable. Then the connected components in the graph correspond to the groups $S_1(\bar{U}_1)$, $S_2(\bar{U}_2)$, … , $S_n(\bar{U}_n)$.

Example 2.1 Consider partitioning the set S of subgoals $p_1(U, Z)$, $p_2(Y, Z)$, $p_3(Z, T)$, and $p_4(W, T)$. $PART(S, \{Z, T\})$ consists of a single group that includes all the subgoals, since a group that contains $p_3(Z, T)$ shares either Z or T with any other subgoal. The following illustrates this partitioning in connection hypergraph:



where the variables in the second argument to $PART$ are shown in boldface. These variables behave like glue that holds the subgoals together while we are trying to split them apart. By contrast, $PART(S, \{T\})$ consists of the three groups $\{p_1(U, Z)\}$, $\{p_2(Y, Z)\}$, and $\{p_3(Z, T), p_4(W, T)\}$:



∎

The notion of subgoal partitioning will be used as follows. For a view defined by (1) and for the insertion of $r(\bar{a}, \bar{b})$, we need to determine the most general condition on V such that $\{\bar{z} \mid S(\bar{b}, \bar{z})\}$ is independent of the underlying database. Consider partitioning the subgoals $S(\bar{Y}, \bar{Z})$ as $PART(S(\bar{Y}, \bar{Z}), \bar{Z})$. Each group g of subgoals in this partition identifies two sets of variables, $\bar{Y}_g$ and $\bar{Z}_g$, which have the following important property: the presence in V of a tuple that agrees with $\bar{b}$ over $\bar{Y}_g$ guarantees that the set of values for $\bar{Z}_g$ that satisfy all the subgoals in g in any underlying database does not depend on the database. Thus, the simultaneous presence of such tuples in V, for all group g in $PART(S(\bar{Y}, \bar{Z}), \bar{Z})$, will guarantee that V is self-maintainable under the insertion of $r(\bar{a}, \bar{b})$.

2.2 Speeding up Incremental Maintenance of Views with no Self-Joins

For views defined by (1), the solution VDU test, VSM test, and view self-maintenance function, under the *deletion* of $r(\bar{a}, \bar{b})$ and under the *insertion* of $r(\bar{a}, \bar{b})$ consist of a small number of table lookups and are summarized in Tables 2 and 3 respectively. These tables represent theorems whose proof is not given here but can be found in [Huy97b]. The tests given in these tables are both *sound and complete*. In each of the tables, the solution is presented for different syntactic cases that can be easily determined from the view definition. Let us clarify some of the notation used

Syntactic cases	VDU test	VSM test	View updates
$\bar{X}' = \bar{X}$ and $\bar{Y}' = \bar{Y}$	No $(\bar{a}, \bar{b}, -)$	Always true	Delete $(\bar{a}, \bar{b}, -)$
$\bar{X}' \neq \bar{X}$ or $\bar{Y}' \neq \bar{Y}$	No $(\bar{a}', \bar{b}', -)$	Same as VDU	None

Table 2: Speeding up view maintenance under deletions.

Syntactic cases	VDU test	VSM test	View update
$\bar{Y}' = \bar{Y}$	$(\bar{a}', \bar{b}, -)$	For each g, $(-, \bar{y}, -)$ where $\bar{y} =_{\bar{Y}_g} \bar{b}$	Insert $(\bar{a}', b, \bar{c}')$ where for each g, $\bar{c}' =_{\bar{Z}_g} \bar{z}'$ for some $(-, \bar{y}, \bar{z}')$ where $\bar{y} =_{\bar{Y}_g} \bar{b}$
$\bar{Y}' \neq \bar{Y}$, and for each g, either $\bar{Y}_g \subseteq \bar{Y}'$ or $\bar{Z}_g \cap \bar{Z}' = \emptyset$	$(\bar{a}', \bar{b}', -)$	Same as VDU	None
Otherwise	Always false	Always false	N/A

Table 3: Speeding up view maintenance under insertions.

in the tables and throughout the rest of the paper. Given vectors of constants $\bar{a}$ and $\bar{b}$, $\bar{a}'$ and $\bar{b}'$ denote their projection over attributes $\bar{X}'$ and $\bar{Y}'$ respectively. Given two vectors of constants $\bar{y}$ and $\bar{b}$, we say that they agree over variables $\bar{Y}_g$ (denoted by $\bar{y} =_{\bar{Y}_g} \bar{b}$) if their projections over $\bar{Y}_g$ are identical. In general, a table lookup consists of looking for some tuple that agrees with some constants over some variables.

To further clarify the notation we use, the VDU test expression in the first case in Table 2 denotes the absence of rows in the view with $\bar{a}$ and $\bar{b}$ as their $\bar{X}$ and $\bar{Y}$ components, and the view update expression consists of deleting all rows that match $\bar{a}$ and $\bar{b}$. In Table 3, g ranges over all the groups in $PART(S(\bar{Y}, \bar{Z}), \bar{Z})$. In the first case, a view lookup is required for each group. Since the number of groups in $PART(S(\bar{Y}, \bar{Z}), \bar{Z})$ cannot exceed the number of subgoals in S, testing VSM and computing view updates require each less than $k - 1$ view lookups, where k is the number of subgoals in the view definition. Note that Table 3 assumes S represents a nonempty set of subgoals. If S is empty (i.e., the updated relation is the only base relation in the view definition), then under the insertion of $r(\bar{a}, \bar{b})$, the VDU test degenerates to looking for $(\bar{a}', \bar{b}')$, the view is always self-maintainable and can be updated by inserting $(\bar{a}', \bar{b}')$.

