Multiple-View Self-Maintenance in Data Warehousing Environments

Nam Huyn *
Stanford University
huyn@cs.stanford.edu

Abstract

A data warehouse materializes views derived from data that may not reside at the warehouse. Maintaining these views efficiently in response to base updates is difficult, since it may involve querying external sources where the base data reside. This paper considers the problem of *view self-maintenance*, where the views are maintained without using all the base data. Without full use of the base data, however, maintaining a view unambiguously is not always possible. Thus, the two critical questions that must be addressed are to determine, in a given situation, whether a view is maintainable, and how to maintain it.

We provide algorithms that answer these questions for a general class of views, and for an important subclass, generate SQL queries that test whether a view is maintainable and update the view if it is. We improve significantly on previous work by solving the view self-maintenance problem in the presence of multiple views, with optional access to a subset of the base data, and under arbitrary mixes of insertions and deletions. We also provide better insight into the view self-maintenance problem by reducing it to a problem of deciding query containment that can be solved.

*Work supported by ARO grant DAAH04–95–1–0192.

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

1 Introduction

Data warehouses have gained importance in recent years ([RED, IK93, Z*95]). A data warehouse is a collection of materialized views derived from relations that may not reside at the warehouse. As a benefit, user queries can often be evaluated much more cheaply using these stored views than using the base relations. The problem, however, is that the views must be updated to reflect changes made to the base relations. In many current commercial data warehousing systems, maintenance of the views is deferred until night time when the views are recomputed from scratch. This solution may be satisfactory today, yet there is increasing demand for a more immediate maintenance solution, where the views are *incrementally* brought up to date in response to base updates. Thus, techniques for incremental view maintenance are important, and, even if immediate maintenance is not required, they can help shorten the maintenance downtime. Incremental view maintenance, however, can still be expensive. For instance, maintaining views defined as a join in response to an update to a base relation usually involves looking up the non-updated base relations, which may reside in external sources.

Thus, in data warehousing environments where maintenance is performed locally at the warehouse, an important incremental view-maintenance issue is how to minimize external base data access. The idea of avoiding base access to speed up view maintenance is illustrated in Figure 1. We take the following approach to reduce maintenance costs. In response to a base update, we try to maintain the views using information that is strictly local to the warehouse. This information includes the view definitions and the contents of all the views. Only when we fail to do so do we resort to accessing the base relations.

As a result of not using all the base relations, there may be situations where there is not enough information to maintain a view unambiguously, even if we are given the specific contents of the views, the base re-

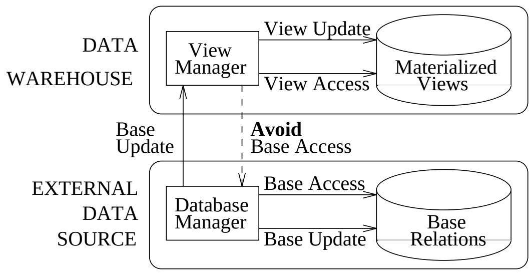

Figure 1: Saving Base Access in View Maintenance.

lations used, and the base update. Such situations never arise in traditional work on materialized view maintenance ([GM95, Kuc91, GMS93, SJ96]) where all the base data is usually assumed to be available. Thus, an important question (originally considered in [TB88, Hu96]), which was never raised in traditional view-maintenance work, is to determine whether a view is *maintainable*, that is, guaranteed to have a unique new state, given an update to the base relations, an instance of the views, and an instance of a subset of the base relations. As a shorthand, such a view is said to be *self-maintainable* [1] in the given situation. A second question, the main question in traditional view-maintenance work, is how to bring the view up to date using only the given information. Together, these two questions define the *view self-maintenance* problem.

Previous work on view self-maintenance specific to a given situation ([TB88, GB95, Hu96]), however, only considered the special case where *no other materialized views* and *no base relations* are used to maintain a given view. We call this case *single-view self-maintenance*. Applying these methods to maintain a warehouse that contains several views, i.e., by maintaining the views separately from each other, often fails to maintain the warehouse when actually the views are self-maintainable collectively. The following example illustrates the need to use all the views to maintain a warehouse.

Example 1.1 Consider a data warehouse with two materialized views $V_1(X,Y,Z)$ and $V_2(Y,Z)$, each defined in terms of the base relations $R(X,Y)$, $S(Y,Z)$, and $T(Z)$ as follows:

```
CREATE VIEW V1(X,Y,Z) AS
SELECT R.X, R.Y, S.Z FROM R,S,T
WHERE R.Y = S.Y AND S.Z = T.Z

CREATE VIEW V2(Y,Z) AS SELECT * FROM S
```

[1] The terms "self-maintainable" and "self-maintenance" have been used in the literature with quite different meanings, depending on the amount of information available. We will more precisely define our notion of self-maintainability later.

That is, V_1 is the natural join of R, S, and T, while V_2 is a copy of S. Suppose we would like to maintain the warehouse in response to the insertion of tuple (a, b) into R, without using either R, S, or T.

First, consider the view instance where $V_1 = \{(a_1, b_1, c_1)\}$ and $V_2 = \{(b_1, c_1), (b, c_2)\}$. While we can infer the contents of S, since V_2 is just a copy of it, we cannot determine the contents of R and T exactly. In fact, it could be that $R = \{(a_1, b_1)\}$ and $T = \{(c_1)\}$, in which case view V_1 is not affected by the insertion of (a, b) into R. But it could also be that $R = \{(a_1, b_1)\}$ and $T = \{(c_1), (c_2)\}$, in which case (a, b, c_2) must be added to view V_1 to keep it consistent with the base relations. Thus, we cannot unambiguously maintain view V_1. V_1 is not self-maintainable under the insertion in this view instance.

Consider another instance where $V_1 = \{(a_1, b_1, c_1), (a_1, b_1, c_2)\}$ and $V_2 = \{(b_1, c_1), (b, c_2), (b_1, c_2)\}$. This time, however, we can infer enough about T to be able to precisely determine the effect of the insertion on V_1. In fact, to evaluate the effect of the insertion on view V_1, we look for tuples from S and T that join with the new tuple (a, b) from R. On the one hand, only one tuple from S qualifies: (b, c_2). On the other hand, to explain the presence of (a_1, b_1, c_2) in V_1, it must be case that T contains (c_2). Thus, we know exactly how to maintain V_1 without even looking at the base relations: add (a, b, c_2) to V_1. While V_1 is clearly self-maintainable under the insertion of (a, b) into R in this view instance, ([TB88, Hu96]) would fail to detect this situation because they attempt to maintain V_1 in isolation from V_2. If V_2 were not available, they would have correctly concluded that V_1 cannot be unambiguously maintained, since the following two base instances, while both consistent with V_1, derive different states of V_1 after the insertion of $r(a, b)$: $R = \{(a_1, b_1\}$, $S = \{(b_1, c_1), (b_1, c_2), (b, c_2)\}$, and $T = \{(c_1), (c_2)\}$ on the one hand, and $R = \{(a_1, b_1\}$, $S = \{(b_1, c_1), (b_1, c_2)\}$, and $T = \{(c_1), (c_2)\}$ on the other hand. But in light of V_2, the latter base instance is clearly not possible. ∎

Thus, to maximize the chance of maintaining the views successfully, we must take full advantage of all the information available, namely, not only the contents of the view to maintain, but also the contents of all the other views. Further, if a base update consists of a set of individual updates to several base relations, it is very important to consider the set of updates as a whole instead of considering each individual update separately. In fact, there are situations where the former approach succeeds to maintain a view but the latter fails. For instance, consider a view that computes whether or not two base relations (with identical schemas) have a tuple in common. Consider a situation where the view is empty and where the same tuple is to be inserted into *both* relations. Clearly, the new state of the view can only be "true". But if we consider the two insertions independently, we cannot unambiguously update the view.

