Multiple-View Self-Maintenance in Data Warehousing Environments

(Technical Report)

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

Abstract

A data warehouse is a collection of materialized views derived from relations that may not reside at the warehouse. Using these stored views, user queries can often be evaluated much more cheaply than using the base relations. Keeping the views consistent with updates to the base relations, however, can be expensive, since it may involve querying external sources where the base relations reside. To reduce maintenance costs, we try to maintain the views, in response to a base update, using information that is strictly local to the warehouse: the view definitions and the view contents. Only when we fail to do so do we resort to accessing a subset of the base relations. However, there may be situations where, under a specific base update and given a specific state of the views and of the subset of base relations used, there is no way to maintain a view unambiguously. Thus, the two critical questions we address are to determine, in a given situation, whether a view can be maintained unambiguously, and how to maintain it.

We provide algorithms that answer these questions for a general class of views. For an important subclass, we provide algorithms that generate SQL queries whose answer determines if a view can be maintained in a given situation. We also generate SQL updates that maintain the view. We improve significantly on previous work by solving the maintenance problem in the presence of multiple views, with partial access to base data, and under arbitrary mixes of insertions and deletions. We provide better insight into the view self-maintenance problem by showing that view self-maintainability can be reduced to the problem of deciding query containment.

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

* Work supported by ARO grant DAAH04-95-1-0192.
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 relations 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:

\[
\begin{align*}
\text{CREATE VIEW V1(X,Y,Z) AS} & \quad \text{CREATE VIEW V2(Y,Z) AS} \\
\text{SELECT R.X, R.Y, S.Z} & \quad \text{SELECT *} \\
\text{FROM R,S,T} & \quad \text{FROM S} \\
\text{WHERE R.Y = S.Y AND S.Z = T.Z} &
\end{align*}
\]

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

\(^1\)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.
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 \(V_2 = \{(b_1, c_1), (b_1, c_2), (b_1, c_2)\}\). 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: \((b_1, c_2)\). 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 \((a, b)\):

\[R = \{(a_1, b_1)\}, \ S = \{(b_1, c_1), (b_1, c_2), (b_1, c_2)\}\]

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 a mediator such as TSIMMIS ([C*94]); the data source may be temporarily disconnected or even permanently destroyed.

The contributions of this paper are as follows:

- We first consider the multiple-view self-maintenance problem where no base relations are used. We call this type of self-maintenance strict\(^2\). We also consider the special class of views defined as conjunctive queries with no projection, corresponding to SPJ (select-project-join) queries with no projection and with only equality comparisons. Given an instance of the views, we show there is a “canonical database”, consistent with all the views, we can use in place of the real unknown database to correctly propagate a base update to a view, if the view is self-maintainable. Based on this idea, we give an algorithm that generates, at view definition time, the query expressions

\(^2\)“Strict” alludes to the fact that the information used in maintenance is strictly local to the warehouse.
required to maintain the views in response to a base update, given as a set of tuples to be inserted to and/or deleted from several base relations. Most importantly, we provide better insight into the view self-maintenance problem by showing that self-maintainability can be solved completely by deciding containment of certain queries. While we can use this reduction to obtain algorithms for deciding self-maintainability at runtime, we take a step further: we give an algorithm that generates, at view definition time, the queries that test self-maintainability.

- We then extend our techniques to other important classes of views: views defined by conjunctive queries with projection (i.e., SPJ with only equality comparisons) or unions of conjunctive queries, and “views” that can be extracted from an update log, called partial copies.

- Finally we consider the generalized self-maintenance problem where, in addition to the views in the warehouse, we are also given access to some of the base relations. Based on this notion, we define a warehouse maintenance strategy that is more refined than one based on strict self-maintenance and that can yield greater efficiency. We show that the algorithms for strict self-maintenance can be extended easily to generalized self-maintenance.

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 the amount of information available for view maintenance. A major distinction is what we call compile-time vs. runtime. 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, [GJM96] addressed the single-view self-maintenance for views defined as SPJ queries and under either insertions, deletions, or updates. More recently, [Q95] 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, the problem of making 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 be solved in polynomial time: in particular, we give an algorithm that generates 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.

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 a ground update to the base relations. $U(D)$ denotes the updated database. We model $U$ as $\delta R_1^-, \delta R_1^+, \delta R_2^-, \delta R_2^+, \ldots, \delta R_n^-, \delta R_n^+$, where $\delta R_i^-$ (resp. $\delta R_i^+$) is the set of tuples to be deleted from (resp. inserted to) relation $R_i$. Updates are assumed to be self-consistent, i.e., $\delta R_j^-$ and $\delta R_j^+$ have no tuples in common for any $j$.