Example 2.2 Consider maintaining a view V defined by

$$v(X, Y, Z, U) \text{ :- } r(X, Y, 1, Z, T) \ \& \ p(X, 2, U) \ \& \ q(Y, U, Y) \ \& \ s(Z, Z, Y)$$

Since this view has no self-joins, only the pattern of variable sharing among the subgoals matters in determining whether or not V is unaffected by or self-maintainable under a base update. In other words, constants 1 and 2 and the repetition of Y and Z within a subgoal in the view definition matter only to the extent that they can help filter out base updates that do not match any subgoal. Thus, the insertion or deletion of $r(a, b, 2, c, d)$ can be ignored since it cannot affect V. Now consider the insertion of $r(a, b, 1, c, d)$, which matches subgoal $r(X, Y, 1, Z, T)$ with the substitution $\sigma = \{X \to a, Y \to b, Z \to c, T \to d\}$. To test VDU, we detect the presence in V of a tuple that agrees, over X, Y, and Z (i.e. all the join variables), with their respective value in σ, i.e., a tuple of the form $(a, b, c, -)$. To test VSM, each group in $PART(\{p, q, s\}, \{U\})$ specifies a tuple we must look for: for group $\{p, q\}$ which uses only join variables X and Y, we look for a tuple of the form

$(a, b, -, -)$; for group $\{s\}$ which uses join variables Y and Z, we look for $(-, b, c, -)$. To update V when it is self-maintainable, we simply insert (a, b, c, u) for each u such that $(a, b, -, u)$ is in V. ∎

3 Allowing a Limited Form of Self-Joins in CQ Views

This section generalizes the results from Section 2 to CQ views that may contain self-joins. When self-joins are allowed in the view definition, testing VSM becomes more complex, and in fact, co-NP-complete with unrestricted self-joins, even if the updated predicate occurs only once ([AD97]). We need to identify restrictions on self-joins that make the problem not only tractable but also efficiently so (e.g. with linear complexity). In this section we present one such case where a solution that is both efficient and complete can be obtained. We consider view definitions that are CQ's with the following restrictions:

- The updated relation occurs only once.
- Any subgoal with a repeated predicate uses only exposed join variables. Self-joins with this restriction are called *exposed* self-joins.

To guarantee view self-maintainability in general, we need to find conditions on the view that guarantee that the satisfaction of the nonupdated subgoals does not depend on the underlying database. While such a condition is unique for a subgoal that uses a distinct predicate, it is no longer so if the subgoal uses a predicate that also occurs in other subgoals (a predicate that occurs in more than one subgoal is said to be *repeated*). This complexity is due to the potential interactions among subgoals with the same predicate. To capture this complexity, we first define and characterize the notion of subgoal *validity*. This notion is then used to express the solution.

3.1 View Conditions For Subgoal Satisfaction

Let V be a view defined by a CQ that may have self-joins. Consider a subgoal I whose variables, $\bar{X}_I$, are all *exposed*. Given a substitution σ_I of $\bar{X}_I$ with some constant $\bar{x}_I$ (denoted $\bar{X}_I \to \bar{x}_I$), we would like to find the most general condition on the view that guarantees the *validity* of $\sigma(I)$, that is, $\sigma(I)$ is satisfied in every instance of the base relations that is consistent with the given view. We are interested in such a condition because they will used later to express the most general view self-maintainability conditions.

When the predicate used in I is not repeated (i.e., not involved in any self-join) in the body of the view definition, the most general condition that guarantees $\sigma(I)$'s validity is trivial: simply look for a tuple in V that agrees with $\bar{x}_I$ over $\bar{X}_I$. However, when the predicate used in I is repeated, the presence of such tuple in V still guarantees $\sigma(I)$'s validity but is no longer the most general condition. The following example illustrates this added complexity introduced by self-joins.

Example 3.1 Consider the view definition

$$v(X, Y, Z) \coloneq p(d, X, Y) \ \&\ p(Z, Y, Z)$$

Let a be a constant and consider the validity of $p(d, a, d)$ under a given view instance V. The presence of $(a, d, -)$ in V is sufficient to guarantee $p(d, a, d)$'s validity, but is not necessary. In fact, it is easy to see that the presence of $(-, a, d)$ also guarantees $p(d, a, d)$'s validity. ∎

For a CQ view and for a particular subgoal, let us now define the notion of most general condition on the view that guarantees that a given instantiation of the subgoal is valid, i.e., can be satisfied in all database instances consistent with the view.

Definition 3.1 (Valid Subgoal Instantiation): Let V be a CQ view and let I be a subgoal in the body of the view definition. Let $\sigma_I : \bar{X}_I \rightarrow \bar{x}_I$ be a substitution of I's variables, $\bar{X}_I$, with some constants $\bar{x}_I$. We define $VALID(I, \sigma_I)$ to be the condition on V that, when evaluated to true, guarantees $\sigma_I(I)$ hold in any database instance consistent with V, and when evaluated to false, guarantees the existence of a database instance consistent with V where $\sigma_I(I)$ is false. ▮

When a subgoal I is involved in an exposed self-join, that is, when every subgoal J with the same predicate as I uses only exposed join variables, $VALID(I, \sigma_I)$ can be completely characterized by simple view lookups, as stated in the following theorem.