Our work focuses on the *multiple-view self-maintenance* problem where again, the two critical questions are maintainability and maintenance of a view as a function of a given base update, a given instance of all the views, and a given instance of a particular subset of the base relations. Our work is mainly motivated by the desire to speed up view maintenance in WHIPS ([H*95]), a data warehousing system prototype developed at Stanford University that performs on-line update to views. Also, view self-maintenance is an effective approach to avoid asynchronous update anomalies ([Z*95]) inherent in any data warehousing environment, by limiting access to base relations that do no change, for instance. Finally, the self-maintenance approach is also important in any environment where source access is expensive for a variety of reasons: the required base data may be archived; the data source may be temporarily disconnected or even permanently destroyed.

The contributions of this paper are as follows:

- We solve the view self-maintenance problem under a wide variety of information that is available in practice: multiple views, partial access to the base relations, partial base copies, arbitrary mixes of base insertions and deletions.
- We provide new insight into the view self-maintainability problem by reducing it to a problem of query containment which can be solved by known algorithms.
- We show that for conjunctive-query views with no projection, not only efficient (polynomial-time) solutions to the view self-maintenance problem exist but also they can be generated at view-definition time in the form of SQL queries.

Related Work on Self-Maintenance

Self-maintenance generally refers to the problem of maintaining views without full use of the base relations. [GM95] gave an excellent classification of different notions of self-maintenance, based on what information is available for view maintenance. A major distinction is what we call *compile-time* vs. *run-time*. Even though both approaches share the goal of maintaining a view using only the information given (namely the instance of the views, the update instance, and perhaps the instance of some subset of the base relations), they differ in the way they guarantee a view can be maintained. In the compile-time approach, this guarantee is made independently of the instance of the views, the instance of the base relations, and the instance of some update type. "Compile-time" alludes to the fact that these instances are not known at compile-time, but only the view definitions and the update type. In the runtime approach, maintainability of a view is guaranteed on an instance basis: for a particular instance of the views, a particular instance of a subset of the base relations, and a particular update instance. Note that the runtime approach is the more aggressive one since it may succeed in maintaining a view where the compile-time approach may fail. In this paper, we take the runtime approach to self-maintenance, even though "runtime" is not explicitly mentioned.

Within the realm of runtime self-maintenance, we are not aware of any work that addresses the question of self-maintainability with respect to an instance of more than one view or base relation, or under an arbitrary mix of insertions and deletions to the base relations. [TB88] and more recently [GB95] gave self-maintainability conditions (they called conditions for *Autonomously Computable Updates*) for views that are SPJ queries with no self-joins and for updates that are either insertions or deletions to a single base relation. Their method cannot be extended easily to take advantage of multiple views or the base relations, or to handle updates that mix insertions with deletions. [Hu96] addressed the single-view strict self-maintenance problem and solved it more efficiently than [GB95], but only for views that are SPJ queries with no self-joins and for updates that are single insertions. Again, their method cannot be generalized easily. In the realm of compile-time self-maintenance, [GJM94] addressed the single-view self-maintenance for views defined as SPJ queries and under either insertions, deletions, or updates. More recently, [Q*96] solved a different but related problem, namely that of *making* a view self-maintainable by introducing a minimal set of auxiliary views to materialize. However, how to make more than one view self-maintainable was not addressed.

Outline of the Paper

In Section 2, we define the multiple-view self-maintenance problem. Sections 3 through 6 deal with strict self-maintenance. Sections 3 through 5 consider the special subclass of SPJ views with no projection. In Section 3, we give an algorithm that generates the view maintenance queries for a self-maintainable view. In Section 4, we reduce self-maintainability to a query containment problem. Based on this reduction, we give an algorithm that decides self-maintainability. Section 5 refines the reduction to what can solved in polynomial time: in particular, we give an algorithm that generates the self-maintainability query tests. In Section 6, we show how to extend the previous results to more complex views. In Section 7, we explain how generalized self-maintenance can be used to define a strategy for efficient warehouse maintenance. We show how to extend the results for strict self-maintenance to generalized self-maintenance. The paper concludes in Section 8. Throughout this paper, results are stated without their proof, which can be found in [Hu97].

2 Defining the Multiple-View Self-Maintenance Problem

Throughout this paper, the warehouse consists of materialized views $V_1, V_2, \ldots, V_m$ derived from base relations $R_1, R_2, \ldots, R_n$. This collection of base relations is referred to as database D. Each V_i is defined by a query Q_i over database D, written as $V_i = Q_i(D)$. A database D is said to be *consistent* with view V_i if $Q_i(D) = V_i$. We assume the existence of a database consistent with all the given views but whose content is not known a priori. We use U to denote an update to the base relations and $U(D)$ to denote the updated database. We model U as δR_1^-, δR_1^+, δR_2^-, δR_2^+, $\ldots$, δR_n^-, δR_n^+, where δR_j^- (resp. δR_j^+) is the set of tuples to be deleted from (resp. inserted to) relation R_j. Updates are assumed to be self-consistent, i.e., δR_j^- and δR_j^+ have no tuples in common for any j.

Strict Self-Maintenance

Given an instance of $V_1, \ldots, V_m$ and an update instance U, view V_k is said to be *self-maintainable* (under U, that is) if its new state (i.e., the new state of V_k that is consistent with the updated database) does not depend on the database, as long as the database (prior to the update) is consistent with *all the views*. We can formalize self-maintainability of V_k as:

$$(\forall D_1, D_2) \quad \bigwedge_i (Q_i(D_1) = V_i) \wedge \bigwedge_i (Q_i(D_2) = V_i) \Rightarrow Q_k(U(D_1)) = Q_k(U(D_2)) \qquad (1)$$

Thus, in strict self-maintenance, self-maintainability is a function of U and $V_1, \ldots, V_m$ (it is also a function of the view definitions Q_i, but that is understood). Note the requirement that D be consistent with all the views, and not just the view to maintain as in single-view self-maintainability.

Only when V_k is self-maintainable does it make sense to maintain it, and a maintenance expression is a function of U and $V_1, \ldots, V_m$ (not just V_k as in single-view self-maintenance).

Generalized Self-Maintenance

We generalize strict self-maintenance by also allowing access to some of the base relations. Thus, given an instance of $V_1, \ldots, V_m$, an update instance U, and the instance of a subset S of the base relations, view V_k is said to be *self-maintainable* (under U and with respect to S, that is) if its new state does not depend on the database, as long as the database (prior to the update) is consistent with all the views and with *the given base relation instances in S*.