Strict Self-Maintenance

In strict self-maintenance, no base relations are used for maintaining a view. 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:

$$\forall i, (Q_i(D_1) = V_i) \land \left(\bigwedge_i Q_i(D_2) = V_i \implies Q_k(U(D_1)) = Q_k(U(D_2))\right)$$

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:

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

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 $\delta R^-$) and their predicate in lower case (e.g., $v_1$, $s$, and $\delta r$). 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:

$$H_i := G_{i1} \& \ldots \& G_{in}$$

where the head $H_i$ uses predicate $v_i$ for view $V_i$ and each subgoal $G_{ij}$ uses predicate $r_j$ 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. For a description of other classes of Datalog queries, see [Ull89].

**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 $E_1, \ldots, E_n$. Instance-specific QC is a variation of the QC problem where the instance of some of the input EDB relations is fixed. 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 $P \subseteq_{F_1,\ldots,F_m} Q$ 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 fixed, are called constant predicates. The EDB predicates $e_i$ are called variable predicates. When the extension of the constant predicates is known, we can always reformulate an instance-specific QC problem to a QC problem by eliminating any constant predicate $f$ as follows:
replace any subgoal $-f(\bar{x})$ 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. However, when the extension of the constant predicates is not known, we would like to find a condition on these predicates that expresses $P \subseteq_{F_1,\ldots,F_m} Q$. Whether or not such condition always exists is still an open question. In the rest of this paper, we will simply use $P \subseteq Q$ to denote $P \subseteq_{F_1,\ldots,F_m} Q$, as it will be clear from the context which input predicate is constant.

### 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:

\[
v_1(X, Y, Z) := r(X, Y) \land s(Y, Z) \land t(Z)
\]

\[
v_2(Y, Z) := s(Y, Z)
\]

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

Intuitively, we are trying to reconstruct the base relations minimally 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.

**Proof:** Let $D$ be the actual underlying database. First, since every $Q_i$ has no projection, by construction of $\hat{D}$, each tuple in $\hat{D}$ is needed in order to explain the presence of some tuple in some view, and thus must be in $D$. So $D \subseteq \hat{D}$. Further, for each $i$, since $Q_i$ is monotonic, $Q_i(\hat{D}) \subseteq Q_i(D)$ holds. Since we have assumed that $D$ is consistent with all the given views, we conclude that $Q_i(\hat{D}) \subseteq V_i$ for each $i$. Second, for each $i$, since $\hat{D}$ contains all the atoms that contribute to every tuple in $V_i$, we can infer $Q_i(\hat{D}) \supseteq V_i$, completing the proof that $Q_i(D) = V_i$ for each $i$.

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 reduce the problem of maintaining a view without using any base relation to a view maintenance problem with unrestricted use of the base relations.

**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 \(Q_k(U(D))\), where \(D\) is the canonical database.

**Proof:** Let \(D\) be the actual underlying database. If view \(V_k\) is self-maintainable under \(U\), referring to the definition (1) of self-maintainability, then its new state must be unique, i.e., \(Q_k(U(D))\) is identical to \(Q_k(U(D_2))\) for any database \(D_2\) that is consistent with all the views, and for \(\hat{D}\) in particular. Thus, without knowing the exact content of \(D\), \(V_k\)’s new state can be computed using \(Q_k(U(\hat{D}))\).

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

**Algorithm 3.1 (Generate multiple-view self-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. Generate the following rules that define the predicates \(\hat{r}_1, \ldots, \hat{r}_n\) for the canonical database, for \(i = 1, \ldots, m\) and \(j = 1, \ldots, n_i\):
   \[
   (A_{ij}) : \quad \hat{G}_{ij} := H_i
   \]
   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 \(\hat{r}_j\).
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 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, for example based on algebraic techniques for differentiating query expressions. 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 \( \delta v_1^+ \) be the predicate for the set of net insertions to \( V_1 \). Algorithm 3.1 generates the following query for \( \delta v_1^+ \):

\[
\begin{align*}
\hat{s}(Y, Z) & : v_1(X, Y, Z) \mid v_2(Y, Z) \\
\hat{t}(Z) & : v_1(X, Y, Z) \\
\delta v_1^+(a, b, Z) & : \hat{s}(b, Z) \land \hat{t}(Z) \land \neg v_1(a, b, Z)
\end{align*}
\]

which can further be simplified into

\[
\begin{align*}
\delta v_1^+(a, b, Z) & : v_1(X, b, Z) \land v_1(X', Y', Z) \land \neg v_1(a, b, Z) \\
\delta v_1^+(a, b, Z) & : v_2(b, Z) \land v_1(X', Y', Z) \land \neg v_1(a, b, Z)
\end{align*}
\]

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 \( V_2 = \{(b_1, c_1), (b_1, c_2)\} \). \( V_1 \) 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 \( V_2 = \{(b_1, c_1), (b_1, c_2)\} \). \( \hat{D} \) in this case consists of \( R = \{(a_1, b_1)\}, S = \{(b_1, c_1), (b_1, c_2)\}, T = \{(c_1)\} \). \( V_1 \) 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 \( R = \{(a_1, b_1)\}, S = \{(b_1, c_1), (b_1, c_2)\}, T = \{(c_1), (c_2)\} \) 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 $\bigwedge_i Q_i(D) \neq V_i$, for every database $D$ ($\hat{D}$ is the canonical database).