Theorem 3.1 *Let V be a CQ view and let I be a subgoal such that every subgoal J with the same predicate as I uses only exposed join variables. Let $\sigma_I : \bar{X}_I \rightarrow \bar{x}_I$ be a constant substitution. Then $VALID(I, \sigma_I)$ can be characterized as*

$$\bigvee_J [\ \alpha_{J,I} \wedge (\exists \bar{x})\ [v(\bar{x}) \wedge \bar{x} =_{\bar{X}_I} \bar{x}_{J,I}]]$$

where J ranges over all the subgoals with the same predicate as I, and for each J, $\alpha_{J,I}$ and $\bar{x}_{J,I}$ are defined as follows:

- *$\alpha_{J,I}$ is the most general condition on the constants that guarantees that J can match with $\sigma_I(I)$, i.e., that there is some substitution that turns J into $\sigma_I(I)$. $\alpha_{J,I}$ is generally a set of equality conditions that relate the constants from $\bar{x}_I$, I, and J.*
- *$\bar{x}_{J,I}$ represents the constants in some substitution $\sigma_{J,I} : \bar{X}_J \rightarrow \bar{x}_{J,I}$ that makes J identical to $I(\bar{x}_I)$. In general, $\bar{x}_{J,I}$ consists of both constants from $\bar{x}_I$ and constants that appear in subgoal I.*

▮

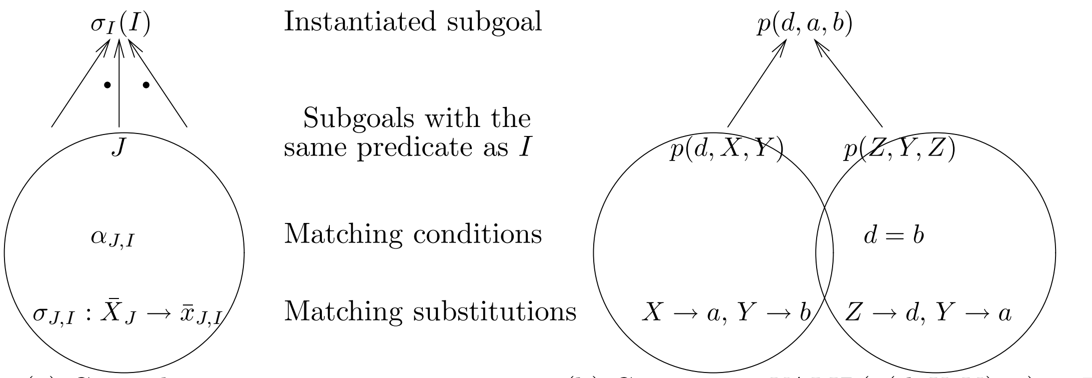
Figure 2(a) illustrates the process of matching $\sigma_I(I)$ with each subgoal J with the same predicate as I involved in computing $VALID(I, \sigma_I)$. Note that unlike the case where the view definition has no self-joins, constants and the specific way variables occur in the self-joins may have a direct impact on the expression of $VALID(I, \sigma_I)$. As a consequence, the VSM condition, which we will see depends on $VALID(I, \sigma_I)$, is sensitive to the way constants and variables appear in individual subgoals and not just how variables are shared among different subgoals.

Example 3.2 Consider the view definition from Example 3.1. Let a and b be two constants, not necessarily distinct from d. Consider the substitution σ such that $\sigma(X) = a$ and $\sigma(Y) = b$. To compute $VALID(p(d, X, Y), \sigma)$, we try to match each subgoal with $p(d, a, b)$. The first subgoal, $p(d, X, Y)$, matches with $p(d, a, b)$ unconditionally with the substitution $\{X \rightarrow a, Y \rightarrow b\}$. The second subgoal, $p(Z, Y, Z)$, matches with $p(d, a, b)$ with the substitution $\{Z \rightarrow d, Y \rightarrow a\}$, provided that $(d = b)$. Thus, $VALID(p(d, X, Y), \sigma)$ specifies the following alternative tuples to detect in V: either $(a, b, -)$, or $(-, a, d)$ if d and b are identical. Figure 2(b) illustrates the process of computing $VALID(p(d, X, Y), \sigma)$. ▮

3.2 Speeding up Incremental Maintenance of Views with Exposed Self-Joins

In general, a CQ view V with exposed self-joins (but where the update predicate occurs only once) can be defined as

$$v(\bar{X}', \bar{Y}', \bar{Z}') :- r(\bar{X}, \bar{Y})\ \&\ M(\bar{U})\ \&\ S(\bar{V}, \bar{Z}) \tag{2}$$



(a) General case (b) Computing $VALID(p(d, X, Y), \sigma)$ in Example 3.2

Figure 2: Computing $VALID(I, \sigma_I)$.

Syntactic cases	**VDU test**	**VSM test**	**View update**
$\bar{Y}' = \bar{Y}$	$(\bar{a}', \bar{b}, -)$	For each $I \in M$, one of the tuples in $VALID(I, \bar{U}_I \to \bar{b}_I)$, and for each g, $(-, \bar{y}, -)$ where $\bar{y} =_{\bar{V}_g} \bar{b}$	Insert $(\bar{a}', \bar{b}, \bar{c}')$ where for each g, $\bar{c}' =_{\bar{Z}_g} \bar{z}'$ for some $(-, \bar{y}, \bar{z}')$ where $\bar{y} =_{\bar{V}_g} \bar{b}$
$\bar{Y}' \neq \bar{Y}$, and for each g, either $\bar{V}_g \subseteq \bar{Y}'$ or $\bar{Z}_g \cap \bar{Z}' = \emptyset$	$(\bar{a}', \bar{b}', -)$ and for each $I \in M$, one of the tuples in $VALID(I, \bar{U}_I \to \bar{b}_I)$	Same as VDU	None
Otherwise	Always false	Always false	N/A

Table 4: Speeding up view maintenance under insertions.

where $\bar{X}$, $\bar{Y}$, and $\bar{Z}$ are disjoint sets of variables, $\bar{X}' \subseteq \bar{X}$, $\bar{Y}' \subseteq \bar{Y}$, $\bar{Z}' \subseteq \bar{Z}$, $\bar{U} \cup \bar{V} = \bar{Y}$, $\bar{U} \subseteq \bar{Y}'$, r is the updated predicate, M represents all the nonupdated subgoals with repeated predicates, and S all the nonupdated subgoals with nonrepeated predicates.