Thus, in generalized self-maintenance, both maintainability and maintenance expression are a function of U, $V_1, \ldots, V_m$, and the base relations in S.

Notation

In this work, we assume a relational database framework in which views are defined by relational queries over base relations. Set semantics is also assumed. Thus, the answer to a query is a set of tuples. We will use the notation of Datalog [Ull89] for all the queries involved in our algorithms. This choice is by convenience, even though any other relational languages could be used. Thus, the view definition for view V_1 from Example 1.1 is written as a Datalog query Q_1 with the single rule:

$$v_1(X, Y, Z) \coloneq r(X, Y) \ \& \ s(Y, Z) \ \& \ t(Z)$$

where $v_1(X, Y, Z)$ is called the rule's *head* and $r(X, Y)$, $s(Y, Z)$, and $t(Z)$ are the rule's *subgoals*. By convention, relation names are written in upper case (e.g., V_1, S, and δR^-) and their predicate in lower case (e.g., v_1, s, and δr^-). The *extension* of a predicate is the instance of the relation for the predicate. In general, a predicate is called an *IDB predicate* if it appears in the head of some rule, an *EDB predicate* otherwise. A particular IDB predicate that is used to return the answers to the query is called the *query predicate*. Thus, in query Q_1, predicates r, s, and t are the EDB predicates, and v_1 the query predicate. Q_1 is an example of a Datalog query with only one rule whose body contains only EDB subgoals. Such a query is called a *conjunctive query* or an SPJ query with only equality comparisons.

For the most part (except Section 6), the queries Q_i that define the views in the warehouse (in terms of the base relations R_j) are assumed to be conjunctive. In rule notation, we write Q_i as:

$$H_i \coloneq G_{i1} \ \& \ \ldots \ \& \ G_{in_i}$$

where the head H_i uses predicate v_i for view V_i and each subgoal G_{ij} uses predicate r_l for some relation among $R_1, R_2, \ldots, R_n$. Constant symbols may appear anywhere in a rule. We also assume that the variables in Q_i's body all appear in Q_i's head. Such a query is said to *have no projection*.

There will be occasions where Datalog queries more complex than conjunctive queries are used, albeit without recursion. *A union of conjunctive queries* is like a conjunctive query except that we have several rules defining the query predicate. We will use H :- $A \mid B$ as a shorthand for the two rules H :- A and H :- B. Finally, more complex queries may use negation and arithmetic comparisons other than equality.

Query Containment

The main technique used in solving the self-maintainability problem (1) is based on showing that it can be reduced to a particular implication problem known in the literature as the query containment (abbrev. QC) problem [Ull89]. Given two Datalog queries P and Q using EDB relations $E_1, \ldots, E_n$ as input, we say that P is *contained in* Q (denoted $P \subseteq Q$) if the answer to P is a subset of the answer to Q, for every instance of $E_1, \ldots, E_n$. In our work, we will encounter a variation of the containment problem where the instance of some of the input EDB relations is also given. We shall continue to call this variation "query containment", even though "instance-specific QC" would have been more accurate. Given two queries P and Q using EDB relations $E_1, \ldots, E_n, F_1, \ldots, F_m$ as input, and given an instance of $F_1, \ldots, F_m$, we say that $P \subseteq Q$ (with respect to the instance of $F_1, \ldots, F_m$, that is) if the answer to P is a subset of the answer to Q for all instances of $E_1, \ldots, E_n$. The EDB predicates f_i, whose extension is given, are called *constant predicates*. The EDB predicates e_i are called *variable predicates*. We can always reformulate an instance-specific QC problem to a QC problem by eliminating any constant predicate f as follows: replace any subgoal $\neg f(\bar{X})$ with $\bigwedge_{\bar{x}}(\bar{X} \neq \bar{x})$ and any subgoal $f(\bar{X})$ with $\bigvee_{\bar{x}}(\bar{X} = \bar{x})$, where $\bar{x}$ ranges over the tuples in f's extension.

3 Generating Queries to Maintain the Views

In this section, we address the question of how to bring a view up to date *if the view is known to be self-maintainable.* Note that if a view is not self-maintainable, there is no unambiguous way to correctly maintain the view without using additional information (see Section 7 on using additional base relations).

We use a very simple idea. If a view is self-maintainable, we do not need to know what the actual database really is to maintain the view since we can use any database that is consistent with all the views to propagate the update to the view. But how can we find such a database? The answer lies in the canonical database. Note that the canonical database is defined relative to an instance of the views.

Definition 3.1 (Canonical database): Let $V_1, \ldots, V_m$ be given views and for $i = 1, \ldots, m$, let Q_i be the conjunctive query (with no projection) over relations $R_1, \ldots, R_n$ that defines V_i. The canonical database, denoted $\hat{D}$, consists of all the tuples obtained as follows: for each view V_i, every tuple in V_i that matches Q_i's head provides a substitution that grounds all the atoms in Q_i's body; include all these ground atoms in $\hat{D}$. ∎

Example 3.1 Consider views V_1 and V_2 defined by:

$$v_1(X,Y,Z) \text{ :- } r(X,Y) \ \& \ s(Y,Z) \ \& \ t(Z)$$
$$v_2(Y,Z) \text{ :- } s(Y,Z)$$

Suppose $V_1 = \{(a_1,b_1,c_1),(a_1,b_1,c_2)\}$ and $V_2 = \{(b_1,c_1),(b,c_2),(b_1,c_2)\}$. The canonical database $\hat{D}$ in this view instance consists of $R = \{(a_1,b_1)\}$, $S = \{(b_1,c_1),(b_1,c_2),(b,c_2)\}$, and $T = \{(c_1),(c_2)\}$. ∎

Intuitively, we are trying to minimally reconstruct the base relations from all the given views. When each Q_i has no projection, there is a unique minimal reconstruction, which is the canonical database $\hat{D}$. The following lemma states the key property of $\hat{D}$ that allows us to use it to maintain the views.

Lemma 3.1 *The canonical database is consistent with all the views.* ∎

The following example illustrates how to maintain the views using the canonical database.

Example 3.2 Continuing from Example 3.1, now consider inserting (a,b) to relation R. If V_1 is self-maintainable under the insertion (and with respect to the given view instance), we know we can obtain the same result for the new state of V_1 no matter which database we use to propagate the insertion and that is consistent with the views. We can use $\hat{D}$ in particular. So to compute the tuples gained by V_1, we simply join $r(a,b)$ with $S = \{(b_1,c_1),(b_1,c_2),(b,c_2)\}$ and $T = \{(c_1),(c_2)\}$ to obtain (a,b,c_2). ∎

The following theorem formalizes the use of $\hat{D}$ to maintain the views.