Proof: 
ONLY IF: Assume $V_k$ is self-maintainable under $U$. Let $D$ be an arbitrary database. Substituting $D$ and $\hat{D}$ for $D_1$ and $D_2$ in definition (1) of self-maintainability, we obtain:

$$\bigwedge_i (Q_i(D) = V_i) \land \bigwedge_i (Q_i(\hat{D}) = V_i) \Rightarrow Q_k(U(D)) = Q_k(U(\hat{D}))$$

And since $\hat{D}$ is consistent with all the views by Lemma 3.1, $\bigwedge_i (Q_i(D) = V_i) \Rightarrow Q_k(U(D)) = Q_k(U(\hat{D}))$ follows.

IF: Conversely, assume that $\bigwedge_i (Q_i(D) = V_i) \Rightarrow Q_k(U(D)) = Q_k(U(\hat{D}))$ holds for every database $D$. Let $D_1$ and $D_2$ be two databases that are consistent with all the views. It follows that $Q_k(U(D_1)) = Q_k(U(D_2)) = Q_k(U(\hat{D}))$. We conclude $Q_k(U(D_1)) = Q_k(U(D_2))$, thus showing that (1) holds.

Note that in Theorem 4.1, we use the implication “different-effect implies inconsistency” instead of “consistency implies same-effect”. While both forms are equivalent, the queries to compare in the first one are slightly simpler. 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 $\tilde{D}$; $(B_k)$ 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 $r''_j$ for relation $R_j$ in $U(D)$; $(H_k)$ defines predicate $v'_k$ for $Q_k(U(\hat{D}))$, the new state of view $V_k$ that derives from $D$ 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 multiple-view 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 $U = \delta R_1^{\pm}, \delta R_2^{\pm}, \ldots, \delta R_n^{\pm}$.

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

Each of $\text{DIFF}$ and $\text{INCON}$ 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 that applies to variable predicates 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 this type 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.

Proof: Let $D$ be a database that is consistent with all the views. When each of the $Q_i$’s is a conjunctive query without projection, each tuple in $\hat{D}$ is needed in order to explain the presence of some tuple in some view. In other words, every tuple in $\hat{D}$ must be present in $D$.

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 $V_1 = \{(a_1, b_1, c_1), (a_2, b_1, c_2)\}$ and $V_2 = \{(b_1, c_1), (b_2, c_2), (b_1, c_2)\}$. 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. Note that to represent a superset of \( \hat{D} \), we did not use an arbitrary database \( D \) subject to the constraint \( D \supseteq \hat{D} \), precisely to avoid the undesirable type of negation mentioned in Section 4.

**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 \( Q_k(U(D \cup \hat{D})) \subseteq Q_k(U(\hat{D})) \) implies \( \forall_i Q_i(D \cup \hat{D}) \subseteq V_i \), for every database \( D \), where \( \hat{D} \) is the canonical 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.

**Proof:** Following Theorem 4.1, \( V_k \) is self-maintainable under \( U \) if and only if \( Q_k(U(D)) = Q_k(U(\hat{D})) \) for every database \( D \) that is consistent with all the views. Applying Lemma 5.1, the validity of the latter statement does not change if we substitute “every superset of \( D \)” for “every database \( D’ \)”. Since a superset of \( \hat{D} \) can be equivalently represented as \( D \cup D \) for some \( D \), it follows that \( V_k \) is self-maintainable under \( U \) if and only if:

\[
(\forall D)^\bigcap_i Q_i(D \cup \hat{D}) = V_i \Rightarrow Q_k(U(D \cup \hat{D})) = Q_k(U(\hat{D}))
\]  

Furthermore, for every \( i \), since \( Q_i \) is monotonic and \( Q_i(\hat{D}) = V_i \), \( Q_i(D \cup \hat{D}) \supseteq V_i \) always holds. The first equality in (2) is equivalent to \( Q_i(D \cup \hat{D}) \subseteq V_i \). Similarly, since both \( Q_k \) and \( U \) are monotonic, \( Q_k(U(D \cup \hat{D})) \supseteq Q_k(U(\hat{D})) \) always holds. Thus, the second equality in (2) is equivalent to \( Q_k(U(D \cup \hat{D})) \subseteq Q_k(U(\hat{D})) \).

To complete the proof, we refer to Table 2. In the rules that define panic, namely rules \((B_k^r)\) and \((I_k)\), negation only apply to \( H_i \) and \( H_k' \), which are both defined entirely in terms of the constant predicates \( v_1, \ldots, v_m \). Further, negation also appears when expanding the subgoals \( G_{kj} \) from rule \((I_k)\), but it only applies to the constant predicates \( \delta r_j^- \).

