Expiring Data in a Warehouse

Hector Garcia-Molina, Wilburt Labio, Jun Yang
Department of Computer Science
Stanford University
{hector, wilburt, junyang}@cs.stanford.edu

Abstract

Data warehouses collect data into materialized views for analysis. After some time, some of the data may no longer be needed or may not be of interest. In this paper, we handle this by expiring or removing unneeded materialized view tuples. A framework supporting such expiration is presented. Within it, a user or administrator can declaratively request expirations and can specify what type of modifications are expected from external sources. The latter can significantly increase the amount of data that can be expired. We present efficient algorithms for determining what data can be expired (data not needed for maintenance of other views), taking into account the types of updates that may occur. **Keywords**: materialized views, data warehousing

1 Introduction

Materialized views are often used to store warehouse data. The amount of data copied into these views may be very large; for instance, [JMS95] cites a major telecommunications company that collects 75GB of detailed call data every day or 27TB a year. Even with cheap disks, it will be desirable to remove some of the data from the views, either because it is no longer of interest or no longer relevant. Often, a summary of the removed data will suffice. In the telecommunication example, for instance, only detailed call data for the most recent year, and summary data from previous years, may be kept.

The traditional way of removing data from materialized views is deletion. When tuples are deleted from a view or a relation, the effect must be propagated to all "higher-level" views defined on the view/relation undergoing the deletion. However, sometimes the desired semantics are different. In particular, when the data is removed due to space constraints alone, it is desirable not to affect the higher-level views. In this paper, we propose a framework that gives us the option to gracefully *expire* data, so that the higher-level views remain unaffected and can be maintained consistently with respect to future updates. The difference between deletion and expiration is illustrated further in the next example.

EXAMPLE 1.1 Suppose the following *base relation* views copy data from *source relations* external to the warehouse. (These views will be used as a running example in this paper.)

- $Customer(custId, info)$ contains information about each customer identified by the key $custId$. For conciseness, we shall refer to $Customer$ as C.

O	$ordId$	$custId$	$clerk$
	1	456	`Clerk1`
	3	789	`Clerk2`

L	$partId$	$ordId$	qty	$cost$
l_1:	a	1	1	19.99
l_2:	b	1	2	250.00
l_3:	c	3	1	500.00

V	$partId$	qty	$cost$	$custId$	$clerk$
v_1:	b	2	250.00	456	`Clerk1`
v_2:	c	1	500.00	789	`Clerk2`

Figure 1: Current state of O, L, and V.

- *Order*($ordId, custId, clerk$), denoted O, contains for each order, the customer who requested the order and the clerk who processed the order.
- *LineItem*($partId, ordId, qty, cost$), denoted L, details the quantity of the parts and the unit cost of each part requested in each order.

Consider a simple materialized view V storing order information for expensive parts. V is defined as a natural join of O and L, with the selection condition $L.cost > 99$, followed by a projection onto relevant attributes. The current state of O, L, and V is depicted in Figure 1.

In reality, tables O and L (often called fact tables) can become quite large. Suppose that the warehouse administrator decides to *delete* "old" L tuples with $ordId < 2$. Thus, l_1 and l_2 are deleted, as if they have never existed in L. As a result, v_1 is deleted from V, which might not be desirable if users still expect V to reflect information about old tuples (especially if the view contains summary data).

The method we propose instead is to *expire* L tuples with $ordId < 2$. Tuple l_1 can be safely removed from L because $l_1.cost$ is less than 99. On the other hand, l_2 must be retained because it might be needed to correctly update V if another tuple with $ordId = 1$ is inserted into O. Notice that V remains unaffected by the expiration of L tuples. Furthermore, after the expiration, there is still enough information in L to maintain V with respect to future updates.

If we know the types of modifications that may take place in the future, we may be able to remove tuples like l_2. For example, suppose both O and L are "append-only." That is, the source relations (that O and L are based on) never delete tuples. Moreover, an insertion to O always has an $ordId$ greater than the current maximum $ordId$ in O; insertions to L always refer to the most recent order, *i.e.*, the O tuple with the maximum $ordId$. In this case, we can expire both l_1 and l_2 since they will never be needed to maintain V. In fact, it is possible to expire the entire L and O views except for the tuple recording the most recent order. In our framework, one can define applications constraints, such as "append-only," using a general constraint language, so that the system can remove as much data as possible, when the warehouse administrator so wishes it. □

To recap, although expired tuples are *physically* removed from the extension of a view, they still exist *logically* from the perspective of the higher-level views. Our expiration scheme guarantees that expiration never results in incomplete answers for view maintenance queries, given any possible source updates. Knowledge of constraints on these updates can dramatically improve the effectiveness of expiration. User queries may, however, request data that has been expired. In such cases an incomplete answer must be provided, with an appropriate warning that describes which of the requested data was actually available.

Unfortunately, current warehouse products provide very little support for gracefully expiring data. Every time there is a need to expire data, it is up to the administrator to *manually* examine view definition queries and view maintenance queries and to check if underlying data is needed for maintenance. This "solution" is clearly problematic since not only is it inefficient, but it is prone to human error which can easily lead to the expiration of needed data. Furthermore, deciding what is needed and what can be expired is complicated by the presence of constraints. If a conservative approach is used (*e.g.*, constraints are not taken into account), then the storage requirement of the warehouse may become prohibitively large.

In this paper we propose a framework wherein expiration of data is managed, not manually, but by the system. In particular:

- The administrator or users can declaratively request to expire part of a view, and the system automatically expires as much unneeded data as possible.
- The administrator can declare in a general way constraints that apply to the application data as well as changes to the data (*e.g.*, table O is append-only), and the system uses this knowledge to increase the amount of data that may be expired.
- The administrator or users can change framework parameters (*e.g.*, by defining additional views or changing application constraints) dynamically, and the system determines the effects of these changes on what data is deemed needed and what data can be expired.

For this framework we develop efficient algorithms that check what data can be expired, handle insertions of new data, and manage changes to views and constraints. We also illustrate, using the TPC-D benchmark [Com], the benefits of incorporating constraints into the management of expired data.

The rest of the paper proceeds as follows. In Section 2, we introduce our expiration framework and identify problems that need to be solved. The central problem of identifying the needed tuples is solved in Section 3, while Section 4 extends the mechanism to take into account input constraints. We illustrate in Section 5 that the "constraint-aware" solution can lead to much more data being expired. In Section 6, we develop algorithms that handle changes to the framework parameters. We discuss related work in Section 7 and conclude the paper in Section 8.

2 Framework

In this section, we present our framework for expiration. We then give an overview of the problems that we address in the rest of the paper to implement the framework.

Tables and Queries: We consider two types of warehouse *tables*: *base relations* and *materialized views*. Each base relation (*e.g.*, *Order*) has an *extension* that stores persistently a bag of tuples obtained from a source relation external to the warehouse. Each (materialized) view V has an extension that stores the answer to its *definition query*, $Def(V)$, which is of the form $\pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}} R)$. (We assume that the π, σ, $\times$ operators have bag semantics.)

For instance, we can define a view $ClerkCust$ to obtain the sum of the purchases made by a customer from some clerk. Furthermore, $ClerkCust$ only considers old customers that placed an order recently for an expensive item. The definition query of $ClerkCust$ is as follows.

$$\pi_{O.clerk, C.custId, \texttt{SUM}(L.qty * L.cost) \texttt{ as } sum, \texttt{COUNT}() \texttt{ as } cnt}$$
$$\sigma_{L.cost > 99 \wedge C.custId < 500 \wedge O.ordId > 1000 \wedge L.ordId = O.ordId \wedge O.custId = C.custId} (C \times O \times L).$$

In general, the project specification $\mathcal{A}$ of a definition query is a set of attributes and aggregate functions (*e.g.*, `SUM`). If $\mathcal{A}$ contains aggregate functions, any element in $\mathcal{A}$ that is not an aggregate function is a grouping attribute (*e.g.*, $C.custId$). Condition $\mathcal{P}$ is a conjunction of atomic conditions, like join condition $O.custId = C.custId$, and selection condition $O.ordId > 1000$. Finally, $\mathcal{R}$ is a set of tables (*i.e.*, no self-joins); each table is either a base relation or a view.