Theorem 3.1 *Let $V_1, \ldots, V_m$ be given views and for $i = 1, \ldots, m$, let Q_i be the conjunctive query (with no projection) over some database D that defines V_i. Let U be a given update to D. If view V_k is self-maintainable under U, then the new state for V_k is $Q_k(U(\hat{D}))$, where $\hat{D}$ is the canonical database.* ∎

Theorem 3.1 provides us with the following algorithm that computes the view maintenance expressions.

Algorithm 3.1 (**Generate Maintenance Queries**)

Input: $Q_1, \ldots, Q_m$, where each Q_i is a conjunctive query (with no projection) that defines v_i using $r_1, \ldots, r_n$ as input.

Output: Queries for incrementally maintaining V_k, using $v_1, \ldots, v_m$, $\delta r_1^-, \delta r_1^+, \ldots, \delta r_n^-, \delta r_n^+$ as input.

Method:

1. For $i = 1, \ldots, m$ and $j = 1, \ldots, n_i$, generate rule (A_{ij}) : $\hat{G}_{ij}$:- H_i, where H_i is the head of Q_i and $\hat{G}_{ij}$ is the subgoal G_{ij} in Q_i's body whose predicate r_l is replaced by predicate $\hat{r}_l$. These rules define the predicates $\hat{r}_1$,...,$\hat{r}_n$ for the canonical database $\hat{D}$.
2. Generate queries that incrementally maintain V_k, using predicates v_k, $r_1, \ldots, r_n$, $\delta r_1^-, \delta r_1^+$, $\ldots, \delta r_n^-, \delta r_n^+$ as input. Call this set of rules M.
3. Let $\hat{M}$ be obtained from M where every occurrence of r_j is replaced by $\hat{r}_j$, for $j = 1, \ldots, n$.
4. Return $\hat{M} \cup \{(A_{ij}), i = 1, \ldots, m, j = 1, \ldots n_i\}$.

∎

Step 1 in Algorithm 3.1 essentially computes the canonical database $\hat{D}$. Step 2 generates queries that incrementally maintain view V_k, i.e., that update V_k to the new state $Q_k(U(D))$ using V_k and all the base relations R_i's (the instance of these base relations is actually taken from the canonical database, which is the purpose of Step 3). Many known algorithms exist in the view-maintenance literature ([Kuc91, SJ96]) that can generate queries for incrementally maintaining a view using both the view and all the base relations. Using for instance [SJ96] in Step 2, Algorithm 3.1 generates the queries that compute the required insertions to and deletions from a view, in time linear in the size of the view definitions. The size of these queries is also linear. In practice, if these queries are optimized, we may not need to actually construct the entire canonical database as Step 1 would suggest.

Example 3.3 Consider the view definitions for V_1 and V_2 from Example 3.1 and consider the insertion of $r(a,b)$. Let δv_1^+ be the predicate for the set of net insertions to V_1. Algorithm 3.1 generates the following query for δv_1^+:

$$\hat{s}(Y,Z) \text{ :- } v_1(X,Y,Z) \mid v_2(Y,Z)$$
$$\hat{t}(Z) \text{ :- } v_1(X,Y,Z)$$
$$\delta v_1^+(a,b,Z) \text{ :- } \hat{s}(b,Z) \ \& \ \hat{t}(Z) \ \& \ \neg v_1(a,b,Z)$$

which can further be simplified into

$$\delta v_1^+(a,b,Z) \text{ :- } v_1(X,b,Z) \ \& \ v_1(X',Y',Z) \ \& \ \neg v_1(a,b,Z)$$
$$\delta v_1^+(a,b,Z) \text{ :- } v_2(b,Z) \ \& \ v_1(X',Y',Z) \ \& \ \neg v_1(a,b,Z)$$

∎

If a view is not self-maintainable, applying the maintenance queries generated by Algorithm 3.1 may incorrectly update the view. Thus, before applying them to maintain a view, it is important to make sure the view is self-maintainable. The next section provides a decision method.

4 Deciding Self-Maintainability

To solve the self-maintainability problem, we reduce it to a problem of query containment whose solution exists in the literature. This reduction is based on the existence of a database we know how to build out of the contents of the views and that is consistent with all the views. Again, the canonical database, defined in Section 3, is such a database. The following example illustrates this reduction.

Example 4.1 Consider views V_1 and V_2 as defined in Example 3.1 and consider the insertion of $r(a,b)$. To determine whether view V_1 is self-maintainable under the insertion, the main idea is to compare the effect of the insertion on V_1 when using $\hat{D}$ with the effect when using any database consistent with both V_1 and V_2. First consider the view instance where $V_1 = \{(a_1,b_1,c_1),(a_1,b_1,c_2)\}$ and $V_2 = \{(b_1,c_1),(b,c_2),(b_1,c_2)\}$. V_1 is self-maintainable in this view instance because inserting $r(a,b)$ into any consistent database exactly causes (a,b,c_2) to be added to V_1, which is precisely the same effect on V_1 as the insertion into $\hat{D}$ (as determined in Example 3.2). Now consider another view instance where $V_1 = \{(a_1,b_1,c_1)\}$ and $V_2 = \{(b_1,c_1),(b,c_2)\}$. $\hat{D}$ in this case consists of $R = \{(a_1,b_1)\}$, $S = \{(b_1,c_1),(b,c_2)\}$, and $T = \{(c_1)\}$. V_1 is not self-maintainable in this view instance since while the insertion into $\hat{D}$ has no effect on V_1, there is a consistent database (namely $R = \{(a_1,b_1)\}$, $S = \{(b_1,c_1),(b,c_2)\}$, and $T = \{(c_1),(c_2)\}$) where the insertion of $r(a,b)$ causes V_1 to gain (a,b,c_2). ∎

Example 4.1 suggests that self-maintainability of a view under a given update can be characterized completely as the following implication problem: for every database D, if D is consistent with the views before the update, then D derives the same view as $\hat{D}$ after the update. This implication has the form of a query containment problem where the queries to compare are boolean queries. This reduction of self-maintainability to query containment is formalized in the following theorem.

Theorem 4.1 *Let $V_1, \ldots, V_m$ be given views and for $i = 1, \ldots m$, let Q_i be the conjunctive query (with no projection) over some database D that defines view V_i. Let U be an update on D. Then V_k is self-maintainable under U if and only if $Q_k(U(D)) \neq Q_k(U(\hat{D}))$ implies $\bigvee_i Q_i(D) \neq V_i$, for every database D ($\hat{D}$ is the canonical database).* ∎

Theorem 4.1 allows us to solve the self-maintainability problem using known techniques for deciding whether a query is contained in another query. In the following algorithm, Algorithm 4.1, the two queries to compare are *DIFF* and *INCON*. Algorithm 4.1 first generates rules for these queries, as summarized in Table 1. They relate to Theorem 4.1 as follows: rules (A_{ij}) compute $\hat{D}$; (B_i) and (C_{ij}) say that $Q_i(D) \neq V_i$; (D_j) defines predicate r'_j for relation R_j in $U(D)$; (F_j) defines predicate $\hat{r}'_j$ for relation R_j in $U(\hat{D})$; (H_k) defines predicate $\hat{v}'_k$ for $Q_k(U(\hat{D}))$, the new state of view V_k that derives from $\hat{D}$ after the update; (I_k) says there is some tuple in $Q_k(U(D))$ but not in $Q_k(U(\hat{D}))$; and (J_{kj}) says there is some tuple in $Q_k(U(\hat{D}))$ but not in $Q_k(U(D))$.