For views defined by (2), while testing VDU and VSM under the *deletion* of $r(\bar{a}, \bar{b})$ has the same solution as in the case where the view definition has no self-joins (i.e., given in Table 2), testing under the insertion of $r(\bar{a}, \bar{b})$ is slightly more complex than in the case of views with no self-joins. The solution is summarized in Table 4, which represents theorems whose proof is not given here but can found in [Huy97b]. Again, the tests presented are both sound and complete. Note that in Table 4, g ranges over all the groups in $PART(S(\bar{V}, \bar{Z}), \bar{Z})$, and I ranges over all the subgoals with repeated predicates. Also note that in the first and second cases, for each I, only one (but not necessarily all) of the tuples specified in $VALID(I, \bar{U}_I \to \bar{b}_I)$ need to be present to satisfy the VSM test. For a predicate p that is repeated with multiplicity $mult(p)$, there are $mult(p)^2$ lookups in the worst case. And for the entire view definition, the worst case in terms of the total number of lookups required for testing VSM happens when all the nonupdated subgoals use the same predicate. The total number of lookups in that case is $(k-1)^2$, where k is the number of subgoals in the view definition.

Note that Table 4 assumes S represents a nonempty set of subgoals. When S is empty, view definition (2) degenerates to $v(\bar{X}', \bar{Y}) :- r(\bar{X}, \bar{Y})\ \&\ M(\bar{Y})$. Consequently, testing VDU simplifies to looking for $(\bar{a}', \bar{b})$, testing VSM to looking for one of the tuples in $VALID(I, \bar{U}_I \to \bar{b}_I)$ (for each

$I \in M$), and self-maintenance to inserting $(\bar{a}', \bar{b})$.

Example 3.3 Consider a view V defined by

$$v(X, Y, Z, T) :- r(X, Y, Z) \ \& \ p(b, X, Y) \ \& \ p(Z, Y, Z) \ \& \ q(X, Z, T)$$

and consider the insertion of $r(a, b, c)$, where a, b, and c are distinct constants. In this view definition, p is the only repeated predicate and both subgoals with predicate p use only join variables that are exposed. Thus, $VALID(p(b, X, Y), \{X \rightarrow a, Y \rightarrow b, Z \rightarrow c\})$ specifies one of $(a, b, -, -)$ and $(-, a, b, -)$ to look for. $VALID(p(Z, Y, Z), \{X \rightarrow a, Y \rightarrow b, Z \rightarrow c\})$ specifies only $(-, b, c, -)$ to look for, since $p(c, b, c)$ and $p(b, X, Y)$ do not match. Finally, $q(X, Z, T)$, the only subgoal with a nonrepeated and nonupdated predicate, specifies $(a, -, c, -)$ to look for. ∎

4 Trading Completeness for Efficiency

In the previous two sections, solution efficiency was achieved by restricting the class of allowable views while retaining solution completeness. However, in order to handle larger classes of views with comparable efficiency, we may need to sacrifice completeness. This section illustrates this approach with a class of views that is more general than those considered in the previous sections: CQ views where the updated relation occurs only once, but with arbitrary self-joins. Note that testing VSM even for these views is co-NP-complete ([AD97]), and no solutions that are both complete and efficient are likely to exist.

One way to find solutions for these views is to "ignore" the self-joins, i.e., treat each occurrence of a repeated predicate as a distinct predicate. In other words, we reduce the problem of maintaining CQ views with arbitrary self-joins to a problem of maintaining CQ views with no self-joins or with only exposed self-joins. While completeness may not be preserved by the reduction, the following theorem shows that soundness is preserved, which is critical.

Theorem 4.1 *Let V be a view defined by conjunctive query $\mathcal{Q}$. Let $\mathcal{U}$ be an update (insertion or deletion) to some base relation. Let $\mathcal{Q}'$ be the query obtained from $\mathcal{Q}$ by replacing some occurrence of some predicate other than the updated predicate with a new predicate. Then*

1. *If V is not affected by (resp. is self-maintainable under) $\mathcal{U}$ under view definition $\mathcal{Q}'$, then V is also not affected by (resp. is self-maintainable under) $\mathcal{U}$ under view definition $\mathcal{Q}$.*
2. *If V is self-maintainable under $\mathcal{U}$ under view definition $\mathcal{Q}'$, then any self-maintenance function under view definition $\mathcal{Q}'$ is also a self-maintenance function under view definition $\mathcal{Q}$.*

∎

As a consequence of Theorem 4.1, even if the view definition contains self-joins that are not exposed, we can still solve the problem by reducing it to a problem where all self-joins are exposed, which we know how to solve efficiently. The following example illustrates this reduction.