A view V needs to be maintained when there are insertions and deletions to the tables that V is defined on. For instance, let us assume that *Def*(V) is $\sigma_{S.b=T.c}(S \times T)$. We assume that insertions and deletions to table S are stored in *delta relation* tables denoted $\triangle S$ and $\bigtriangledown S$ respectively. To compute the insertions to V (*i.e.*, $\triangle V$), the *maintenance query* given by Query (1) is used. The deletions to V (*i.e.*, $\bigtriangledown V$) are computed using Query (2). These queries use the *pre-state* of S and T, *i.e.*, before the insertions, and then the deletions, are applied. Other queries may be used if updates are applied differently, but they should still have the same form. We use *Maint*(V) to denote the set of maintenance queries for computing the insertions to and deletions from V.

$$\sigma_{\triangle S.b=T.c}(\triangle S \times T) \;\cup\; \sigma_{S.b=\triangle T.c}(S \times \triangle T) \;\cup\; \sigma_{\triangle S.b=\triangle T.c}(\triangle S \times \triangle T) \;\cup\; \sigma_{\bigtriangledown S.b=\bigtriangledown T.c}(\bigtriangledown S \times \bigtriangledown T) \quad (1)$$

$$\sigma_{\bigtriangledown S.b=T.c}(\bigtriangledown S \times T) \;\cup\; \sigma_{S.b=\bigtriangledown T.c}(S \times \bigtriangledown T) \;\cup\; \sigma_{\bigtriangledown S.b=\triangle T.c}(\bigtriangledown S \times \triangle T) \;\cup\; \sigma_{\triangle S.b=\bigtriangledown T.c}(\triangle S \times \bigtriangledown T) \quad (2)$$

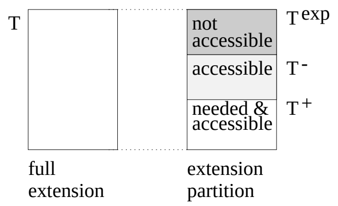

Figure 2: Extension Partition of T

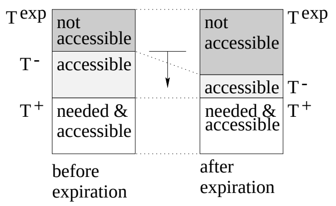

Figure 3: Effect of Expiration on T^- and T^{exp}

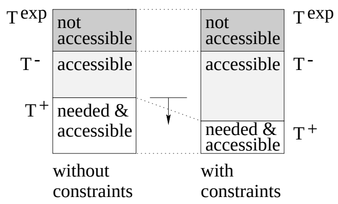

Figure 4: Effect of Constraints on T^+ and T^-

Expiration: A user may issue an *expiration request* of the form $\sigma_{\mathcal{P}}(T)$ on any base relation or view T. This requests that all the T tuples in $\sigma_{\mathcal{P}}(T)$ be removed from T's extension. Once a tuple is expired, it can no longer be accessed by any query. However, in our framework, we only expire $\sigma_{\mathcal{P}}(T)$ tuples that are not "needed" (later defined formally) by maintenance queries. Conceptually, we partition the extension of each base relation or view T into T^+, T^-, and T^{exp}, as shown in Figure 2. The tuples in T^+ are accessible to any query and are needed by maintenance queries. The tuples in T^- are accessible to any query but are not needed by maintenance queries. The tuples in T^{exp} are expired, are not accessible, and are not needed by maintenance queries. The tuples in T^+ and T^- comprise T's *real extension*, which is the extension kept persistently. The tuples in T^+, T^-, and T^{exp} comprise T's *full extension*. (The full extension of T is referred to in queries simply as "T".) The conceptual partitions T^+ and T^- are realized in T's real extension by keeping a boolean attribute *needed* for each tuple. The *needed* attribute of a tuple t is set to `true` if $t \in T^+$ and `false` otherwise. Given an expiration request $\sigma_{\mathcal{P}}(T)$, conceptually the request is satisfied by removing $\sigma_{\mathcal{P}}(T^-)$ from T^- and "moving" them to T^{exp}, as depicted in Figure 3. We assume that for any two consecutive expiration requests on T, denoted $\sigma_{\mathcal{P}_i}(T)$ and $\sigma_{\mathcal{P}_j}(T)$, the subsequent request asks for more tuples to be expired than the earlier one (*i.e.*, $\mathcal{P}_i$ implies $\mathcal{P}_j$). This is guaranteed by keeping the most recent expiration request on T in $LastReq(T) = \sigma_{\mathcal{P}'}(T)$. When a new expiration request $\sigma_{\mathcal{P}}(T)$ is issued, the request is modified as $\sigma_{\mathcal{P}\vee\mathcal{P}'}(T)$ and $LastReq(T)$ is set to $\sigma_{\mathcal{P}\vee\mathcal{P}'}(T)$.[1]

Effect of Expiration on Queries: Although all queries (user queries, maintenance queries and definition queries) are formulated in terms of full extensions, only the tuples in the real extensions can be used in answering the query. Conceptually, the answer returned for Q is the answer for the "query" $Access(Q)$, which is the same as Q but with each T referred to in Q replaced by $T^+ \cup T^-$. Similarly, the complete answer to Q is the answer returned for the "query" $Complete(Q)$, which is the same as Q but with each T referred to in Q replaced by $T^+ \cup T^- \cup T^{exp}$ (*i.e.*, suppose that tuples in T^{exp} are accessible to $Complete(Q)$). We say the answer to Q is *complete* if the answer to $Access(Q)$ is the same as the answer to $Complete(Q)$. Otherwise, the answer is *incomplete*. We say that a tuple $t \in T$ (*i.e.*, $t \in (T^+ \cup T^- \cup T^{exp})$) is *needed* in answering Q if the answer to $Complete(Q)$ is different depending on whether t is removed from T's extension or not.[2]

Since we guarantee that only tuples not needed by maintenance queries can be expired, the answer to any maintenance query Q is always complete. On the other hand, the answer to a user or definition query Q may be incomplete. In case of a user query, a query Q', where $Access(Q) = Complete(Q')$, is returned in addition to Q's incomplete answer. Q' is used to help describe the incomplete answer returned. Incidentally, we believe that many database systems return incomplete answers, because databases cannot hold all possible data. However, in current systems, users are simply not told about missing data. We think returning descriptive information like Q' is an improvement. In case of a definition query $Q = Def(V)$, if the answer to Q is incomplete, V is not initialized and a query Q', where $Access(Q) = Complete(Q')$, is returned as an alternative definition query for V. Note that for both user and definition queries, it may be possible to obtain more answer tuples by accessing not only the views referred to in the query, but also the underlying tables these views are defined on. Such an extension is feasible in our framework, but it is not considered in this paper.

Constraints: To help decrease the number of tuples that are deemed needed (see Figure 4), we may associate with each table T a set of constraints, $Constraints(T)$, that describe in some language (Section 4) the contents of the delta relations $\triangle T$ and $\triangledown T$. The constraints of base relations are provided by the administrator based on his knowledge of the application (*e.g.*, "table O is append-only"). The constraints of a view V are computed from the constraints of the tables that V is defined on. We do *not* assume that the input constraints characterize the application completely. We only assume that the administrator inputs constraints that he knows are implied by the application. In the worst case, the administrator may not know any guarantees on the delta

[1] Algorithms for removing redundant conditions in $\mathcal{P} \vee \mathcal{P}'$ can certainly be employed.

[2] This definition of needed works for aggregate views since we require the `COUNT` aggregate function to be included. This is reasonable because `COUNT` is helpful in maintaining views with `AVG`, `SUM`, `MAX` or `MIN` ([Qua96]).

relations and may set *Constraints*(T) to be empty.

Framework Summary: Table 1 gives a summary of the concepts used in the framework. Henceforth, we denote the set of all tables as $\mathcal{T}$, the set of all constraints as $\mathcal{C}$ (*i.e.*$\bigcup_{T \in \mathcal{T}}$ *Constraints*(T)), and the set of all maintenance queries as $\mathcal{E}$ (*i.e.*, $\bigcup_{\text{view } V \in \mathcal{T}}$ *Maint*(V)).

base relation T	1. real extension ($T^{+} \cup T^{-}$); 2. full extension ($T^{+} \cup T^{-} \cup T^{exp}$); 3. *Constraints*$(T)$; 4. *LastReq*$(T)$
view T	1. real extension ($T^{+} \cup T^{-}$); 2. full extension ($T^{+} \cup T^{-} \cup T^{exp}$); 3. *Constraints*$(T)$; 4. *Def*$(T)$; 5. *Maint*$(T)$; 6. *LastReq*$(T)$
delta relation $\triangle T$	extension (with no conceptual partitions) containing insertions to T
delta relation $\bigtriangledown T$	extension (with no conceptual partitions) containing deletions from T
expiration request $\sigma_{\mathcal{P}}(T)$	satisfied by removing $\sigma_{\mathcal{P}}(T^{-})$ from T's real extension
query Q	refers to full extensions (*e.g.*, as "T") only and never partitions
user query Q	1. cannot refer to delta relations; 2. if answer is incomplete, Q' (*Access*(Q) = *Complete*(Q')) is returned to describe incomplete answer
definition query Q	1. cannot refer to delta relations; 2. if answer is incomplete, Q' (*Access*(Q) = *Complete*(Q')) is returned as alternative definition
maintenance query Q	1. can refer to delta relations; 2. answer is always complete
$\mathcal{T}$	set of all warehouse tables
$\mathcal{C}$	$\bigcup_{T \in \mathcal{T}}$ *Constraints*(T)
$\mathcal{E}$	$\bigcup_{\text{view } V \in \mathcal{T}}$ *Maint*(V)

Table 1: Summary of Framework

Problems: There are several problems that need to be solved to implement our framework:

1. **Initial Extension Marking:** Given an initial configuration of tables $\mathcal{T}$ where none of the tables have any expired tuples yet, we must identify and mark which tuples are needed by the maintenance queries $\mathcal{E}$ by setting the *needed* attribute of these tuples to `true`.
2. **Initial Extension Marking With Constraints:** This problem is the same as (1) but in addition, we are also given a set of constraints $\mathcal{C}$, which can potentially decrease the number of tuples whose *needed* attribute is set.
3. **Constraints of Views:** In solving the first two problems, we must compute the constraints of each view $V \in \mathcal{T}$ from the constraints of underlying tables.
4. **Incomplete Answers:** For each user query Q, we must determine if the answer to Q is complete. If not, we must determine a modified query Q' whose complete answer is the same as the incomplete answer returned for Q.
5. **Changes to $\mathcal{T}$:** When a new view V is being added to the initial configuration of tables $\mathcal{T}$, we must determine if the answer to $Q = Def(V)$ is complete. Techniques for (4) apply here. If the answer to Q is not complete, we must determine a modified view definition query Q' as a suggested alternative definition query. Once $Def(V)$ has a complete answer, for each table T that V is defined on, we must determine which tuples are now needed because of the addition of V, and mark these tuples appropriately.