Algorithm 4.1 **(Decide Self-Maintainability)**

Input: Views $V_1, \ldots, V_m$, and for $i = 1, \ldots, m$, Q_i, a conjunctive query (with no projection) that defines v_i using $r_1, \ldots, r_n$ as input, and an update $U = \delta R_1^-, \delta R_1^+, \ldots, \delta R_n^-, \delta R_n^+$.

Output: A decision whether V_k is self-maintainable under U in the given view instance $V_1, \ldots, V_m$.

Method:

1. Generate rules for queries *DIFF* and *INCON* as shown in Table 1. Both queries use the 0-ary predicate *panic* for their query predicate.
2. Return the decision whether $DIFF \subseteq INCON$ for every instance of $R_1, \ldots, R_n$.

∎

Each of *DIFF* and *INCON* is a Datalog query that can be transformed (after expanding rules (D_j) and eliminating all constant predicates) to a union of conjunctive queries. These queries involve negation and $\neq$ comparisons, and use $r_1, \ldots, r_n$ as input. The [LS93] algorithm can decide containment of unions of such queries in time exponential in the size of the views. As long as we use the reduction from Theorem 4.1, this complexity is probably the best that can be achieved, since the queries to compare use negation and use a number of constant symbols the size of the views. In the next section, we will give a more refined reduction that eliminates the use of negation, thus allowing more efficient containment checking algorithms to be used and, most importantly, self-maintainability to be decided in polynomial time.

5 Generating Queries to Test View Self-Maintainability

In Section 4, we reduced self-maintainability to testing containment of queries which involve negation. In this section, we show a more refined reduction that results in simpler queries to compare. Previously, self-maintainability of V_k under update D essentially reduces to checking whether after update, all databases D that are consistent with the views derive the same relation as $Q_k(U(D))$. The key observation here is that instead of checking all databases, we only need to check those that contain the canonical database $\hat{D}$. The simple reason is that any database that does not contain $\hat{D}$ cannot be consistent with the views, as formalized in the following lemma.

Lemma 5.1 *Let $V_1, \ldots, V_m$ be given views, and for each $i = 1, \ldots, m$, let Q_i be the conjunctive query (with no projection) that defines V_i over some database D. If a database D is consistent with all the views, then $D \supseteq \hat{D}$, where $\hat{D}$ is the canonical database.* ∎

The following example informally explains how checking self-maintainability can be improved.

Example 5.1 Consider the same views as defined in Example 4.1, the insertion of $r(a, b)$, and the view instance where $V_1 = \{(a_1, b_1, c_1), (a_1, b_1, c_2)\}$ and $V_2 = \{(b_1, c_1), (b, c_2), (b_1, c_2)\}$. The new view V'_1 that results from updating the canonical database has the additional tuple (a, b, c_2) besides those already in V_1. Previously, in order to determine if V_1 is self-maintainable under the insertion, we consider *every* database D and check whether D exactly derives both V_1 and V_2 before the insertion and whether D exactly derives V'_1 after. Improving upon the previous method, instead of considering all databases, now we consider only those that contain $\hat{D}$. Not only fewer databases need to be considered, their checks become considerably simpler since a database that contains $\hat{D}$ cannot derive less tuples than V_1 and V_2 before the update and less tuples than V'_1 after the update. ∎

The new reduction is formalized in the following theorem, where $D \cup \hat{D}$ represents an arbitrary database that contains $\hat{D}$. The use of "set union" makes sense since a database is a set of tuples.

Theorem 5.1 *Let $V_1, \ldots, V_m$ be given views, and for $i = 1, \ldots, m$, let Q_i be the conjunctive query (with no projection) that defines V_i. Let U be an update to D. Then V_k is self-maintainable under U if and only if $Q_k(U(D \cup \hat{D})) \not\subseteq Q_k(U(\hat{D}))$ implies $\bigvee_i Q_i(D \cup \hat{D}) \not\subseteq V_i$, for every database D, where $\hat{D}$ is the canonical*

Table 1: Rules generated for queries whose containment decides self-maintainability of V_k.

Rules	Range	*DIFF*	*INCON*
(A_{ij})	$i=1,\ldots,m,\ j=1,\ldots,n_i$	$\hat{G}_{ij}$:- H_i	
(B_i)	$i=1,\ldots,m$		*panic* :- G_{i1} & ... & G_{in_i} & $\neg H_i$
(C_{ij})	$i=1,\ldots,m,\ j=1,\ldots,n_i$		*panic* :- H_i & $\neg G_{ij}$
(D_j)	$j=1,\ldots,n$	r'_j :- r_j & $\neg\delta r_j^-$ | δr_j^+	
(F_j)	$j=1,\ldots,n$	$\hat{r}'_j$:- $\hat{r}_j$ & $\neg\delta r_j^-$ | δr_j^+	
(H_k)		$\hat{H}'_k$:- $\hat{G}'_{k1}$ & ... & $\hat{G}'_{kn_k}$	
(I_k)		*panic* :- G'_{k1} & ... & G'_{kn_k} & $\neg\hat{H}'_k$	
(J_{kj})	$j=1,\ldots,n_k$	*panic* :- $\hat{H}'_k$ & $\neg G'_{kj}$	

H_i :- G_{i1} & ... & G_{in_i} is the rule defining V_i; $\hat{G}_{ij}$ is the subgoal G_{ij} whose predicate r_l is replaced by predicate $\hat{r}_l$; $\hat{H}'_k$ is the head H_k whose v_k is replaced by $\hat{v}'_k$; $\hat{G}'_{kj}$ is G_{kj} whose r_l is replaced by $\hat{r}'_l$; G'_{kj} is G_{kj} whose r_l is replaced by r'_l.

database. Furthermore, the boolean queries in the containment equation can be expanded to unions of conjunctive queries with negation that only applies to constant EDB predicates. ∎

Theorem 5.1 improves on Theorem 4.1 in eliminating the uses of $\not\supseteq$ which were the main source of exponential complexity in Algorithm 4.1: $\not\supseteq$ introduced negation that applies to the variable EDB predicates $r_1,\ldots,r_n$. While negation still remains, it only applies to constant EDB predicates which can be eliminated. In other words, the queries to compare are essentially unions of conjunctive queries with only $\neq$ comparisons, which are much simpler to deal with. In the following algorithm, the rules generated are slightly different than those generated in Algorithm 4.1. These rules are summarized in Table 2. Most notably, rules (C_{ij}) and (J_{kj}) from Table 1, which introduced negated subgoals with some predicate r_l, have been eliminated. The rules in Table 2 relate to Theorem 5.1 as follows: rules (A_{ij}) compute $\hat{D}$; (K_j) defines predicate r''_j for relation R_j in $D \cup \hat{D}$; (B'_i) represent the fact that $Q_i(D\cup\hat{D}) \not\subseteq V_i$; (D'_j) defines predicate r'_j for relation R_j in $U(D\cup\hat{D})$; (F_j) defines predicate $\hat{r}'_j$ for relation R_j in $U(\hat{D})$; (H_k) defines predicate $\hat{v}'_k$ for $Q_k(U(\hat{D}))$, the new state of view V_k that derives from $\hat{D}$ after the update; (I_k) expresses the fact that $Q_k(U(D\cup\hat{D})) \neq Q_k(U(\hat{D}))$.