Example 4.1 Consider a view V defined by

$$v(X, Y, Z, T) :- r(X, Y, Z) \ \& \ p(b, X, Y) \ \& \ p(Z, Y, Z) \ \& \ p(X, Z, T)$$

and consider the insertion of $r(a, b, c)$. Since variable T in the last subgoal is not a join variable, this view has a self-join that is not exposed and the results from Section 3 are not applicable. However, in order to test VSM under the insertion, one solution can be obtained by ignoring all self-joins, i.e., by treating each of the three occurrences of predicate p as a distinct predicate. Under the modified view definition

$$v(X, Y, Z, T) :- r(X, Y, Z) \ \& \ p_1(b, X, Y) \ \& \ p_2(Z, Y, Z) \ \& \ p_3(X, Z, T),$$

the following VSM test can be derived using the results from Section 2:

$$v(a, b, -, -) \wedge v(-, b, c, -) \wedge v(a, -, c, -)$$

A better (more general) solution can be obtained by replacing only those occurrences of a repeated predicate in subgoals that use either hidden variables or non-join variables with distinct predicates. Thus the modified view definition

$$v(X, Y, Z, T) :- r(X, Y, Z) \ \& \ p(b, X, Y) \ \& \ p(Z, Y, Z) \ \& \ p_3(X, Z, T)$$

has only exposed self-joins, and we can use the results from Section 3 to obtain a VSM test similar to that derived in Example 3.3:

$$[v(a, b, -, -) \vee v(-, a, b, -)] \wedge v(-, b, c, -) \wedge v(a, -, c, -)$$

∎

5 Performance

This section discusses when the method we propose in this paper for view maintenance has any performance advantage over the traditional method which computes view updates from the nonupdated base relations directly. The following lists key factors that impact performance of our method:

- *How much querying the view is more efficient than querying the source.* Querying the source is expensive often because the nonupdated source relations do not have indexes defined on the appropriate join attributes. The matter becomes worse when the query result is transmitted back to the warehouse over a slow network connection or when the nonupdated source resides on separate sites, requiring a distributed join. Depending on the application, querying the source directly can be a few orders of magnitude slower than querying the view.
- *How often the VDU and VSM tests succeed.* A base insertion satisfies these tests when the values in certain attributes in the inserted row are already in the view. Thus, when these values repeatedly appear in a stream of insertions, the VDU and VSM tests are highly likely to succeed. Conversely, when the inserted rows involved mostly new values, for instance when the attributes in question are key attributes, these tests are not likely to succeed.
- *Cost of maintaining the indexes on the view.* Maintaining indexes on a table does not come for free and updating an indexed view can be many times slower than updating the same view but with no indexes defined.
- *Cost of evaluating the VDU and VSM tests.* Since these tests consist of simple view lookups, we can drastically reduce their costs by using appropriate indexes on the view. With these indexes in place, the cost of these tests is typically dominated by other costs.

Notation	Parameter Description	Measured In Experiments
t_{vdu}	Average time to query the existence of $v(x, y, -)$.	.31 msec
t_{vsm}	Average time to query the existence $v(x, -, -)$.	.28 msec
t_{qbase}	Average time to query $s(x, z)$ for the z values.	2.2 msec
t_{qview}	Average time to query $v(x, -, z)$ for the z values.	2.2 msec
t_{ins}	Average time to insert $v(x, y, z)$ in the absence of indexes.	1.9 msec
t_{xins}	Average time to insert $v(x, y, z)$ in the presence of indexes.	4.8 msec
p_{vdu}	Probability that $v(x, y, -)$ exists.	0%–90%
p_{vsm}	Probability that $v(x, -, -)$ exists.	0%–99%

Table 5: Parameters used in our Analysis.

What combined effect do these factors have on performance? Of course, the precise answer depends on the particular view definition, the particular mix of base updates, and the particular cost of querying the source relations. In the remainder of this section, we focus our analysis on a case study. Through a simple scenario, we can more precisely pinpoint cases where our method is not beneficial and cases where our method can significantly improve performance over the traditional method. Consider the view definition

$$v(X, Y, Z) :\!-\ r(U, X, Y)\ \&\ s(X, Z)$$

and for simplicity sake, let us consider only insertions to R. Under a stream of insertions of $r(u, x, y)$, we compare the performance of our view maintenance plan , expressed in pseudo code as:

> **if** $v(x, y, -)$ holds **then** done.
> **else if** $v(x, -, -)$ holds **then insert** $v(x, y, z)$ for each z such that $v(x, -, z)$ holds.
> **else insert** $v(x, y, z)$ for each z such that $s(x, z)$ holds.

and where indexes are defined on the X and Y attributes of the view, with the traditional plan which simply consists of:

> **insert** $v(x, y, z)$ for each z such that $s(x, z)$ holds.

and where no indexes are defined on the view. Let us emphasize that this traditional plan has been highly simplified. In a more realistic plan, we need to ensure the insertions do not introduce duplicates in the view. This additional work would make the traditional plan even less efficient. Therefore, the analysis that follows is conservative.

5.1 Performance Analysis

Table 5 lists the main parameters we use to analyze performance. Note that since VDU is a special case of VSM, the probability that VSM succeeds and VDU fails is $(p_{vsm} - p_{vdu})$, and the probability that both tests fails is $(1 - p_{vsm})$. The following gives a good approximation for *speedup*, the ratio between the running time of the traditional plan and the average running time of our plan:

$$speedup = \frac{t_{qbase} + t_{ins}}{t_{vdu} + t_{vsm}(1 - p_{vdu}) + (t_{qview} + t_{xins})(p_{vsm} - p_{vdu}) + (t_{qbase} + t_{xins})(1 - p_{vsm})} \tag{3}$$