6. **Changes To $\mathcal{C}$:** If the constraints are changed to expire more tuples, we must determine the effects of the change on the extension marking of each table T.
7. **Insertions:** If there are insertions $\triangle T$ to a table T, we must determine the *needed* attribute value of each tuple inserted. (There is no problem with deletions.)

Note that the first two problems need to be solved once, when the initial configuration is given. Hence, efficiency is not at a premium. The third, fifth and sixth problems are also solved infrequently. On the other hand, the fourth and seventh problems are solved fairly frequently and require reasonably efficient solutions. In the rest of the paper, Section 3 is devoted to the first problem; Section 4 is devoted to the second problem; and Section 6 is devoted to the last three problems. The algorithms developed are reasonably efficient. Due to space constraints, we do not present our solution to the third problem. That is, for this paper, we assume that the administrator provides not only the constraints of the base relations but also the constraints of the views. We also do not present our solution to the fourth problem. Our preliminary solutions to the third and fourth problems, which build on previous work in [Lev96], appear in [GML97].

3 Extension Marking

In this section, we assume we are given an initial configuration $\mathcal{T}$ (base relations and views) and none of tables have any expired tuples yet. For each table $T \in \mathcal{T}$, we identify which T tuples are needed by maintenance queries. We mark the needed tuples by setting the *needed* attribute.

As mentioned earlier, this marking is done only when the initial configuration is submitted and not for each expiration request. Once the marking is done, any subsequent expiration request $\sigma_{\mathcal{P}}(T)$ is satisfied very efficiently by removing the tuples $\sigma_{\mathcal{P} \wedge needed=\texttt{false}}(T)$ from T's real extension.

Before we present how the needed tuples are identified, we introduce *maintenance expressions*, which are the subqueries of the maintenance queries that we work with. For instance, suppose we have a view V whose definition query is of the form $\pi_{\mathcal{A}} \sigma_{\mathcal{P}}(\times_{R \in \mathcal{R}} R)$, where $\mathcal{A}$ does not have any aggregate functions. The maintenance queries (*e.g.*, Queries (1) and (2)) of V are of the form

$$\bigcup_i \pi_{\mathcal{A}_i} \sigma_{\mathcal{P}_i}(\times_{R \in \mathcal{R}_i} R),$$

where $\mathcal{R}_i$ may include delta relations. We call each subquery $\pi_{\mathcal{A}_i} \sigma_{\mathcal{P}_i}(\times_{R \in \mathcal{R}_i} R)$ a *maintenance expression*. Notice that if a tuple is needed by some maintenance expression, it is needed by some maintenance query. Also, if a tuple is not needed by any maintenance expression, it is not needed by any maintenance query. In Appendix B, we show that the maintenance queries of views that use aggregates can also be decomposed into maintenance expressions. (Note that our example view $ClerkCust$ has aggregates.) Henceforth, we use $\mathcal{E}$ for the maintenance expressions of $\mathcal{T}$.

We now present a lemma that defines a function $\texttt{Needed}(T, \mathcal{E})$ and identifies using this function, all and only the T tuples that are needed by the maintenance expressions in $\mathcal{E}$. We refer to the following functions in the lemma: `Closure`, `Ignore`, and `Map`.

Function $\texttt{Closure}(\mathcal{P})$ returns the closure of the input conjunctive condition ([Ull89]). For instance, if $\mathcal{P}$ is $R.a > S.b \wedge S.b > T.c$, $\texttt{Closure}(\mathcal{P})$ returns $R.a > S.b \wedge S.b > T.c \wedge R.a > T.c$. (`Closure` is an $O(n^3)$ operation, where n is the number of distinct attributes in $\mathcal{P}$.)

Function $\texttt{Ignore}(\mathcal{P}, \mathcal{T})$ modifies the conjunctive condition $\mathcal{P}$ by replacing any atomic condition that uses an attribute of a table in $\mathcal{T}$ with $\texttt{true}$. For instance, if $\mathcal{P}$ is $R.a > S.b \wedge S.b > T.c$, $\texttt{Ignore}(\mathcal{P}, \{S\})$ is $\texttt{true} \wedge \texttt{true}$ or simply $\texttt{true}$. Notice that $\texttt{Ignore}(\texttt{Closure}(\mathcal{P}), \{S\})$ is $R.a > T.c$.

Finally, function $\texttt{Map}$ acts on a maintenance expression $E = \pi_{\mathcal{A}} \sigma_{\mathcal{P}}(\times_{R \in \mathcal{R}} R)$, where there is a set $\mathcal{D} \subseteq \mathcal{R}$ of delta relations (possibly empty) involved in E's cross product. Function $\texttt{Map}(E, T)$ is defined as follows.

$$\texttt{Map}(E = \pi_{\mathcal{A}} \sigma_{\mathcal{P}}(\times_{R \in \mathcal{R}} R), T) = \begin{cases} \{\} & \text{if } T \notin \mathcal{R} \\ \pi_{\texttt{Attrs}(T)} \sigma_{\texttt{Ignore}(\texttt{Closure}(\mathcal{P}), (\mathcal{D} - \{T\}))} (\times_{R \in (\mathcal{R} - (\mathcal{D} - \{T\}))} R) & \text{otherwise} \end{cases}$$

That is, if T is not referred to in E, $\texttt{Map}$ returns $\{\}$. This is the common case since most maintenance expressions do not refer to a specific table T. If T is referred to in E, $\texttt{Map}$ returns a new expression obtained by first removing the delta relations in $\mathcal{D}$ from the cross product (except T if T is a delta relation). Then, the closure of the condition $\mathcal{P}$ is computed. Then, $\mathcal{P}$ is modified to ignore any atomic condition that refers to any delta relation (except T if T is a delta relation). Finally, the projected attributes is changed to $\texttt{Attrs}(T)$, the attributes of the table T.

Lemma 3.1 *Given a table T and a set of maintenance expression $\mathcal{E}$,* $\texttt{Needed}$*(T, E) is defined as*

$$\bigcup_{E \in \mathcal{E}} \texttt{Map}(E, T).$$

The query $T \ltimes_{\texttt{Attrs}(T)} \texttt{Needed}(T, \mathcal{E})$ returns all and only the tuples in T that are needed by the maintenance expressions in $\mathcal{E}$. □

Note that $\texttt{Needed}$ may list a needed tuple $t \in T$ more times than t appears in T. (This is illustrated in the next example.) Hence, the semijoin ($\ltimes$) operation, which is equivalent to an $\texttt{exists}$ condition (*e.g.*, SQL EXISTS condition), is used to obtain the T tuples needed for $\mathcal{E}$. The proof of Lemma 3.1 is in Appendix A. We give the intuition behind the proof in the next example.

EXAMPLE 3.1 Suppose we are given one of the maintenance expressions of $ClerkCust$ as the maintenance expression E in question.

$$E = \pi_{\triangle O.clerk, C.custId, L.qty, L.cost} \\ \sigma_{L.cost > 99 \ \wedge \ C.custId < 500 \ \wedge \ \triangle O.ordId > 1000 \ \wedge \ L.ordId = \triangle O.ordId \ \wedge \ \triangle O.custId = C.custId} (C \times \triangle O \times L)$$

Let us consider what L tuples are needed by E. We claim that $\texttt{Map}(E, L)$, shown below, identifies all these L tuples.

$$\pi_{\texttt{Attrs}(L)} \sigma_{L.cost > 99 \ \wedge \ C.custId < 500 \ \wedge \ L.ordId > 1000} (C \times L)$$

Notice that $\texttt{Map}(E, L)$ excludes $\triangle O$ from the cross product and consequently ignores all the atomic conditions in E that refer to $\triangle O$ attributes. Intuitively, this means that we cannot say that an L tuple t_L is not needed even if there does not exist a $\triangle O$ tuple that t_L can join with. This is reasonable because although t_L may not join with any of the current insertions to O (*i.e.*, current extension of $\triangle O$), it may join with future insertions (*i.e.*, extension of $\triangle O$ at some later point in time). We can only set $t_L.needed$ to $\texttt{false}$ if for *any* $\triangle O$, t_L only joins with $\triangle O$ tuples that are not needed themselves. For instance, any $\triangle O$ tuple that has an $ordId$ less than or equal to 1000 is not needed in answering E. Since there is an atomic condition $L.ordId = \triangle O.ordId$ in E, any L tuple that has an *ordId* less than or equal to 1000 is also not needed in answering E. This illustrates the need for computing the closure of the atomic conditions before ignoring the atomic conditions that use delta relation attributes. Thus, in our example, $\texttt{Map}(E, L)$ has the atomic condition $L.ordId > 1000$.