Algorithm 5.1 (Generate Self-Maintainability Test)

Input: $Q_1,\ldots,Q_m$, where each Q_i is a conjunctive query (with no projection) that defines v_i using $r_1,\ldots,r_n$ as input.

Output: A query that decides whether V_k is self-maintainable under U, using predicates $v_1,\ldots,v_m$ and $\delta r_1^-,\delta r_1^+,\ldots,\delta r_n^-,\delta r_n^+$ as input.

Method:

1. Generate rules for queries *DIFF* and *INCON* as shown in Table 2. Both queries use the 0-ary predicate *panic* for their query predicate.
2. Generate a query *TEST* that decides whether *DIFF* $\subseteq$ *INCON*. Return *TEST*.

∎

In Algorithm 5.1, instead of generating a test in Step 2, we could have solved *DIFF* $\subseteq$ *INCON* directly by using known algorithms in the literature ([G*94, Klu88]) for deciding containment of unions of conjunctive queries with arithmetic comparisons. Even though these algorithms are more efficient than those for deciding containment of queries with negation, a naive way of applying them would require eliminating all constant EDB predicates. Unfortunately, the resulting complexity would still have been exponential in the size of the views, because the expanded queries have exponential size.

The next theorem is very important since it gives us a *polynomial-time* solution to the self-maintainability problem. The key to solve *DIFF* $\subseteq$ *INCON* without eliminating the constant EDB predicates is to translate it into a logical expression that involves these constant predicates rather than their extension. This expression is then rewritten as a query, which can be evaluated in time polynomial in the size of their extension.

Theorem 5.2 *Let P and Q be two conjunctive queries using both constant and variable EDB predicates as input. Negation may be used but only applies to the constant predicates. Each rule is assumed to have a negated subgoal that uses all the body variables. Furthermore each rule in Q has exactly one negated subgoal. Then there is a boolean query that decides whether $P \subseteq Q$ for every extension of the variable predicates, and that uses constant predicates as input.* ∎

Based on Theorem 5.2, we develop an algorithm which can be used in Step 2 of Algorithm 5.1 to translate the containment question to a test in query form.

Table 2: Rules generated for the queries to compare in the new reduction.

Rules	Range	*DIFF*	*INCON*
(A_{ij})	$i = 1, \ldots, m$, $j = 1, \ldots, n_i$	$\hat{G}_{ij}$:- H_i	$\hat{G}_{ij}$:- H_i
(K_j)	$j = 1, \ldots, n$	r''_j :- $r_j \mid \hat{r}_j$	r''_j :- $r_j \mid \hat{r}_j$
(B'_i)	$i = 1, \ldots, m$		$panic$:- G''_{i1} & ... & G''_{in_i} & $\neg H_i$
(D'_j)	$j = 1, \ldots, n$	r'_j :- r''_j & $\neg \delta r^-_j \mid \delta r^+_j$	
(F_j)	$j = 1, \ldots, n$	$\hat{r}'_j$:- $\hat{r}_j$ & $\neg \delta r^-_j \mid \delta r^+_j$	
(H_k)		$\hat{H}'_k$:- $\hat{G}'_{k1}$ & ... & $\hat{G}'_{kn_k}$	
(I_k)		$panic$:- G'_{k1} & ... & G'_{kn_k} & $\neg \hat{H}'_k$	

Notation: $\hat{G}_{ij}$ is G_{ij} whose predicate r_l is replaced by $\hat{r}_l$; G''_{ij} is G_{ij} whose r_l is replaced by r''_l; $\hat{H}'_k$ is H_k whose v_k is replaced by $\hat{v}'_k$; $\hat{G}'_{kj}$ is G_{kj} whose r_l is replaced by $\hat{r}'_l$; G'_{kj} is G_{kj} whose r_l is replaced by r'_l.

Due to space limitation, this algorithm is not shown here. It can be found in [Hu97]. Essentially, given two queries, the algorithm first generates a logical expression that captures the condition for containment for the queries. This logical expression, while not finitely evaluable, is then rewritten as another logical expression that can be finitely evaluated, or equivalently, as a query in safe, nonrecursive Datalog with negation and $\neq$ comparisons. Such a query test can be implemented easily in SQL.

Thus, in contrast to Algorithm 4.1 which decides self-maintainability at runtime, Algorithm 5.1 translates, at view-definition time, self-maintainability to a query test that can be evaluated against the views and the update instance at runtime. As such, not only can we test self-maintainability in polynomial time, but also we can optimize and compile the test more effectively than a test in procedural form such as Algorithm 4.1. The running time of Algorithm 5.1 and the size of the query test it generates do not depend on the instance of the views and update. They are exponential in the size of the view definitions. This complexity is not surprising, in light of the NP-completeness of checking query containment [CM77]. While the complexity of test generation is not as critical as the complexity of test execution, the availability of good query optimization techniques can help simplify the tests and further improve their execution speed.

Example 5.2 Consider the definition of views V_1 and V_2 from Example 3.1 and consider the problem of testing self-maintainability of V_1 under the insertion of $r(a, b)$. Algorithm 5.1 generates a test which simplifies to the following query (using the 0-ary predicate *maintainable* as the query predicate):

$$
\begin{aligned}
&p(Z) \text{ :- } v_1(X, Y, Z)\\
&q(Z) \text{ :- } v_2(Y, Z) \;\&\; v_1(X, Y, Z')\\
&\mathit{depend} \text{ :- } v_2(b, Z) \;\&\; \neg p(Z) \;\&\; \neg q(Z)\\
&\mathit{maintainable} \text{ :- } \neg \mathit{depend}
\end{aligned}
$$

or equivalently in SQL:

```
NOT EXISTS
(SELECT * FROM V2 WHERE V2.Y = b
 AND NOT EXISTS (SELECT * FROM V1
                 WHERE V1.Z = V2.Z)
 AND NOT EXISTS (SELECT * FROM V2 as V3, V1
                 WHERE V3.Z = V2.Z
                 AND V2'.Y = V1.Y))
```

6 Maintaining Other More Complex Views

The techniques developed in the previous sections for maintaining a special class of views (defined by conjunctive queries without projection) have, in fact, much wider applicability. In this section, we show how to extend them to larger classes of views that are important in practice: conjunctive queries with projection, unions of conjunctive queries, and partial copies. Further extensions are possible (e.g., for queries with arithmetic comparisons and queries over base relations constrained by dependencies) but are not described here due to space limitation.