Table 6 shows the speedup factor under extreme cases, when t_{qbase} either is comparable to t_{qview} or is very large, and p_{vsm} is close to either 0% or 100%. The upper left cell in Table 6 represents

		p_{vsm}	
		low	high
t_{qbase}	low	$\dfrac{t_{qbase} + t_{ins}}{t_{vdu} + t_{vsm} + t_{qbase} + t_{xins}}$	$\dfrac{t_{qbase} + t_{ins}}{t_{vdu} + (t_{vsm} + t_{qview} + t_{xins})(1 - p_{vdu})}$
	high	1	$1/(1 - p_{vsm})$

Table 6: *Speedup* Under Low and High Values for t_{qbase} and p_{vsm}.

the worst case, when both VDU and VSM tests never succeed and querying the source is no less efficient than querying the view. As expected, our method is less efficient than the traditional method, but our method is only limited by how fast we can maintain the view indexes (in one of the experiments we show later, the traditional method is less than twice as fast as our method.) But when the cost of querying the source dominates the cost of querying the view (lower left cell in Table 6), the difference in performance becomes minimal. Both right cells in Table 6 represent cases where either VDU or VSM succeeds often. The lower right cell shows the asymptotic speedup value when the cost of querying the source grows faster than p_{vsm} approaches 100% (in some experiments, our method is between one and two orders of magnitude faster than the traditional method.)

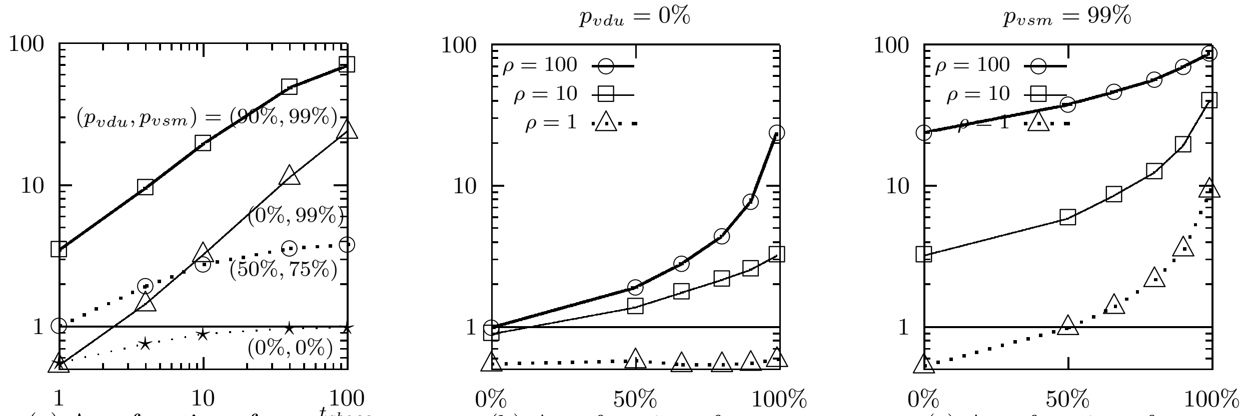
While most parameters listed in Table 5 can be measured easily, it is not immediately obvious how to obtain p_{vsm} and p_{vdu}. However, observe that when $r(u, x, y)$ is inserted, VSM fails in only two cases: either x appears for the first time in the stream of insertions, or x never appears as an X-value in S, the nonupdated relation. In other words, the VSM test will succeed for all subsequently inserted tuples that have the same value x, if x also appears in S. Let $R(X)$ denote the set of distinct X-values in R, the base relation after the insertions have been applied. The number of tuples in R whose X-value has already appeared in the insertion stream is $(card_R - card_{R(X)})$. Of these tuples, some do not join with S and thus do not contribute to the view. Those who join with S are exactly the tuples that satisfy VSM. Therefore, assuming uniform distribution of the X-values in R, a good approximation for p_{vsm} can be obtained as follows:

$$\left(1 - \frac{card_{R(X)}}{card_R}\right) \frac{card_{R(X) \cap S(X)}}{card_{R(X)}}$$

where $S(X)$ denotes the set of distinct X-values in S. Similarly, $p_{vdu} \approx \left(1 - \frac{card_{R(X,Y)}}{card_R}\right) \frac{card_{R(X) \cap S(X)}}{card_{R(X)}}$. In practice, good estimates for the parameters involved in these formulas can be obtained easily.

5.2 Experimental Results

We wrote a program that simulates view maintenance using either our plan or the traditional plan, as sketched at the beginning of this section. In our experiments, we used two tables: a table V for the view and a table S for the nonupdated base relation. Initially, V is empty. To ensure that insertions and queries do not have trivial execution times, we populated S with values uniformly distributed among its X and Z-components. The X-values in S were chosen so that any tuple inserted into R will join with S. An index was created on $S(X)$ to speed up base queries. We use an index on $V(X, Y)$ for view maintenance using our plan, and no index otherwise. We simulated insertions to R by simply iterating over a set of 100 to 1000 different tuples (u, x, y). In each experiment, we used a different insertion mix by choosing a different number of distinct values x, y, or u can take. We also simulated different values for t_{qbase}, the average time to query base relation



(a) As a function of $\rho = \frac{t_{qbase}}{t_{qview}}$ (b) As a function of p_{vsm} (c) As a function of p_{vdu}

Figure 3: *Speedup* factor.

S, by simply adding a delay that consists of artificially executing the base query multiple times. The program, written in Oracle PL/SQL, is shown in Appendix A.

Figure 3(a) shows *speedup* as a function of ρ, the factor by which querying the view is more efficient than querying the source. For this result, we experimented with four different insertion mixes, where a uniform distribution is used for the values in each component of the inserted tuples.