While $\texttt{Map}(E, L)$ identifies all the needed L tuples, it may list an L tuple t_L more times than t_L appears in L. For instance, $\texttt{Map}(E, L)$ performs a cross product between C and L without applying any conditions between them. Hence, $\texttt{Map}(E, L)$ lists t_L as many times as there are C tuples. Thus, to obtain the correct bag of tuples, the query $L \ltimes_{\texttt{Attrs}(L)} \texttt{Map}(E, L)$ is used. □

The example illustrated that $\texttt{Map}(E, T)$ may perform cross products. Cross products can be easily avoided by constructing a join graph for E, whose nodes represent the tables in E. An edge between tables R and S is in the E's join graph if there is an atomic condition in E that uses both R and S attributes. Given E's join graph, $\texttt{Map}(E, T)$ can be modified as follows. If a table R is not reachable from T, remove R from the cross product and ignore all the atomic conditions that refer to R attributes. This simple procedure can be used to avoid all cross products.

4 Extension Marking With Constraints

Given a set of tables $\mathcal{T}$, maintenance expressions $\mathcal{E}$, and now a set of constraints $\mathcal{C}$, our goal is to mark the tuples that are needed by the maintenance expressions. The constraints may lead to a decrease of the number of needed tuples.

Marking tuples entails solving two problems. First, the maintenance expressions in $\mathcal{E}$ need to be modified using $\mathcal{C}$ to produce a new set of expressions $\mathcal{E}_\mathcal{C}$. Second, the function $\texttt{Needed}(T, \mathcal{E})$ needs to be modified to $\texttt{Needed}_\mathcal{C}(T, \mathcal{E}_\mathcal{C})$ that acts on the new set of maintenance expressions. `Needed` is not adequate because it assumes a maintenance expression of the form $\pi_\mathcal{A}\sigma_\mathcal{P}(\times_{R\in\mathcal{R}} R)$, which is devoid of `exists` and `not exists` conditions (expressed using the $\ltimes$ and $\overline{\ltimes}$ operators). Unfortunately, the expressions in $\mathcal{E}_\mathcal{C}$ may contain such conditions.

Before we solve these two problems, we present a simple constraint language *CL* for specifying the constraints in $\mathcal{C}$. In Section 4.2, we give the algorithm that uses $\mathcal{C}$ for producing $\mathcal{E}_\mathcal{C}$ from $\mathcal{E}$. We present in Section 4.3 the function $\texttt{Needed}_\mathcal{C}$ that acts on $\mathcal{E}_\mathcal{C}$. We illustrate in Section 5 that $\texttt{Needed}_\mathcal{C}$ may return a much smaller bag of tuples compared to `Needed`.

4.1 Constraint Language

A *CL* constraint is an equivalence conforming to one of the two forms shown below, where each R and T is either a base relation, a delta relation or a view.

$$\sigma_{\mathcal{P}_{LHS}}(\times_{R\in\mathcal{R}} R) \equiv \sigma_{\mathcal{P}_{RHS}}(\times_{R\in\mathcal{R}} R) \ltimes T \quad \text{or} \quad \sigma_{\mathcal{P}_{LHS}}(\times_{R\in\mathcal{R}} R) \equiv \sigma_{\mathcal{P}_{RHS}}(\times_{R\in\mathcal{R}} R) \overline{\ltimes} T$$

A *CL* constraint c states that the query on c's left hand side is guaranteed to return the same bag of tuples as the query on c's right hand side. We denote the query on the right hand side and the left hand side of a constraint c as $RHS(c)$ and $LHS(c)$, respectively. In any constraint c, the conditions in $RHS(c)$ logically imply the conditions in $LHS(c)$ (*i.e.*, $\mathcal{P}_{RHS} \Rightarrow \mathcal{P}_{LHS}$). Also, `exists` or `not exists` (*i.e.*, $\overline{\ltimes}$ operator) conditions can be introduced in $RHS(c)$. Even though $RHS(c)$ has more conditions than $LHS(c)$, constraint c states that the two queries are equivalent.

In the discussion, we often refer to a constraint c of the form $R \equiv \sigma_{\mathcal{P}_{RHS}}(R) \ltimes T$ (or $\overline{\ltimes} T$) as *context-free*, since R can be substituted by $RHS(c)$ in any query that R is in. More general constraints that have selection or join conditions on the left hand side are called *context-sensitive*.

CL can express many constraints that occur in warehousing applications. For instance, it can express equality generating dependencies (*e.g.*, functional dependencies, key constraints) and many tuple generating dependencies (*e.g.*, inclusion dependencies, referential integrity constraints). In addition to these conventional database constraints, *CL* can also express "semantic" constraints [SO89] such as "transition" constraints [NY82] and implication constraints. Examples of these constraints are append-only constraints and ad hoc constraints like "`Clerk1` handles `CustA`". *CL* cannot however express join dependencies and extending *CL* to handle these dependencies makes the algorithms we introduce later very complex. Even with this deficiency, we believe *CL* is expressive enough to capture many constraints that occur in practice as illustrated next. Furthermore, we will see that *CL*'s syntax is particularly well suited for modifying maintenance expressions.

EXAMPLE 4.1 We give the *CL* constraints which an administrator may input because they are implied by the scenario in Example 1.1. Note that most of the constraints are context-free.

Append-only constraints: We alluded in Example 1.1 that O is append-only. That is, no tuple is ever deleted from O and every inserted O tuple has an *ordId* value greater than the maximum *ordId* value so far. The append-only behavior of O is captured by Constraint (3), which states that $\bigtriangledown O$ is always empty, and by Constraint (4), which states that the *ordId* values of the inserted O tuples are greater than the maximum *ordId* value so far.

$$\bigtriangledown O \equiv \sigma_{\texttt{false}}(\bigtriangledown O) \tag{3}$$

$$\bigtriangleup O \equiv \bigtriangleup O \overline{\ltimes}_{\bigtriangleup O.ordId \leq O.ordId} O \tag{4}$$

L also has an append-only behavior which is captured in Constraints (5), (6) and (7). Intuitively, insertions to L represent new line items of the most recent order (O tuple with maximum *ordId*) or of new incoming orders ($\bigtriangleup O$ tuples). Constraints (6) and (7) are used to describe the insertions to L. That is, inserted L tuples that join with $\bigtriangleup O$ have *ordId* values greater than the maximum *ordId*. Inserted L tuples that join with O have *ordId* values equal to the maximum *ordId*.

$$\bigtriangledown L \equiv \sigma_{\texttt{false}}(\bigtriangledown L) \tag{5}$$

$$\sigma_{\bigtriangleup O.ordId=\bigtriangleup L.ordId}(\bigtriangleup O \times \bigtriangleup L) \equiv \sigma_{\bigtriangleup O.ordId=\bigtriangleup L.ordId}(\bigtriangleup O \times (\bigtriangleup L \overline{\ltimes}_{\bigtriangleup L.ordId \leq O.ordId} O)) \tag{6}$$

$$\sigma_{O.ordId=\bigtriangleup L.ordId}(O \times \bigtriangleup L) \equiv \sigma_{O.ordId=\bigtriangleup L.ordId}(O \times (\bigtriangleup L \overline{\ltimes}_{\bigtriangleup L.ordId < O.ordId} O)) \tag{7}$$

Key constraints: The schema in Example 1.1 assumes that *custId* is the key of C. The constraints below are implied by this key constraint. Constraints (8) and (9), which use the table renaming operator ρ, enforce the functional dependency implied by the key constraint. Finally, Constraint (10) enforces that none of the keys of the inserted tuples are in C. Similar constraints are implied by the assumptions that *ordId* is the key of O and both *ordId* and *partId* make up the key of L.

$$C \equiv C \; \overline{\ltimes}_{(C.custId=C'.custId) \; \wedge \; (C.info \neq C'.info)} \; \rho_{C'}(C) \tag{8}$$

$$\bigtriangledown C \equiv \bigtriangledown C \; \overline{\ltimes}_{(\bigtriangledown C.custId=\bigtriangledown C'.custId) \; \wedge \; (\bigtriangledown C.info \neq \bigtriangledown C'.info)} \; \rho_{\bigtriangledown C'}(\bigtriangledown C) \tag{9}$$

$$\bigtriangleup C \equiv \bigtriangleup C \; \overline{\ltimes}_{\bigtriangleup C.custId=C.custId} \; C \tag{10}$$

Referential integrity constraints: Given the schema introduced in Example 1.1, it is reasonable to assume that there is a referential integrity constraint from attribute $O.custId$ to key $C.custId$. The following constraints express this assumption. Similar constraints are used to express a referential integrity constraint from attribute $L.ordId$ to key $O.ordId$.

$$O \equiv O \ltimes_{O.custId=C.custId} C \tag{11}$$

$$\triangle O \equiv \triangle O \ltimes_{\triangle O.custId=C.custId} C \tag{12}$$

$$\triangledown O \equiv \triangledown O \ltimes_{\triangledown O.custId=C.custId} C \tag{13}$$

Weak minimality constraints: It is also reasonable to assume that deletions from C are *weakly minimal* [GL95]. That is, all the deleted C tuples were previously in C.