**Theorem 5.1** improves on Theorem 4.1 in eliminating the uses of \( \supseteq \) which were the main source of exponential complexity in Algorithm 4.1: \( \supseteq \) 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 \( r_j \), have been eliminated. The rules in Table 2 relate to Theorem 5.1 as follows: rules \((A_{ij})\) compute \( \hat{D} \); \((K_j)\) defines a predicate \( r_j^w \) for relation \( R_j \) in \( D \cup \hat{D} \); \((B_j^r)\) represent the fact that \( Q_i(D \cup \hat{D}) \nsubseteq V_i \); \((D_j^r)\) defines a predicate \( r_j^u \) for relation \( R_j \) in \( U(D \cup \hat{D}) \); \((F_j)\) defines predicate \( r_j^f \) for relation \( R_j \) in \( U(\hat{D}) \); \((H_k)\) defines predicate \( v_k^f \) 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 multiple-view 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.
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. 

Proof: (Sketch) The proof has two parts. In the first part, we show that $P \subseteq Q$ can be characterized completely by a logical expression. We begin by stating a theorem from [G*94] on which the results that follow will be based. The theorem allows us to translate containment of conjunctive queries with arithmetic comparisons to a logical expression. We paraphrase it as follows:

<table>
<thead>
<tr>
<th>Rules</th>
<th>Range</th>
<th>DIFF</th>
<th>INCON</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(A_{ij})$</td>
<td>$i = 1, \ldots, m$ $j = 1, \ldots, n_i$</td>
<td>$G_{ij} : = H_i$</td>
<td>$G_{ij} : = H_i$</td>
</tr>
<tr>
<td>$(K_j)$</td>
<td>$j = 1, \ldots, n$</td>
<td>$r_j^0 : = r_j \mid \bar{r}_j$</td>
<td>$r_j^0 : = r_j \mid \bar{r}_j$</td>
</tr>
</tbody>
</table>
| $(B_i)$ | $i = 1, \ldots, m$ | $\text{panic} : = G_{i1}^0 \& \ldots \& G_{im}^0 \& \neg H_i$ | \$

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

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 the boolean queries $\text{DIFF}$ and $\text{INCON}$ as shown in Table 2. Both queries use the 0-ary predicate $\text{panic}$ for their query predicate.

2. Generate a query $\text{TEST}$ that decides whether $\text{DIFF} \subseteq \text{INCON}$. Return $\text{TEST}$.

In Algorithm 5.1, instead of generating a test in Step 2, we could have solved $\text{DIFF} \subseteq \text{INCON}$ directly by using known algorithms in the literature ([G*94, Khu88]) 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 $\text{DIFF} \subseteq \text{INCON}$ 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.
Theorem 5.3 ([G*94]) Consider the conjunctive queries \( Q_i : H_i : B_i, A_i \) where \( B_i \) represents a conjunction of ordinary subgoals and \( A_i \) a conjunction of arithmetic comparisons. \( B_i \) is assumed not to use the same variable twice or any constant. Let \( M \) be the set of containment mappings from \((H_2, B_2)\) to \((H_1, B_1)\). Then \( Q_1 \subseteq Q_2 \) if and only if \( A_1 \Rightarrow \bigvee_{h \in M} h(A_2) \).

Theorem 5.3 can be easily generalized by replacing “a conjunction of arithmetic comparisons” with “a boolean combination of arithmetic comparisons”. From this generalization, we can derive the following theorem, which essentially expresses containment of queries involving constant predicates as a logical expression using these predicates rather than their extension.

Theorem 5.4 Consider the queries \( Q_i : H_i (X_i', Y_i', Z_i') := A_i (X_i', Y_i', Z_i), B_i (Y_i, Z_i), \neg c_i (X_i', Y_i, Z_i) \) where \( A_i \) represents a conjunction of subgoals with constant predicates (some of these subgoals may be negated), \( B_i \) a conjunction of positive subgoals with variable predicates and \( c_i \) an atom with a constant predicate. Let \( M \) be the set of containment mappings from \((H_2, \text{rectified } B_2)\) to \((H_1, B_1)\). Let \( G_h (Y_1, Z_1) \) be the result of applying some \( h \in M \) to the equality comparisons obtained from the rectification of \( B_2 \). Then \( Q_1 \subseteq Q_2 \) if and only if:

\[
(\forall X_1, Y_1, Z_1) \ A_1 (X_1, Y_1) \land \bigwedge_{h \in M} [\neg G_h (Y_1, Z_1) \lor [(\forall X_2) \ A_2 (X_2, h(Y_2)) \Rightarrow c_2 (X_2, h(Y_2), h(Z_2))] \]

\[
\Rightarrow c_1 (X_1, Y_1, Z_1) \Rightarrow \bigwedge_{h \in M} [\neg G_h (Y_1, Z_1) \lor [(\forall X_2) \ A_2 (X_2, h(Y_2)) \Rightarrow c_2 (X_2, h(Y_2), h(Z_2))] \]

(3)

In this theorem, \( X_i \), \( Y_i \), and \( Z_i \) represent mutually disjoint sets of variables, and \( X_i' \), \( Y_i' \), and \( Z_i' \) are subsets of \( X_i \), \( Y_i \), and \( Z_i \) respectively. Also, a set of subgoals is said to be rectified when no variables occur more than once among the subgoals, and no constant symbols are used. Refer to [G*94] for details on rectification.