- Only the X-component changes. For this insertion mix, both VDU and VSM always fail.
- Only the Y-component changes (ranging over 100 different values), and both the U and X-components are constant. For this mix, VSM succeeds only half of the time, and VDU always fails.
- Each of the U, X, and Y-components ranges over 10 different values. For this mix, VDU and VSM succeed 90% and 99% of the time respectively.
- Each of the U and Y-components ranges over 2 different values, and the X-component ranges over 50. This mix represents an "intermediate" case, where VDU and VSM succeed 50% and 75% of the time respectively.

Note that within each of these insertion streams, the insertion order should not affect the performance result. Recall that in our analysis, only the image size for the X component and the XY-component matters.

For the worst insertion mix ($p_{vdu} = p_{vsm} = 0\%$), not only we realize no savings but also we pay for the overhead of testing and maintaining the view indexes. Our method underperforms the traditional method, interestingly by a factor of two only. However the gap narrows for large ρ values, since the cost of base query dominates the overhead. In the next insertion mix ($p_{vdu} = 0\%$, $p_{vsm} = 99\%$), most of the savings come from cheaper view updates evaluation (by querying the view instead of the base relation). Our method begins to outperform the traditional method as soon as ρ reaches 2, and the speedup factor is almost linear in ρ. In the best insertion mix we experimented ($p_{vdu} = 90\%$, $p_{vsm} = 99\%$), we enjoy additional savings obtained from avoiding unnecessary view updates. Note that for small values of ρ, the speedup factor is even larger than ρ, since the savings come more from avoiding unnecessary view updates than from cheaper view updates evaluation. As ρ increases, the latter type of savings becomes more and more preponderant.

We achieve a substantial speedup factor of 70 when ρ reaches 100 (theoretical limit for *speedup* is 100 for this insertion mix.) Finally, for the intermediate insertion mix ($p_{vdu} = 50\%$, $p_{vsm} = 75\%$), even if querying the view is no more efficient than querying the base relation, our method does not underperform the traditional method: the savings obtained from avoiding unnecessary view updates are able to offset the overhead of testing and maintaining the view indexes. But for large values of ρ, our method outperforms the traditional method easily, due to additional savings obtained from cheaper view updates evaluation.

Figures 3(b) and 3(c) show *speedup* when we vary p_{vsm} and p_{vdu}: we fix p_{vdu} to 0% and vary p_{vsm} in Figure 3(b), and we fix p_{vsm} to 99% and vary p_{vdu} in Figure 3(c). Recall that since VSM subsumes VDU, p_{vdu} can never be larger than p_{vsm}. In Figure 3(b), note how *speedup* is independent of p_{vsm} when $\rho = 1$. This is because all of the potential savings come from cheaper view updates evaluation, but querying the view and querying the base relation cost the same. For larger values of ρ, the dependency of *speedup* on p_{vsm} becomes more pronounced. In Figure 3(c), the degree of dependency of *speedup* on p_{vdu} is reversed. In general, as p_{vdu} increases, the source of savings shifts from cheaper view updates evaluation to avoiding unnecessary view updates. The effects of this shifting is more pronounced for lower values of ρ, since the savings obtained from cheaper view updates evaluation are small compared with the savings obtained from avoiding unnecessary view updates.

In summary, both the base insertion mix and the efficiency of view queries (relative to base queries) do have a strong performance impact on our method, as expected. In the worst case, our method may be less efficient than the traditional method, but the degradation in performance is limited by how well we can manage the costs of testing (indexed lookups) and maintaining view indexes (indexed insertion) in the worst case. However, the payoff of our method in favorable cases can be substantial. The experimental results fits well with our analysis.

Our method does not exclude the use of other view maintenance methods. For instance, we can use the traditional method to propagate most base updates and apply our method to handle selected base updates. Heuristics for choosing our method include: small image sizes for the updated attributes (relative to the updated base relation), large percentage of the updated relation joining with the nonupdated base relations, efficient view queries (relative to base queries). Through a case study, we demonstrated techniques for predicting performance. While further work is needed to apply and extend these techniques to the general case, we believe we made a useful first step.

6 Conclusion and Future Work

In a data warehouse environment, incremental view maintenance can be expensive. We proposed a practical method to speed up view maintenance that hinges on being able to efficiently detect situations where the view is either unaffected by or self-maintainable under a base update. These tests take the current state of the data warehouse into account. We identified important subclasses of CQ views that admit linear-time solutions in the overhead, which can be implemented with sublinear running time if selected indexes on the view relation are maintained. In this paper, we presented the results for single updates to the base relations. These results extend easily to sets of updates to the same base relation [Huy97b]. In future work, we seek other important subclasses of CQ views with comparable efficiency. Of particular interest is the class of views defined by acyclic queries ([Yan81]), for which many hard database problems have efficient solutions. We are also interested in extending the techniques presented in this paper to handle updates that consist of arbitrary mixes of insertions and deletions to more than one base relation. Finally, we would like to develop additional techniques for trading completeness for efficiency and for predicting performance of our method in general.

References

[AD97] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *Proc. 17th ACM Symp. on PODS*, pp. 254–263, Seattle, Washington, 1998.

[BCL89] J. A. Blakeley, N. Coburn, and P. A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. In *ACM TODS* **14**:3, pp. 369–400, 1989.

[BC79] O. P. Buneman and G. K. Clemons. Efficiently monitoring relational databases. In *ACM Trans. on Database Systems* **4**:3, pp. 368–382, 1979.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. 17th Int. Conf. on Very Large Data Bases*, pp. 577–589, 1991.

[GB95] A. Gupta and J. A. Blakeley. Using partial information to update materialized views. In *Information Systems* **20**:8, pp. 641–662, 1995.