6.1 Views with Projections

Consider views defined by conjunctive queries where some variables used in a rule's body do not appear in the rule's head. A view where some attributes have been projected out looses information, and from an instance of the view, there is no unique way of "reconstructing" a minimal database. The notion of canonical database from Section 3 must be revised to capture this non-uniqueness. So, we redefine our new canonical database $\hat{D}$ as follows: for each V_i, since each tuple in V_i that matches Q_i's head provides a substitution for only some of the variables in Q_i's body, this substitution is extended to the remaining variables by binding each of them to a *new symbol*; the ground atoms obtained after making this extended substitution into Q_i's body are included in $\hat{D}$.

A tuple in $\hat{D}$ that contains a new symbol represents a fact involving some object whose value is not known.

This value could be any of the known constants from the instance of the views or the update instance, or could be some constant not in any of those instances. Thus, if we consider all the symbol mappings h that map each of the new symbols to either one of themselves or a known constant, then $\hat{D}$ represents not a single database but a class of possible databases, each of which is obtained by applying some substitution h to $\hat{D}$. We can show that there is always a mapping h such that database $h(\hat{D})$ is consistent with all the views. Such a mapping is said to be *consistent*. The following example illustrates the non-uniqueness of minimal databases due to projections in views.

Example 6.1 Consider the view definition $v(X, Z) :- s(X, Y) \ \& \ s(Y, Z)$ where Y has been projected out. Consider the instance $V = \{(d, c)\}$ and the insertion of (a, b) to S. The canonical database $\hat{D}$, obtained as $S = \{(d, y), (y, c)\}$ where y is a new symbol, actually can be interpreted in five possible ways (by mapping y to either y, a, b, d, or c): $S = \{(d, y), (y, c)\}$, $S = \{(d, a), (a, c)\}$, $S = \{(d, b), (b, c)\}$, $S = \{(d, d), (d, c)\}$, or $S = \{(d, c), (c, c)\}$. The last two databases are not consistent with V since they respectively derive tuples (d, d) and (c, c) which are not in the view. Among the remaining consistent databases, after the insertion, the second one derives tuple (d, b) not derived by the first one. Thus, view V is not self-maintainable under the insertion of (a, b) to S. ∎

Once a consistent database $h(\hat{D})$ is found, we can apply the same idea as in Section 3 to maintain a view if the view is self-maintainable: propagate an update to the view using $h(\hat{D})$ as the actual database. Similarly, the self-maintainability question is settled by extending the reduction from Section 5 to take into account the non-uniqueness of a minimal consistent database. We can now informally state a theorem analogous to Theorem 5.1: V_k is self-maintainable under U if and only if (1) the new state of V_k does not depend on which consistent database $h(\hat{D})$ we use as the actual database, and (2) for each consistent $h(\hat{D})$, every database that contains $h(\hat{D})$ and that is consistent with all the views must derive the same state for V_k after the update, as $h(\hat{D})$.

To sum it up, we obtain algorithms similar to Algorithms 3.1 and 5.1 except that they use a minimal database $h(\hat{D})$ that is consistent with all the views, instead of just $\hat{D}$. While the use of projection in views seems to make the problem considerably harder since the number of consistent mappings h can be exponential in the worst case, results from [Hu96] suggest that it does not have to be so. For example, [Hu96] showed that even with projection, self-maintainability of a single conjunctive-query view with no self-join can be efficiently decided with a simple query. Thus, an important future direction is to further refine our techniques and identify restrictions on the view that allow the problem to be solved efficiently.

6.2 Union Views

We now consider views that are unions of conjunctive queries which, for simplicity of discussion, do not have projections. Unions of conjunctive queries with projections can be treated easily by combining the techniques from both this subsection and the previous one. For a view V defined by more than one rule (say by $mult(v)$ many rules), the presence of each tuple in the view can be explained by more than one set of facts ($mult(v)$ many sets to be exact). Thus, given an instance of $V_1, \ldots, V_m$, the number d of possible minimal databases that can be consistent with the views is $mult(v_1)^{size(V_1)} \times \ldots \times mult(v_m)^{size(V_m)}$. Thus, like projections, unions introduce non-uniqueness of a minimal database consistent with the views. Among all such possible databases, denoted $D_1, \ldots, D_d$, we can show there is always one that is consistent with all the views. The following example illustrates the non-uniqueness of minimal databases due to unions in views.

Example 6.2 Consider the view definitions $v_1(X, Y, Z) :- r(X, Y) \ \& \ s(Y, Z) \mid r(X, Y) \ \& \ s(Y, Z)$ and $v_2(X, Y) :- r(X, Y) \mid t(X, Y)$. Consider the instances $V_1 = \{(a, b, c)\}$ and $V_2 = \{(a, b)\}$. There are four minimal databases: $R = \{(a, b)\}$, $S = \{(b, c)\}$; $R = \{(a, b)\}$, $S = \{(b, c)\}$, $T = \{(a, b)\}$; $R = \{(a, b)\}$, $S = \{(a, b)\}$, $T = \{(b, c)\}$; and $S = \{(a, b)\}$, $T = \{(a, b), (b, c)\}$. Among these minimal databases, only the first two are consistent with both V_1 and V_2. ∎

Using those D_i's that are consistent with all the views, the self-maintenance problem can be solved in a way that parallels the case of views with projection: for instance, to check self-maintainability of a view V_k, make sure that (1) all those D_i's that are consistent with the views derive the same state for V_k after the update, and (2) for each D_i that is consistent with the views, every database that contains D_i and that is consistent with all the views must derive the same state for V_k after the update, as D_i. Thus, we obtain algorithms similar to Algorithms 3.1 and 5.1 except that they use a database among $D_1, \ldots, D_d$ that is consistent with all the views , instead of just $\hat{D}$.

6.3 Partial Copies

Suppose a log of the most recent updates on a base relation R is kept at the warehouse. Unlike a full copy of R stored at the warehouse, the most the log can tell us about R is that R must include certain tuples (say represented by set R^+) but exclude others (say R^-). We call this information about relation R a *partial copy*. Thus, given R^+ and R^-, the question is how to take full advantage of the additional information in view self-maintenance. A solution can be obtained by simply revising both Definition 3.1 of the canonical database to additionally include R^+ and Theorem 5.1 to use $R \cap R^- \neq \emptyset$ as another inconsistency condition on the right hand of the implication. Algorithms similar to Algorithms 3.1 and 5.1 can be obtained the obvious way. Furthermore, the solutions can be extended in a straightforward manner to logs that contain the most recent updates on more than one base relation. View self-maintenance with partial copies has the same complexity as with regular materialized views.

7 Maintaining Views with Partial Access to the Base Relations

As stated, the main motivation behind self-maintenance is in maintaining a warehouse efficiently. In strict self-maintenance, we attempt to maintain the views using information that can be obtained strictly locally from the warehouse, namely the materialized views and the update. When a view is not self-maintainable in the strict sense, we can only fall back to the "normal" but expensive maintenance mode with unrestricted access to the base relations, as depicted in Figure 7(a).