The second part shows that the logical expression (3) can always be equivalently rewritten as a safe boolean query that uses the constant predicates as input. A general transformation we often use is to rewrite \( A \lor B \Rightarrow C \) as the conjunction of the two formulas \( A \Rightarrow C \) and \( B \Rightarrow C \). Thus we can eliminate the \( \lor \) in (3) to obtain a conjunction of formulas each having the following form:

\[
(\forall Z) \ a_1 (Z_1) \land a_2 (Z_2) \land \ldots \land a_n (Z_n) \Rightarrow r(Z) \]

(4)

where \( Z_i \subseteq Z \) for each \( i \), \( r \) represents a safe query, and \( a_i (Z_i) \) denotes a query having one of the following forms:

1. \( q_i (Z_i) \) or \( \neg q_i (Z_i) \) where \( q_i \) represent a safe query.

2. \( (\forall X_i) \ p_i (X_i) \Rightarrow q_i (X_i, Z_i) \), where \( p_i \) and \( q_i \) represent safe queries.

3. \( (\forall X_i) \ p_i (X_i, Z_i') \Rightarrow q_i (X_i, Z_i) \) where \( Z_i' \subseteq Z_i \) and \( Z_i' \neq \emptyset \), and \( p_i \) and \( q_i \) represent safe queries.

4. \( \mu \neq \nu \), where \( \mu \) and \( \nu \) are either constants or variables from \( Z_i \). We assume that no comparison involves the same variable.

We now define the notion of finite and infinite queries we will use later: a finite query \( F \) in \( X \), denoted \( F (X) \), is constructed from \( p (X) \) where \( p \) is the predicate for a safe query, \( (\exists Y) F' (X, Y) \) where \( F' \) is a finite query, the conjunction of finite queries, or the conjunction of a finite query with any query in some subset of \( X \); an infinite query \( I (X) \) is constructed from \( \neg F (X) \) where \( F \) if a finite query, or from \( \neq \) comparisons that involves variables in \( X \).

Next, we express each \( a_i (Z_i) \) as the union of a finite query and an infinite query. This expression varies, depending on which of the four forms \( a_i \) can take (listed above):
1. $a_i(Z_i)$ is already in the form of a finite/infinite query.

2. Rewrite $a_i(Z_i)$ as the union of the finite query in $\tilde{Z}_i$, $(\exists \tilde{X}_i) \ q_i(\tilde{X}_i, \tilde{Z}_i) \land a_i(Z_i)$, with the infinite query (in an empty set of variable), $- (\exists \tilde{X}_i) \ p_i(\tilde{X}_i)$.

3. Rewrite $a_i(Z_i)$ as the union of the finite query in $\tilde{Z}_i$, $[(\exists \tilde{X}_i) \ q_i(\tilde{X}_i, \tilde{Z}_i)] \land a_i(Z_i)$, with the infinite query in $\tilde{Z}_i'$, $- (\exists \tilde{X}_j) \ p_j(\tilde{X}_j, \tilde{Z}_j')$.

4. $a_i(Z_i)$ is already in the form of an infinite query.

After replacing each $a_i(Z_i)$ with a union of finite and infinite queries, and after eliminating the resulting $\lor$'s, (4) rewrites to a conjunction of formulas, each having the following form:

$$\forall \tilde{Z}(\bigwedge_i F_i(\tilde{X}_i) \land \bigwedge_j I_j(\tilde{Y}_j) \Rightarrow F(\tilde{Z}))$$

(5)

where $\tilde{X}_i \subseteq \tilde{Z}$ and $\tilde{Y}_j \subseteq \tilde{Z}$, $F$ and the $F_i$'s are finite queries, and the $I_j$'s are infinite queries.

Formula (5) is safe if $\bigcup_i \tilde{X}_i = Z$. Otherwise, the formula is equivalent to the following safe formula:

$$-(\exists \bigcup_i \tilde{X}_i) \bigwedge_i F_i(\tilde{X}_i) \land \bigwedge_k I_k(\tilde{Y}_k)$$

where $k$ ranges over those $j$ such that $\tilde{Y}_j \subseteq \bigcup_i \tilde{X}_i$.