$$\triangledown C \equiv \triangledown C \ltimes_{(\triangledown C.custId=C.partId) \wedge (\triangledown C.info=C.info)} C \tag{14}$$

Ad hoc constraints: Finally, we illustrate that CL can be used to express fairly ad hoc constraints. For instance, the constraint $\sigma_{custId<1000}(O) \equiv \sigma_{(custId<1000) \wedge (clerk=\text{``}\mathtt{Clerk1}\text{''})}(O)$, expresses that customers with $custId < 1000$ are handled by `Clerk1`. □

4.2 Modifying Maintenance Expressions

Given a maintenance expression E, we now modify E by applying a given set of CL constraints to it. Intuitively, since $LHS(c)$ and $RHS(c)$ of a CL constraint c are equivalent, whenever $LHS(c)$ "matches" a subquery of E, we can substitute $RHS(c)$ for $LHS(c)$ in E. We say a constraint c is *applied* to E when we successfully match $LHS(c)$ to a subquery of E and replace the matching subquery with $RHS(c)$. The challenge is of course in determining whether $LHS(c)$ matches some subquery of E since a syntactic check does not suffice. For instance, if E is $\sigma_{a>10}(\triangle R)$ and $LHS(c)$ is $\sigma_{a>5}(\triangle R)$, $LHS(c)$ matches a subquery of E since E can be rewritten as $\sigma_{a>10}(\sigma_{a>5}(\triangle R))$. The next example provides additional illustration of how a constraint is applied.

EXAMPLE 4.2 Most of the constraints in Example 4.1 are context-free and applying them is trivial. For instance, applying Constraint (3) (*i.e.*, $\triangledown O \equiv \sigma_{\mathtt{false}}(\triangledown O)$) simply requires finding occurrences of $\triangledown O$ in a maintenance expression E and replacing it with $\sigma_{\mathtt{false}}(\triangledown O)$. Since E has a conjunctive condition that includes `false`, E is guaranteed to result in an empty answer.

To make the current example more interesting, let us consider applying the context-sensitive constraint c (*i.e.*, Constraint (7), Example 4.1)

$$\sigma_{O.ordId=\triangle L.ordId}(O \times \triangle L) \equiv \sigma_{O.ordId=\triangle L.ordId}(O \times (\triangle L \overline{\ltimes}_{L.ordId<O.ordId} O)),$$

to the following maintenance expression E of the $ClerkCust$ view.

$$\pi_{O.clerk,C.custId,\triangle L.qty,\triangle L.cost}$$
$$\sigma_{\triangle L.cost>99 \wedge C.custId<500 \wedge O.ordId>1000 \wedge O.ordId=\triangle L.ordId \wedge O.custId=C.custId}(C \times O \times \triangle L)$$

The previous maintenance expression can be rewritten as

$$\pi_{O.clerk,C.custId,\triangle L.qty,\triangle L.cost}$$
$$\sigma_{\triangle L.cost>99 \wedge C.custId<500 \wedge O.ordId>1000 \wedge O.custId=C.custId}(C \times \sigma_{O.ordId=\triangle L.ordId}(O \times \triangle L)).$$

Clearly $LHS(c)$ matches a subquery of E. Hence, we can replace the matching subquery with $RHS(c)$, yielding the following maintenance expression.

$$\pi_{O.clerk, C.custId, \triangle L.qty, \triangle L.cost}$$
$$\sigma_{\triangle L.cost>99 \ \wedge \ C.custId<500 \ \wedge \ O.ordId>1000 \ \wedge \ O.ordId=\triangle L.ordId \ \wedge \ O.custId=C.custId}$$
$$(C \times O \times (\triangle L \ \overline{\triangleright\!\!<}_{L.ordId<O.ordId} \ O)) \qquad \Box$$

The previous example illustrated algorithm *Apply* (Algorithm 4.1, Figure 5) for applying a constraint c on a maintenance expression E. *Apply* first checks if the tables in $LHS(c)$ are also in E (Line 1).[3] It then checks if the conditions in E imply the conditions in $LHS(c)$ (Line 2). This can be done efficiently because the conditions involved are conjunctive [Ull89].[4] If both checks are passed, then $LHS(c)$ matches a subquery of E. For instance, suppose that E is

$$\pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R) \triangleright\!\!< S \ldots \overline{\triangleright\!\!<} T \ldots,$$

and $LHS(c)$ is $\sigma_{\mathcal{P}_{LHS}}(\times_{U\in\mathcal{U}}U)$. If $\mathcal{U} \subseteq \mathcal{R}$ and $\mathcal{P} \Rightarrow \mathcal{P}_{LHS}$, it is guaranteed that E is equivalent to

$$\pi_{\mathcal{A}}\sigma_{\mathcal{P}}((\times_{R\in(\mathcal{R}-\mathcal{U})}R) \times \sigma_{\mathcal{P}_{LHS}}(\times_{U\in\mathcal{U}}U)) \triangleright\!\!< S \ldots \overline{\triangleright\!\!<} T \ldots .$$

The subquery of E that matches $LHS(c)$ can then be replaced by $RHS(c)$. Redundant conditions are eliminated in Line 3 of *Apply* by solving another implication problem. Finally, any conditions added are pulled out of the cross product to facilitate the application of other constraints.

Although *Apply* always modifies E to an equivalent expression, it is not complete since it may not apply a constraint even when equivalence is preserved. This is because Line 2 only takes into account the selection and join conditions in $\mathcal{P}$, but not the `exists` and `not exists` conditions given by the $\triangleright\!\!<$ and $\overline{\triangleright\!\!<}$ operators. (`Exists` conditions can be handled but it is not shown in *Apply*.) To obtain a complete algorithm, the implication problem $\mathcal{P}' \Rightarrow \mathcal{P}_{LHS}$ must be solved, where $\mathcal{P}'$ is the conjunction of all the selection, join, `exists` and `not exists` conditions. Unfortunately, there are no known complete algorithms to solve the general implication problem with a mixture of existential and universal quantifiers ([YL87]).

In Section 4.3, we develop an algorithm to compute the closure of a conjunctive condition which may include `exists` conditions but only atomic `not exists` conditions. This algorithm can be useful in solving a more general implication problem than the one in Line 2. However, we do not show it here since taking into account `exists` and `not exists` conditions is not critical in *Apply*. This is because in practice, many constraints are context-free and can be applied easily. Context-sensitive constraints, like the append-only and implication constraints in Example 4.1, usually only require examining the selection and join conditions of E.

So far, we have discussed how a single constraint is applied to E. When there is a set of constraints to be applied, the order of application does not matter. More specifically, applying a constraint c_1 to E before c_2 does not jeopardize the "applicability" of c_2 because applying c_1 only adds conditions to E. On the other hand, if initially c_2 cannot be applied, applying c_1 may add enough conditions to E so that c_2 can now be applied. Thus, after a constraint is applied, we must

[3] This check suffices since we only handle view definitions with no self-joins. Otherwise, all possible mappings from the tables in c to those in E have to be checked.

[4] It can be done in $O(n^3)$ time, where n is the number of distinct attributes in the conditions. This assumes that the cardinality of the domain of the attributes is greater than or equal to n to handle $\neq$'s.

Algorithm 4.1 *Apply*
Input: maintenance expression E, CL constraint c
Output: `true` if c is applied, `false` otherwise **Side effect:** may modify E

Let E be of the form: $\pi_{\mathcal{A}}(\sigma_{\mathcal{P}}(\times_{R \in \mathcal{R}} R)\ltimes S \ldots \overline{\ltimes} T \ldots)$
Let c be of the form: $\sigma_{\mathcal{P}_{LHS}}(\times_{U \in \mathcal{U}} U) \equiv \sigma_{\mathcal{P}_{RHS}}(\times_{U \in \mathcal{U}} U)\ltimes V$ (or $\overline{\ltimes} V$)

1. If $\mathcal{U} \subseteq \mathcal{R}$
    2. If $\mathcal{P} \Rightarrow \mathcal{P}_{LHS}$
        3. Remove any conditions in $\mathcal{P}$ that are implied by $\mathcal{P}_{RHS}$
        4. $E \leftarrow \pi_{\mathcal{A}}(\sigma_{\mathcal{P} \wedge \mathcal{P}_{RHS}}(\times_{R \in \mathcal{R}} R))\ltimes S \ldots \ltimes V \ldots \overline{\ltimes} T \ldots$
        5. Return `true`
6. Return `false` $\diamond$

Algorithm 4.2 *Modify*
Input: maintenance expression E, a set of CL constraints $\mathcal{C}$
Side effect: may modify expression E

1. *change* $\leftarrow$ `true`
2. While (*change* $=$ `true`)
    3. *change* $\leftarrow$ `false`
    4. For (each constraint c in $\mathcal{C}$)
        5. If (*Apply*$(E, c) =$ `true`)
            6. Remove c from $\mathcal{C}$, *change* $\leftarrow$ `true` $\diamond$

Figure 5: Algorithm For Modifying a Maintenance Expression

check if any of the unapplied constraints can be applied. Also note that any constraint can only be applied once and it can only match one subquery of E since E has no self-joins.