However, instead of switching to the normal maintenance mode immediately, we may be able to use some (but not necessarily all) of the base relations to successfully maintain the view. In fact, there are many cases where a view is not self-maintainable (in the strict sense) but can be maintained using some of the base relations. Thus, a more refined strategy based on generalized self-maintenance can be used instead. Figure 7(b) illustrates this strategy. Note that in Figure 7(b), the choice of which base relations to use next is left open. How to make the optimal choice is an important area for future research.

In the following, we show how to solve the generalized self-maintenance problem. There is a close resemblance between *allowing access to a base relation* and *having a copy of the base relation materialized at the warehouse*. In fact, if we assume that:

- The materialized views are *simultaneously* updated, that is, the required updates to each view are determined prior to updating any view, and
- The base relations are accessed in a state that reflects update U but no other later updates (assuming that the warehouse received updates in the order they are applied to the database),

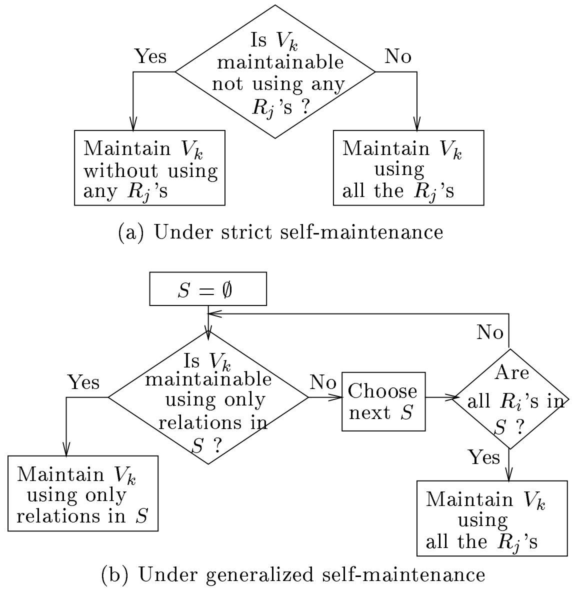

Figure 2: Strategies for Efficient Maintenance.

then, the generalized self-maintenance problem can be treated as a strict self-maintenance problem where *a copy of the given base relations is available at the warehouse*, with the exception that the actual base relations and the copy only differ by the update.

Example 7.1 Consider a warehouse with a single view V_1 defined as in Example 3.1. Assume we can access base relation S but not R or T. Consider an update with δR^-, δR^+, δS^-, δS^+, δT^-, and δT^+. The maintenance expression and maintainability test for this generalized self-maintenance problem can be obtained as follows. Consider the self-maintenance problem with both view V_1 and a view V_2 that is a copy of S. The solutions to this problem use predicates v_1 and v_2 as input. Replace every occurrence of v_2 with a new predicate s' defined by the following rules:

$$s'(Y,Z) \quad :- \quad s(Y,Z) \ \& \ \neg\delta s^+(Y,Z) \mid \delta s^-(Y,Z)$$

Predicate s' represents the state of relation S prior to the given update. ∎

Thus, results for the strict self-maintenance problem can be carried over by simply replacing every reference to the "copy" of a base relation by a reference to its "before image". In practice, allowing access to a base relation when maintaining a materialized view must be handled carefully. When a base relation is asynchronously updated by the source, it may be read by the warehouse in a different state than what is assumed by the warehouse. This situation may lead to erroneous updates to the warehouse, as reported in [Z*95]. Thus, a warehouse system that uses generalized self-maintenance must either allow access only to base relations that change in lock step with the warehouse, or combine our techniques with the compensation techniques developed in [Z*95].

8 Conclusion and Future Work

We have given algorithms that test view maintainability and incrementally maintain a view in response to a base update, based on the current state of all the views in the warehouse and of a specified subset of the base relations. We improve significantly on previous work, because our methods allow us to take full advantage of all the views stored at the warehouse and to handle base updates that consist of arbitrary mix of insertions and deletions. The techniques used in obtaining the algorithms are applicable to a wide variety of views, and in some cases, allow us to generate tests and maintenance expressions in the form of SQL queries. We intend to use these algorithms as the basis for maintaining a warehouse efficiently. The practicality of our approach can further be enhanced with a better ability to optimize the queries generated by the algorithms, a better strategy for selecting which subset of the base relations to access if a view turns out to be not maintainable, and by extending our techniques to deal with multi-set semantics.

Acknowledgements

We thank Jeff Ullman for his valuable comments regarding the presentation of this material.

References

[CM77] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. 9th ACM Symp. on Theory of Computing,* pp. 77–90, 1977.

[GB95] A. Gupta and J. A. Blakeley. Using Partial Information to Update Materialized Views. In *Information Systems,* 20(8), pp. 641–662, 1995.

[GJM94] A. Gupta, H. V. Jagadish and I. S. Mumick. Data Integration Using Self-Maintainable Views. In *Technical Memorandum 113880-941101-32,* AT&T Bell Labs, November 1995.

[GM95] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *IEEE Data Engineering Bulletin, Special Issue on Materialized Views & Data Warehousing,* 18(2), June 1995.

[GMS93] A. Gupta, I. S. Mumick, and V.S. Subrahmanian. Maintaining Views Incrementally. In *Proc. ACM SIGMOD Conf.,* June 1993.

[G*94] A. Gupta, Y. Sagiv, J. D. Ullman and J. Widom. Constraint Checking with Partial Information. In *Proc. 13th ACM Symp. on Principles of Database Systems,* pp. 45–55, 1994.

[H*95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. In *IEEE Data Engineering Bulletin,* 18(2), pp. 41–48, June 1995.

[Hu96] N. Huyn. Efficient View Self-Maintenance. In *Proc. Workshop on Materialized Views: Techniques and Applications,* pp. 17–25, 1996.

[Hu97] N. Huyn. Multiple-View Self-Maintenance in Data Warehousing Environments. *Technical Report,* available as `http://www-db.stanford.edu/pub/huyn/1997/mvsm.ps`, 1997.

[IK93] W. H. Inmon and C. Kelley. Rdb/VMS: Developing the Data Warehouse. *QED Publishing Group,* Boston, Massachusetts, 1993.

[Klu88] A. Klug. On Conjunctive Queries Containing Inequalities. In *J. ACM* **35**:1, pp. 146–160, 1988.

[Kuc91] V. Kuechenhoff. On the Efficient Computation of the Difference Between Consecutive Database States. In *Int. Conf. on DOOD,* 1991.

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

[Q*96] D. Quass, A. Gupta, I. Mumick and J. Widom. Making Views Self-Maintainable for Data Warehousing. In *Proc. 4th Int. Conf. on Parallel and Distributed Information Systems,* Miami Beach, FL, Dec. 1996.

[RED] Red Brick Systems. *Red Brick Warehouse,* 1995.

[SJ96] M. Staudt and M. Jarke. Incremental Maintenance of Externally Materialized Views. In *Proc. 22nd VLDB Conf.,* 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.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems,* Volumes 1 and 2. Computer Science Press, Rockville, MD, 1989.

[Z*95] Y. Zhuge, H. Garcia-Molina, J. Hammer and J Widom. View Maintenance in a Warehousing Environment. In *Proc. ACM SIGMOD,* San Jose, California, 1995.