Theorem 5.2 allows us to translate $P \subseteq Q$ to a safe boolean query, but for $P$ and $Q$ that are conjunctive queries. To translate $DIFF \subseteq INCQ$, we need to solve $P \subseteq Q$ for $P$ and $Q$ that are unions of conjunctive queries. Let $P$ be the union of $P_i$'s and let $Q$ be the union of $Q_j$'s. $P \subseteq Q$ is equivalent to the conjunction over $i$ of $P_i \subseteq Q$. And Theorem 5.2 can be easily extended to handle the latter. We only need minor modifications to the proof: in Theorem 5.4, replace all references to 2 with $j$; in (3), replace $\bigwedge_{h \in M} \bigwedge_{l \in M_j} \bigwedge_{h \in M_j}$ where $M_j$ is the set of containment mappings from $(H_j, rectified \ B_j)$ to $(H_B, B_1)$.

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 query test in query form. This algorithm is shown Appendix A.

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 view 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):

$$p(Z) : v_1(X, Y, Z)$$
$$q(Z) : v_2(Y, Z) \land v_1(X, Y, Z')$$
$$depend : v_2(b, Z) \land \neg p(Z) \land \neg q(Z)$$
$$maintainable : \neg \mathrm{depend}$$
or equivalently in SQL:

\[
\text{NOT EXISTS (SELECT \ast \text{ FROM } V_2 \text{ WHERE } V_2.Y = b}
\text{ AND NOT EXISTS (SELECT \ast \text{ FROM } V_1 \text{ WHERE } V_1.Z = V_2.Z)}
\text{ AND NOT EXISTS (SELECT \ast \text{ FROM } V_2 \text{ as } V_2', V_1}
\text{ \text{ WHERE } V_2'.Z = V_2.Z \text{ AND } V_2'.Y = V_1.Y))}
\]

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 in this paper.

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 loses 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. In the following, we redefine our notion of canonical database.

**Definition 6.1 (Canonical database in views with projection):** Let \( V_1, \ldots, V_m \) be given views and for \( i = 1, \ldots, m \), let \( Q_i \) be the conjunctive query over relations \( R_1, \ldots, R_n \) that defines \( V_i \). The canonical database, denoted \( D \), consists of all the tuples obtained 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 \( D \).

**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 \( V = \{(a_1, c_1), (a_2, c_2)\} \). The redefined canonical database \( D \) consists of \( S = \{(a_1, y_1), (y_1, c_1), (a_2, y_2), (y_2, c_2)\} \), where \( y_1 \) and \( y_2 \) are new symbols.

A tuple in \( D \) 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 \( D \) represents not a single database but a class of possible databases, each of which is obtained by applying some substitution \( h \) to \( D \). The following example illustrates the non-uniqueness of minimal databases due to projections in views.

**Example 6.2** Consider the same view definition as in Example 6.1, but a different view instance \( V = \{(d, c)\} \). Consider the insertion of \( (a, b) \) into \( S \). The canonical database \( \hat{D} \), \( S = \{(d, y), (y, c)\} \) where \( y \) is a new symbol, actually can be interpreted in five possible ways (by mapping \( y \) to either \( y, a, b, d, \) or \( c \)):

- \( S = \{(d, y), (y, c)\} \)
- \( S = \{(d, a), (a, c)\} \)
- \( S = \{(d, b), (b, c)\} \)
- \( S = \{(d, d), (d, c)\} \)
- \( S = \{(d, c), (c, c)\} \).

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 \(S\). \[
\]
In the following theorem, we state that among all the possible interpretations (each corresponding to a mapping), there is always one that is consistent with all the views. A mapping that gives an interpretation that is consistent with all the views is said to be consistent.

**Theorem 6.1** Let \(V_1, \ldots, V_m\) be given views and for \(i = 1, \ldots, m\), let \(Q_i\) be the conjunctive query over relations \(R_1, \ldots, R_n\) that defines \(V_i\). Let \(\bar{D}\) be the canonical database, and \(\text{new}\) the set of new symbols generated. Let \(S_{\text{views}}\) be the set of symbols that appears in the view instance, and \(S\) another set of symbols. Then there is always a mapping \(h\) that maps every symbol from \(\text{new}\) to some symbol in \(\text{new} \cup S_{\text{views}} \cup S\), and such that database \(h(\bar{D})\) is consistent with all the views. \[
\]
To maintain a view, \(S\) in the theorem is not important and can be ignored in the search for a consistent mapping \(h\). One we find such an \(h\), 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 \(h(\bar{D})\) as the actual database.

For the self-maintainability question, we must choose \(S\) to contain all the symbols from the given update instance. Then, the reduction from Section 5 is extended to take into account the non-uniqueness of a minimal consistent database. The following theorem formalizes the new reduction.