[GLT97] T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for the incremental recomputation of active relational expressions. In *IEEE TKDE* **9**:3, pp. 508–511, 1997.

[Huy96] N. Huyn. Efficient view self-maintenance. In *Proc. Int. Workshop on Materialized Views: Techniques and Applications*, pp. 17–25, Montreal, Quebec, 1996.

[Huy97a] N. Huyn. Multiple-view self-maintenance in data warehousing environments. In *Proc. 23rd Int. Conf. on Very Large Data Bases*, pp. 26–35, Athens, Greece, 1997.

[Huy97b] N. Huyn. *Maintaining data warehouses under limited source access.* Ph.D. Thesis, pp. 51–86, Computer Science Department, Stanford University, Sept. 1997. Also available as URL `http://www-db.stanford.edu/pub/papers/ huyn-thesis.ps`.

[LS93] A. Levy and Y. Sagiv. Queries independent of updates. In *Proc. 19th Int. Conf. on Very Large Data Bases*, pp. 171–181, Dublin, Ireland, 1993.

[Q*96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. 4th Int. Conf. on PDIS*, Miami Beach, FL, Dec. 1996.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. In *IEEE Trans. on Knowledge and Data Engineering* **3**:3, pp. 337–341, 1991.

[SJ96] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *Proc. 22nd Int. Conf. on Very Large Data Bases*, pp. 75–86, Mumbai, India, 1996.

[TB88] F. W. Tompa and J. A. Blakeley. Maintaining materialized views without accessing base data. In *Information Systems* **13**:4, pp. 393–406, 1988.

[Yan81] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. 7th Int. Conf. on Very Large Data Bases*, pp. 82–94, 1981.

A Appendix: Program Used in our Experiments

Our view maintenance plan is implemented by procedure `sim_vsm`, and the traditional plan by procedure `sim_qsrc`. The plot in Figure 3 for instance were obtained by running both procedures with arguments `(1,100,1,delay)`, `(1,1,100,delay)`, `(10,10,10,delay)`, and `(2,50,2,delay)`, with `delay = 1, 4, 10, 40, and 100`. Here is the program used in our experiments, written in Oracle PL/SQL.

```
DROP TABLE v;
CREATE TABLE v(x INTEGER, y INTEGER, z INTEGER);
CREATE INDEX v_idx ON v(X,Y);
DROP TABLE v1;
CREATE TABLE v1(x INTEGER, y INTEGER, z INTEGER);

DROP TABLE s;
CREATE TABLE s(x INTEGER, z INTEGER);
CREATE INDEX s_idx ON s(X);

BEGIN
  FOR xval IN 1 .. 100 LOOP
    FOR zval IN 1 .. 100 LOOP
      INSERT INTO s VALUES(xval, zval);
    END LOOP;
  END LOOP;
END;
/
CREATE OR REPLACE PROCEDURE mydelay(xval IN INTEGER, m IN INTEGER) AS
   n INTEGER;
   BEGIN
     FOR i IN 1 .. m LOOP
       SELECT sum(z) INTO n FROM s
       WHERE x=xval;
     END LOOP;
   END mydelay;
/
CREATE OR REPLACE FUNCTION vdu
   (xval IN INTEGER, yval IN INTEGER)
   RETURN BOOLEAN IS
   CURSOR cv IS SELECT z FROM v WHERE x=xval AND y=yval;
   tmp INTEGER;
   result BOOLEAN;
   BEGIN
```

```
    result := FALSE;
    OPEN cv;
    FETCH cv INTO tmp;
    IF cv%FOUND THEN result := TRUE; END IF;
    CLOSE cv;
    RETURN result;
  END vdu;
/
CREATE OR REPLACE FUNCTION vsm
  (xval IN INTEGER, yval OUT INTEGER)
  RETURN BOOLEAN IS
  CURSOR cv IS SELECT y FROM v WHERE x=xval;
  result BOOLEAN;
  BEGIN
    result := FALSE;
    OPEN cv;
    FETCH cv INTO yval;
    IF cv%FOUND THEN result := TRUE; END IF;
    CLOSE cv;
    RETURN result;
  END vsm;
/
CREATE OR REPLACE PROCEDURE sim_vsm
  (umax IN INTEGER, xmax IN INTEGER, ymax IN INTEGER, delay IN INTEGER) AS
  ytmp INTEGER;
  BEGIN
    FOR uval IN 1 .. umax LOOP
      FOR xval IN 1 .. xmax LOOP
        FOR yval IN 1 .. ymax LOOP
          IF NOT vdu(xval,yval)
          THEN
            IF vsm(xval,ytmp)
            THEN
              INSERT INTO v SELECT xval, yval, z FROM v
              WHERE x=xval AND y=ytmp;
            ELSE
              mydelay(xval, delay-1);
              INSERT INTO v SELECT xval, yval, z FROM s
              WHERE x=xval;
            END IF;
          END IF;
```

```
        END LOOP;
      END LOOP;
    END LOOP;
  END sim_vsm;
/
CREATE OR REPLACE PROCEDURE sim_qsrc
  (umax IN INTEGER, xmax IN INTEGER, ymax IN INTEGER, delay IN INTEGER) AS
  BEGIN
    FOR uval IN 1 .. umax LOOP
      FOR xval IN 1 .. xmax LOOP
        FOR yval IN 1 .. ymax LOOP
           mydelay(xval, delay-1);
           INSERT INTO v1 SELECT xval, yval, z FROM s
           WHERE x=xval;
        END LOOP;
      END LOOP;
    END LOOP;
  END sim_qsrc;
/
```