Algorithm 4.2 (Figure 5) shows the algorithm *Modify* for applying a set of constraints $\mathcal{C}$ to E. Although efficiency is not at a premium when marking extensions, *Modify* has a tolerable overall complexity of $O(|\mathcal{C}|^2 \cdot n^3)$, assuming the check in Line 1 of *Apply* is done in constant time. $|\mathcal{C}|$ is the number of constraints and n is the number of distinct attributes used in $\mathcal{P}$ of E.

4.3 Deriving `Needed`$_{\mathcal{C}}$

Given the maintenance expressions $\mathcal{E}$, we can use *Modify* to alter each expression in $\mathcal{E}$ based on $\mathcal{C}$, and produce a new set of expressions $\mathcal{E}_{\mathcal{C}}$. In this section, we first discuss why using `Needed` on $\mathcal{E}_{\mathcal{C}}$ is not satisfactory. A function that identifies all and only the tuples needed by $\mathcal{E}_{\mathcal{C}}$ is complex since it needs to solve hard problems, *e.g.*, closure of a non-conjunctive condition. Thus, in this section, we develop a fairly efficient `Needed`$_{\mathcal{C}}$ function which handles `exists` and some `not exists` conditions, namely, those composed of one or a disjunction of atomic conditions. In the latter part of the section, we give a lemma that formally describes the properties of `Needed`$_{\mathcal{C}}$.

Problem with `Needed`: Strictly speaking, `Needed` was not defined to work with maintenance expressions with `exists` and `not exists` conditions. Nevertheless, function `Needed`$(T, \mathcal{E}_{\mathcal{C}})$ can be adapted to apply to $\mathcal{E}_{\mathcal{C}}$ by modifying `Map`(E, T) to return the following query for each $E \in \mathcal{E}_{\mathcal{C}}$.

$$\pi_{\texttt{Attrs}(T)}\sigma_{\texttt{Ignore}(\texttt{Closure}(\mathcal{P}),(\mathcal{D}-\{T\}))}(\times_{R\in(\mathcal{R}-(\mathcal{D}-\{T\}))}R)\ltimes S\ldots\overline{\ltimes}T\ldots$$

`Map` must also ignore `exists` and `not exists` conditions involving tables in $\mathcal{D}-\{T\}$. The above query still works but may deem more tuples as needed since `Closure` only takes into account the selection and join conditions but not the `exists` and `not exists` conditions.

Later in this section, we develop a new function $\texttt{Closure}_{\mathcal{C}}$, which takes into account `exists` and atomic `not exists` conditions. We then define $\texttt{Map}_{\mathcal{C}}$ similar to `Map` but using $\texttt{Closure}_{\mathcal{C}}$, and $\texttt{Needed}_{\mathcal{C}}$ similar to `Needed` but using $\texttt{Map}_{\mathcal{C}}$. Before we derive $\texttt{Closure}_{\mathcal{C}}$, we illustrate why taking into account the `exists` and `not exists` conditions is important in computing the closure.

EXAMPLE 4.3 In this example, we compare the tuples returned by $\texttt{Map}(E_{\mathcal{C}},O)$ and $\texttt{Map}_{\mathcal{C}}(E_{\mathcal{C}},O)$, where $E_{\mathcal{C}}$ is obtained by applying a set of constraints to

$$E=\pi_{O.clerk,\triangle C.custId,L.qty,L.cost}\\ \sigma_{L.cost>99\ \wedge\ \triangle C.custId<500\ \wedge\ O.ordId>1000\ \wedge\ O.ordId=L.ordId\ \wedge\ O.custId=\triangle C.custId}\ (\triangle C\times O\times L).$$

Let us suppose that only the constraints expressing the following information are applied to E: (1) $custId$ is the key of C (Constraint (10)); and (2) a referential integrity holds from $O.custId$ to $C.custId$ (Constraint (11)). The modified maintenance expression $E_{\mathcal{C}}$ is as follows:

$$E_{\mathcal{C}}=\pi_{O.clerk,\triangle C.custId,L.qty,L.cost}\\ \sigma_{L.cost>99\ \wedge\ \triangle C.custId<500\ \wedge\ O.ordId>1000\ \wedge\ O.ordId=L.ordId\ \wedge\ O.custId=\triangle C.custId}\\ ((\triangle C\ \overline{\ltimes}_{\triangle C.custId=C.custId}C)\times(O\ \ltimes_{O.custId=C.custId}C)\times L).$$

Notice that $\texttt{Map}(E_{\mathcal{C}},O)$ returns

$$\pi_{\texttt{Attrs}(O)}\sigma_{L.cost>99\ \wedge\ O.custId<500\ \wedge\ O.ordId>1000\ \wedge\ O.ordId=L.ordId}\ ((O\ \ltimes_{O.custId=C.custId}C)\times L),$$

after computing the closure of the selection and join conditions, ignoring the conditions referring to $\triangle C$, and removing $\triangle C$ from the cross product.

On the other hand, let us suppose that $\texttt{Map}_{\mathcal{C}}$ uses the function $\texttt{Closure}_{\mathcal{C}}$ to "handle" `exists` and `not exists` conditions obtaining the following expression from $E_{\mathcal{C}}$.

$$\pi_{O.clerk,\triangle C.custId,L.qty,L.cost}\\ \sigma_{L.cost>99\ \wedge\ \triangle C.custId<500\ \wedge\ O.ordId>1000\ \wedge\ O.ordId=L.ordId\ \wedge\ O.custId=\triangle C.custId}\\ ((\triangle C\ \overline{\ltimes}_{\triangle C.custId=C.custId}C\ \ltimes_{\triangle C.custId=C.custId}C)\times\\ (O\ \ltimes_{O.custId=C.custId\wedge O.custId\neq C.custId}C\ \overline{\ltimes}_{O.custId=C.custId}C)\times L)$$

Given the above expression, $\texttt{Map}_{\mathcal{C}}$ returns the following query

$$\pi_{\texttt{Attrs}(O)}\ \sigma_{L.cost>99\ \wedge\ O.custId<500\ \wedge\ O.ordId>1000\ \wedge\ O.ordId=L.ordId}\\ ((O\ \ltimes_{O.custId=C.custId\wedge O.custId\neq C.custId}C\ \overline{\ltimes}_{O.custId=C.custId}C)\times L).$$

This query has an empty answer because the `exists` condition on O is contradictory! Hence, $\texttt{Map}_{\mathcal{C}}(E_{\mathcal{C}},O)$ correctly states that no O tuple is needed in answering E, which makes sense because the new customers do not have any orders yet according to the constraints. On the other hand, $\texttt{Map}(E_{\mathcal{C}},O)$ returns a possibly severe overestimate of the O tuples needed. □

Alternative representation of $\ltimes$'s and $\overline{\ltimes}$'s: For convenience, we develop $\texttt{Closure}_{\mathcal{C}}$ to work on maintenance expressions that represent `exists` and `not exists` conditions differently. Instead of representing them using the $\ltimes$ and $\overline{\ltimes}$ operators, we represent them as conditions that are combined with the selection and join conditions. For instance, the query $R \ltimes_{R.a=S.a} S$ is represented as $\sigma_{\exists S_i \in S(R.a=S_i.a)}(R)$, where S_i is a tuple variable ([Ull89]). The query $R \overline{\ltimes}_{R.a=S.a} S$ is represented as $\sigma_{\neg\exists S_i^{asj} \in S(R.a=S_i^{asj}.a)}(R)$, or alternatively $\sigma_{\forall S_i^{asj} \in S(R.a \neq S_i^{asj}.a)}(R)$. We call this new representation the *quantifier representation*, and the previous one, the *operator representation.*

In the quantifier representation, we make implicit tuple variables, like "R" in the `exists` condition $\exists S_i \in S(R.a = S_i.a)$, explicit. Hence, given the maintenance expression

$$\pi_{O.clerk,\triangle C.custId,L.qty,L.cost}$$
$$\sigma_{L.cost>99\ \wedge\ \triangle C.custId<500\ \wedge\ O.ordId>1000\ \wedge\ O.ordId=L.ordId\ \wedge\ O.custId=\triangle C.custId}$$
$$((\triangle C\ \overline{\ltimes}_{\triangle C.custId=C.custId} C) \times (O \ltimes_{O.custId=C.custId} C) \times L),$$

its quantifier representation is $\pi_{O_0.clerk,\triangle C_0.custId,L_0.qty,L_0.cost}\sigma_{\mathcal{P}'}(\triangle C \times O \times L)$, where $\mathcal{P}'$ is

$$L_0.cost > 99 \quad \wedge \quad \triangle C_0.custId < 500 \wedge O_0.ordId > 1000 \wedge O_0.ordId = L_0.ordId \wedge O_0.custId = \triangle C_0.custId$$
$$\wedge \quad \forall C_2^{asj}(\triangle C_0.custId \neq C_2^{asj}.custId) \wedge \exists C_1(O_0.custId = C_1.custId). \tag{15}$$

We assign the tuple variables mechanically as follows. For a table T appearing in the cross product (*e.g.*, $\triangle C$, O, L), we assign the tuple variable T_0 (*e.g.*, $\triangle C_0$, O_0, L_0). For a table T appearing in an `exists` condition $R \ltimes T$, we assign a unique tuple variable T_i (*e.g.*, C_1), where $i > 0$. For a table T appearing in a `not exists` condition $R\ \overline{\ltimes}\ T$, we assign a unique tuple variable T_j^{asj} (*e.g.*, C_2^{asj}), where $j > 0$. Henceforth, we use "T" to denote either a free variable T_0, or an existentially quantified variable T_i, or a universally quantified tuple variable T_j^{asj}.