**Theorem 6.2** Let \(V_1, \ldots, V_m\) be given views, and for \(i = 1, \ldots, m\), let \(Q_i\) be the conjunctive query that defines \(V_i\). Let \(\bar{D}\) be the canonical database, \(\text{new}\) the set of new symbols generated, and \(S_{\text{views}}\) be the set of symbols that appears in the view instance. Let \(U\) be an update to \(\bar{D}\), and \(S_{\text{upd}}\) the set of symbols that appears in \(U\). Let \(M\) be the set of mappings from \(\text{new}\) to \(\text{new} \cup S_{\text{views}} \cup S_{\text{upd}}\). Then \(V_k\) is self-maintainable under \(U\) if and only if (1) \(Q_k(U(h(\bar{D}))) = Q_k(U(h'(\bar{D})))\) holds for every consistent \(h\) and \(h'\) in \(M\), and (2) \(Q_k(U(\bar{D} \cup h(\bar{D}))) \subseteq Q_k(U(h(\bar{D})))\) implies \(V_i Q_k(\bar{D} \cup h(\bar{D})) \not\subseteq V_i\), for every database \(\bar{D}\) and for every consistent \(h\) in \(M\). \[
\]
To sum it up, we obtain algorithms similar to Algorithms 3.1 and 5.1 except that they use a minimal database \(h(\bar{D})\) that is consistent with all the views, instead of just \(\bar{D}\). 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 \(\text{mult}(v)\) many rules), the presence of each tuple in the view can be explained by more than one set of facts (\(\text{mult}(v)\) many sets to be exact). Thus, given an instance of \(V_1, \ldots, V_m\), the number \(d\) of possible canonical databases that are consistent with the views is

\[
\text{mult}(v_1)^{\text{size}(V_1)} \times \ldots \times \text{mult}(v_m)^{\text{size}(V_m)}.
\]
Thus, like projections, unions introduce non-uniqueness of canonical databases consistent with the views. Canonical databases have the following properties:

- Any database that is consistent with all the views must contain some canonical database that is necessarily consistent with all the views (since the queries defining the views are monotonic).
- Among all the canonical databases, there is at least one that is consistent with all the views (following the view realizability assumption).
- Not every canonical database is consistent with all the views.

The following example illustrates the non-uniqueness of canonical databases due to unions in views.

**Example 6.3** 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 \( V_2 \).

To maintain a view \( V_k \), we can apply the same idea as in Section 3: propagate the update to \( V_k \) using some canonical database that is consistent with all the views. The choice of a canonical database is not important.

To answer the self-maintainability question, we must consider all the canonical databases that are consistent with all the views. Using these databases, the problem can be solved in a way that parallels the case of views with projections.

**Theorem 6.3** Let \( V_1, \ldots, V_m \) be given views, and for \( i = 1, \ldots, m \), let \( Q_i \) be the union of conjunctive queries that defines \( V_i \). Let \( D_1, \ldots, D_n \) be all the canonical databases that are consistent with all the views. Let \( U \) be an update to \( D \). Then \( V_k \) is self-maintainable under \( U \) if and only if (1) \( Q_k(U(D_j)) = Q_k(U(D_k)) \) holds for every pair \( D_j \) and \( D_k \), and (2) \( \bigwedge_i Q_i(D \cup D_j) \subseteq V_i \) implies \( Q_k(U(D \cup D_j)) \subseteq Q_k(U(D_j)) \), for every database \( D \) and for every \( D_j \).

**Proof:**

**IF:** Let \( D_1 \) and \( D_2 \) be two databases that are consistent with the views. \( D_1 \) contains some \( D_j \) and \( D_2 \) contains some \( D_l \). Applying (2), we infer that \( Q_k(U(D_1)) = Q_k(U(D_j)) \) and \( Q_k(U(D_2)) = Q_k(U(D_l)) \). Applying (1), we conclude that \( Q_k(U(D_1)) = Q_k(U(D_2)) \). Thus, \( V_k \) is self-maintainable under \( U \).

**ONLY-IF:** Conversely, assume \( V_k \) is self-maintainable under \( U \). Any pair of databases that are consistent with the views must derive the same view after update, in particular any pair \( D_j \) and \( D_l \). So (1) holds. To verify (2), let \( D \) be a database and \( D_j \) a canonical database that is consistent with the views. Assume that \( D \cup D_j \) (call it \( D' \)) is consistent with the views. Since \( V_k \) is self-maintainable under \( U \), it follows that \( Q_k(U(D')) = Q_k(U(D_j)) \).

It is easy to see that Theorem 6.3 continues to hold if we consider only those canonical databases (consistent with the views) that are minimal. That is, we can ignore those canonical databases that contain some canonical databases consistent with the views.

In summary, to test self-maintainability of views that are unions of conjunctive queries:

- We compute all the minimal canonical databases that are consistent with the views.
- We then determine if updating any one of them has the same effect on all the views.
- For each minimal canonical database, we execute a query similar to what Algorithm 5.1 generates against the views and the canonical database.
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, an obvious strategy is to fall back to the “normal” but expensive maintenance mode with unrestricted access to the base relations, as depicted in Figure 2(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 2(b) illustrates this strategy. Note that in Figure 2(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),

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 \( \delta R^- \), \( \delta R^+ \), \( \delta S^- \), \( \delta S^+ \), \( \delta T^- \), and \( \delta T^+ \). 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:

\[ s(Y, Z) := s(Y, Z) \land \neg \delta s^+(Y, Z) \lor \delta s^-(Y, Z) \]
Predicate $\mathcal{S}$ 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

A Generating Safe Queries Whose Answer Decide Containment

Essentially, given two queries, the first algorithm generates a logical expression that captures the condition for containment for the queries. Using the second algorithm, 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.

Algorithm A.1 (Generate safe logical expression for containment test)
Input: $P$, the union of $P_i$’s, and $Q$, the union of $Q_j$’s. $P_i$’s and $Q_j$’s are conjunctive queries where negation only applies to subgoals with constant predicates. Each $P_i$ is assumed to have a negated subgoal that uses all the variables in $P_i$. Each $Q_j$ is assumed to have only one negated subgoal, and that subgoal uses all the variables in $Q_j$.

Notation: For a conjunctive query $R$, we use the following notation: $\text{head}(R)$ is $R$’s head; $\text{negated}(R)$ is the literal of a negated subgoal that uses all the variables in $R$ (the choice being arbitrary but fixed); $\text{known}(R)$ is the conjunction of all subgoals with constant predicates, excluding $\text{negated}(R)$; $\text{unknown}(R)$ is the conjunction of all subgoals with variable predicates; $\text{privar}(R)$ denotes the variables in the positive subgoals in $\text{known}(R)$ but not in $\text{unknown}(R)$; $\text{rect}(R)$ is the conjunction of all the equality comparisons obtained from rectifying the subgoals in $\text{unknown}(R)$.

Output: A safe logical expression (in terms of the constant predicates) that is satisfied if and only if $P \subseteq Q$ for all instances of the variable predicates.

Method:

1. For each $i$, generate $E_i$, the following formula:

   $$\left(\forall \text{vars}(P_i)\right)$$
   $$\left(\text{known}(P_i)\right) \land$$
   $$\left(\bigwedge_i \bigwedge_j \neg h_{ij}(\text{rect}(Q_j)) \lor (\forall \text{privar}(P_i))[h_{ij}(\text{known}(Q_j)) \Rightarrow h_{ij}(\text{negated}(Q_j))]ight)$$
   $$\Rightarrow \text{negated}(P_i)$$

   where $h_{ij}$ ranges over all containment mappings that map rectified $\text{unknown}(Q_j)$ to $\text{unknown}(P_i)$ and $\text{head}(Q_j)$ to $\text{head}(P_i)$.

2. By distributing the conjunctions over the disjunctions, rewrite $E_i$ as a conjunction of implications $F_{ij}$’s, each of which has only conjunctions in the premise.

3. Translate each $F_{ij}$ to $G_{ij}$, a safe formula.

4. Return $\bigwedge_i \bigwedge_j G_{ij}$.

Step 3 in Algorithm A.1 can be implemented using the following algorithm:

Algorithm A.2 (Make Safe)

Input: $H$, a finite query in $\overline{Z}$, and for $i = 1, \ldots, n$, $G_i$, a formula with free variables $\overline{Z}_i$. For each $i$, assume $\overline{Z}_i \subseteq \overline{Z}$ and $G_i$ has one of the four forms mentioned in the proof for Theorem 5.2.

Notation: As stated in the proof for Theorem 5.2, each $G_i$ is equivalent to the union of a finite query, denoted $\text{finite}(G_i)$, with an infinite query, denoted $\text{infinite}(G_i)$; for each $i = 1, \ldots, n$, $\text{safewars}(i)$ denotes $\text{vars}(\text{finite}(G_i))$, and $\text{unsafewars}(i)$ denotes $\text{vars}(\text{infinite}(G_i))$; for any subset $I$ of $\{1, \ldots, n\}$, $\text{safewars}(I)$ denotes $\bigcup_{i \in I} \text{safewars}(i)$.

Output: A safe formula that is equivalent to $(\forall \overline{Z}) \bigwedge_i G_i \Rightarrow H$.

Method:

1. For each $I \subseteq \{1, \ldots, n\}$, let $E_I$ be a safe expression constructed as follows:
If $\text{saf evars}(I) = \text{vars}(H)$, then $E_I$ is:

$$-(\exists \text{vars}(H)) \land \bigwedge_{i \in I} \text{finite}(G_i) \land \bigwedge_{j \in J} \text{infinite}(G_j) \land \neg H$$

If $\text{saf evars}(I) \neq \text{vars}(H)$, then $E_I$ is:

$$-(\exists \text{saf evars}(I)) \land \bigwedge_{i \in I} \text{finite}(G_i) \land \bigwedge_{j \in J, \text{unsaf evars}(j) \subseteq \text{saf evars}(I)} \text{infinite}(G_j)$$

where $J$ denotes $\{1, \ldots, n\} - I$.

2. Return $\bigwedge_{I \subseteq \{1, \ldots, n\}} E_I$.  