Deriving $\texttt{Closure}_{\mathcal{C}}$, $\texttt{Map}_{\mathcal{C}}$, and $\texttt{Neeeded}_{\mathcal{C}}$: In general, given a maintenance expression $E = \pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}} R)$ in quantifier representation, we can always obtain the *prenex normal form* (PNF) of $\mathcal{P}$, where all the quantifiers precede a quantifier-free condition expression ([PMW90]). That is $\mathcal{P}$ in PNF is of the form shown below where $\mathcal{P}'$ is a quantifier-free condition.

$$\exists R_i..\exists S_j..\forall T_k^{asj}..\forall U_l^{asj}(\mathcal{P}')$$

Assuming $\mathcal{P}'$ is conjunctive for now, $\texttt{Closure}_{\mathcal{C}}$ simply derives new atomic conditions from atomic conditions that use universally quantified tuple variables (*e.g.*, T_i^{asj}), and then uses the old `Closure` function to obtain the closure. More specifically, `Closure` uses standard axioms, such as the transitivity axiom, to derive atomic conditions ([Ull89]). $\texttt{Closure}_{\mathcal{C}}$ adds the following two axioms to derive additional atomic conditions from ones that use universally quantified variables.

1. $S_i^{asj}.a\ \theta\ T.b \Rightarrow S.a\ \theta\ T.b$, where θ is either $=,\neq,\leq,<,\geq$, or $>$.
2. $S_i^{asj}.a = T_j.b \Rightarrow S_i^{asj}.a = S_k^{asj}.a$.

The first (additional) axiom states that if $S_i^{asj}.a\ \theta\ T.b$ holds, it means that the a attribute of all the S tuples are related to $T.b$ in the same way. Hence, an atomic condition $S.a\ \theta\ T.b$ holds regardless of whether S is existentially or universally quantified. The second axiom states that if $S_i^{asj}.a$ is equated to an attribute of an existentially quantified tuple variable, it must be the case that the a attributes of all the S tuples have the same value. Note that S_k^{asj} must be distinct from S_i^{asj}. If no such tuple variable exist, we introduce a new one for the purpose of applying the second axiom. We illustrate $\texttt{Closure}_{\mathcal{C}}$ in the next example.

EXAMPLE 4.4 Let us suppose we are given a maintenance expression $E = \pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}} R)$, where $\mathcal{P}$ is Expression (15). $\mathcal{P}$ in PNF is $\exists C_1 \forall C_2^{asj}\ (\mathcal{P}')$, where $\mathcal{P}'$ is

$$\begin{aligned} L_0.cost > 99 \quad \wedge \quad & \triangle C_0.custId < 500 \wedge O_0.ordId > 1000 \wedge O_0.ordId = L_0.ordId \wedge O_0.custId = \triangle C_0.custId \\ \wedge \quad & \triangle C_0.custId \neq C_2^{asj}.custId \wedge O_0.custId = C_1.custId. \end{aligned}$$

Since both C_1 and C_2^{asj} are tuple variables ranging over the domain of table C's tuples, and C_2^{asj} is a universally quantified tuple variable, any atomic condition that applies to C_2^{asj} must also apply to C_1 (i.e., the first axiom). That is, a condition that applies to all tuples must apply to a particular tuple. For instance, the atomic condition $\triangle C_0.custId \neq C_2^{asj}.custId$ implies the atomic condition $\triangle C_0.custId \neq C_1.custId$. Notice that when `Closure` is run on $(\mathcal{P}' \wedge (\triangle C_0.custId \neq C_1.custId))$, the contradictory atomic conditions $O_0.custId = C_1.custId$ and $O_0.custId \neq C_1.custId$ is derived from $\triangle C_0.custId \neq C_1.custId$, $O_0.custId = \triangle C_0.custId$ and $O_0.custId = C_1.custId$. Consequently, $\texttt{Map}(O, E)$ is guaranteed to return an empty answer which is consistent with Example 4.3. On the other hand, if `Closure` is run on $\mathcal{P}'$ alone, no contradictory atomic conditions are derived. □

Algorithm 4.3 $\texttt{Closure}_{\mathcal{C}}$
Input: conjunctive condition $\mathcal{P}$ possibly with `exists` and
(atomic) `not exists` conditions in quantifier representation
Output: closure of $\mathcal{P}$

1. Derive PNF of $\mathcal{P}$ of the form $\exists..\exists..\forall..\forall..(\mathcal{P}')$, where $\mathcal{P}'$ is quantifier-free
2. Derive $\mathcal{P}'$ from $\mathcal{P}$ by applying the two axioms
   concerning universally quantified tuple variables.
3. Return $\exists..\exists..\forall..\forall..(\texttt{Closure}(\mathcal{P}'))$ ◇

Figure 6: $\texttt{Closure}_{\mathcal{C}}$

The example illustrated $\texttt{Closure}_{\mathcal{C}}$ (Algorithm 4.3, Figure 6) which computes the closure of a conjunctive condition $\mathcal{P}$, possibly with `exists` and `not exists` conditions. $\texttt{Closure}_{\mathcal{C}}$ first converts $\mathcal{P}$ to its PNF, obtaining a quantifier-free condition $\mathcal{P}'$ (Line 1). To ensure that $\mathcal{P}'$ is still conjunctive, we assume that `not exists` conditions only have a single atomic condition. That is, they are of the form $\neg\exists T_i^{asj} p$ (or $\forall T_i^{asj} \neg p$), where p is a single atomic condition.[5] Any `not exists` conditions that do not conform to the previous restriction are ignored (replaced with `true`) when computing the closure. (The `not exists` condition added by Constraint (9) is an example of an ignored `not exists` condition.) $\texttt{Closure}_{\mathcal{C}}$ then derives new atomic conditions (Line 2) based on the two additional axioms introduced previously. Finally, the old `Closure` function is used to compute the closure of the quantifier-free conjunctive condition $\mathcal{P}'$ as if it was a conjunction of selection and join conditions.

$\texttt{Closure}_{\mathcal{C}}$ is reasonably efficient and can be done in $O(n^3 + m^2 \cdot a)$, where n is the number of distinct attributes, m is the number of distinct tuple variables, and a is the number of atomic conditions in $\mathcal{P}$. (Line 2 is done in $O(m^2 \cdot a)$ time and Line 5 is done in $O(n^3)$ time.)

[5] A `not exists` condition composed of a disjunction of atomic conditions is allowed but this can be expressed as separate `not exists` conditions with a single atomic condition.

Using $\texttt{Closure}_{\mathcal{C}}$, we define $\texttt{Map}_{\mathcal{C}}$ to be the same as $\texttt{Map}$ except that it uses $\texttt{Closure}_{\mathcal{C}}$, and $\texttt{Needed}_{\mathcal{C}}$ to be the same as $\texttt{Needed}$ except that it uses $\texttt{Map}_{\mathcal{C}}$.

Lemma 4.1 *Given a table T and a set of maintenance expression $\mathcal{E}_{\mathcal{C}}$ obtained by applying the constraints $\mathcal{C}$ on a set of maintenance expression $\mathcal{E}$, the query*

$$\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}}) = \bigcup_{E_{\mathcal{C}} \in \mathcal{E}_{\mathcal{C}}} \texttt{Map}_{\mathcal{C}}(E_{\mathcal{C}}, T),$$

returns all the tuples in T that are needed by the maintenance expressions in $\mathcal{E}_{\mathcal{C}}$. If all constraints in $\mathcal{C}$ using `not exists` *conditions are of the form $\sigma_{\mathcal{P}_{LHS}}(\times_{R \in \mathcal{R}} R) \equiv \sigma_{\mathcal{P}_{RHS}}(\times_{R \in \mathcal{R}} R) \overline{\rhd\!\!\!<}_p T$ where p is a disjunction of atomic predicates, the query $T \rhd\!\!\!<_{\texttt{Attrs}(T)} \texttt{Needed}(T, \mathcal{E})$ returns only the tuples in T that are needed by the maintenance expressions in $\mathcal{E}_{\mathcal{C}}$. Furthermore, for any set of constraints $\mathcal{C}$, it is guaranteed that $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T, \mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T, \mathcal{E})$.* □

The proof for Lemma 4.1, together with all the details of on the completeness of $\texttt{Closure}_{\mathcal{C}}$ and its impact on $\texttt{Needed}_{\mathcal{C}}$, can be found in Appendix A.

5 Discussion

Although Lemma 4.1 itself does not guarantee that $\texttt{Needed}_{\mathcal{C}}$ always returns strictly fewer tuples than `Needed`, we now illustrate that in practice, $\texttt{Needed}_{\mathcal{C}}$ often returns much fewer tuples.

ClerkCust View: The *ClerkCust* view has 27 maintenance expressions, which we assume to comprise $\mathcal{E}$. (The maintenance expressions are listed in report [GMLY98].) $\mathcal{C}$ are the append-only, key, referential integrity, weak minimality and ad hoc constraints in Example 4.1. Table 2 gives the queries returned by $\texttt{Needed}(T, \mathcal{E})$ and $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$ for tables L, O and C.

The second row of Table 2 shows that $\texttt{Needed}_{\mathcal{C}}(L, \mathcal{E}_{\mathcal{C}})$ identifies accurately that none of the L tuples are needed by $\mathcal{E}$, while $\texttt{Needed}(L, \mathcal{E})$ deems a large number of L tuples as needed. $\texttt{Needed}_{\mathcal{C}}$ is much better because it eliminates any maintenance expression E where $\texttt{Map}(L, E)$ is guaranteed to return an empty answer given the constraints. Of the 27 expressions in $\mathcal{E}$, 20 are eliminated. Of the 7 remaining expressions, none refer to L. ($\triangle L$ and $\bigtriangledown L$ are used but not L.)

The third row of Table 2 shows that $\texttt{Needed}_{\mathcal{C}}(O, \mathcal{E}_{\mathcal{C}})$ identifies accurately (using a `not exists` condition) that only the *one* O tuple with the maximum *ordId* value is needed. On the other hand, $\texttt{Needed}(O, \mathcal{E})$ deems a large number of O tuples as needed.

The fourth row of Table 2 shows that $\texttt{Needed}_{\mathcal{C}}(C, \mathcal{E}_{\mathcal{C}})$ and $\texttt{Needed}(C, \mathcal{E})$ identify the same bag of needed tuples. This illustrates that using $\texttt{Needed}_{\mathcal{C}}$ does not always help in reducing the number of tuples that are deemed needed.

TPC-D Benchmark: We now investigate what TPC-D ([Com]) base relation tuples are needed assuming certain TPC-D queries are used as views. In particular, we focus on 4 out of the 9 TPC-D base relations: *LINEITEM* (L), *ORDER* (O), *CUSTOMER* (C) and *PART* (P). Fact tables L and O contain 86% of the tuples in the benchmark. Hence, expiration requests will likely be issued on these two tables. We consider two views, V_3 and V_5, whose definition queries are the TPC-D queries $Q3$ ("Shipping Priority Query") and $Q5$ ("Local Supplier Volume Query"), respectively. We assume that either the maintenance expressions of V_3 or V_5 comprise $\mathcal{E}$. (Other queries that refer to the four tables give similar results.) Finally, the set of constraints $\mathcal{C}$ we consider is based on the TPC-D "update model" specification (see [Com]).

Table T	$\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$	$\texttt{Needed}(T, \mathcal{E})$
L	$\{\ \}$	$\pi_{\texttt{Attrs}(L)}\sigma_{L.cost>99 \wedge L.ordId>1000}(L)$
O	$\pi_{\texttt{Attrs}(O)}\sigma_{O.custId<500 \wedge O.ordId>1000}$ $(O \overline{\ltimes}_{O.ordId<O'.ordId} \rho_{O'} O)$	$\pi_{\texttt{Attrs}(O)}\sigma_{O.custId<500 \wedge O.ordId>1000}(O)$
C	$\pi_{\texttt{Attrs}(C)}\sigma_{C.custId<500}(C)$	$\pi_{\texttt{Attrs}(C)}\sigma_{C.custId<500}(C)$

Table 2: Comparison of $\texttt{Needed}_{\mathcal{C}}$ and $\texttt{Needed}$ Using $ClerkCust$

Table T	$\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$	$\texttt{Needed}(T, \mathcal{E})$
L	0%	100%
O	0%	100%
C	20%	20%
P	0%	0%

Table 3: Comparison of $\texttt{Needed}_{\mathcal{C}}$ and $\texttt{Needed}$ Using TPC-D Query $Q3$

Table T	$\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$	$\texttt{Needed}(T, \mathcal{E})$
L	0%	100%
O	0%	100%
C	100%	100%
P	100%	100%

Table 4: Comparison of $\texttt{Needed}_{\mathcal{C}}$ and $\texttt{Needed}$ Using TPC-D Query $Q5$

To simplify the presentation, we do not give the queries returned by the functions but instead give the percentage of the base relation tuples that are needed. We obtained this percentage for each table T (*i.e.*, L, O, C, and P) by running the queries returned by $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$ and $\texttt{Needed}(T, \mathcal{E})$. We then counted the number of tuples in the result and divided it by the number of T tuples.

Table 3 gives the tuples that are needed by the maintenance expressions of V_3 assuming the constraints in $\mathcal{C}$. $\texttt{Needed}_{\mathcal{C}}$ identifies that none of the L and O tuples are needed, and 20% of the C tuples are needed. Since P is not referred to in V_3's definition query, none of its tuples are needed to maintain V_3. None of the L and O tuples are needed because of the append-only behavior of L and O specified in the benchmark, *i.e.*, $\triangle L$ tuples only join with $\triangle O$ tuples and vice versa. Only 20% of the C tuples are needed because $\texttt{Needed}_{\mathcal{C}}$ applies a selection condition on C with 20% selectivity. On the other hand, $\texttt{Needed}$ deems all of the L and O tuples as needed.

Table 4 shows similar results assuming the maintenance expressions of view V_5 comprise $\mathcal{E}$. The only difference is that both $\texttt{Needed}_{\mathcal{C}}$ and $\texttt{Needed}$ identify that all the tuples of C and P are needed. This is because V_5's definition query does not apply any selection conditions on C nor P. Had there been constraints that state that "some of the customers no longer place orders", or "some parts can no longer be ordered", then $\texttt{Needed}_{\mathcal{C}}$ would mark some C and P tuples as unneeded.

The previous study shows that using constraints allows greater flexibility for expiration and can significantly decrease storage requirements when data is no longer needed. Furthermore, it is likely that the efficiency of view maintenance is improved because the expired data is no longer processed by the maintenance expressions. Also, we illustrated that constraints can be used to eliminate some of the maintenance expressions altogether which definitely improves view maintenance.

6 Dynamic Setting

In the previous two sections, we focused on an initial static setting wherein we are given a set of tables $\mathcal{T}$, a set of maintenance expressions $\mathcal{E}$, and a set of constraints $\mathcal{C}$. In this section, we explore how to cope with a dynamic setting wherein some of these parameters can be changed. Furthermore, we also drop the assumption that none of the tuples have been expired.

Before discussing the algorithms, it is important to note that even when parameters change, expiration requests are satisfied the same way. That is, given an expiration request $\sigma_{\mathcal{P}}(T)$ on T, it is satisfied by removing the tuples in $\sigma_{\mathcal{P} \wedge needed=\texttt{false}}(T)$.

Also, note that the queries returned by $\texttt{Needed}_{\mathcal{C}}$ (and $\texttt{Needed}$) still have complete answers even after some tuples have been expired. This is because any query returned by $\texttt{Needed}_{\mathcal{C}}$ takes the union of expressions derived from maintenance expressions using $\texttt{Map}_{\mathcal{C}}$. Since we guaranteed that all the tuples that are needed by maintenance expressions are not expired, the completeness of the queries returned by $\texttt{Needed}_{\mathcal{C}}$ follows. We now outline the algorithms for coping with various changes.

Changes to $\mathcal{T}$: Suppose *Def*(V) has a complete answer and V is added to $\mathcal{T}$. We must identify for each table T that V is defined on, which of the T tuples previously deemed as unneeded is now needed to maintain V. A reasonably efficient solution to the problem is to use the query $\sigma_{needed=\texttt{false}}(T) \ltimes_{\texttt{Attrs}(T)} \texttt{Needed}(T, \mathcal{E}_V)$, where $\mathcal{E}_V$ are the maintenance expressions of V. This query identifies the unneeded T tuples that now need to be marked as needed.

Changes To $\mathcal{C}$: We only allow changes to $\mathcal{C}$ that expire more tuples. There are two types of changes that satisfy this condition. First, a constraint may have been added to $\mathcal{C}$. Second, a constraint c previously in $\mathcal{C}$ may have been changed so that conditions are removed from $LHS(c)$ (*i.e.*, more opportunities for applying c) or added to $RHS(c)$ (*i.e.*, more conditions added whenever c is applied). To update the extension markings, for each table T, we use the query

$$\sigma_{needed=\texttt{true}}(T) \overline{\ltimes}_{\texttt{Attrs}(T)} \texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}),$$

to identify the T tuples that were previously deemed needed (*i.e.*, *needed* = $\texttt{true}$), but must now be marked as unneeded since they are not in $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E})$. Further, assuming the change to $\mathcal{C}$ is due to a change in *Constraint*(S), for some table S, we only need to modify the extension marking of a table T defined on S. This is valid under our assumption that the constraint of a view is not computed from the constraints of the underlying tables (*i.e.*, the administrator inputs all constraints). Even without this assumption, we can still identify the tables whose extension marking may be modified by defining a *table graph*. The nodes in a table graph represent base relations or views. There is an edge $U \rightarrow V$ if V is defined on U. In general then, we only need to modify the extension marking of a table T if T is a node in the sub-graph "rooted" at S.

Insertions: Periodically, insertions $\triangle T$ and deletions $\triangledown T$ are computed for each table T. While deleting the $\triangledown T$ tuples from T does not pose any problem, inserting the $\triangle T$ tuples into T may. First, the inserted tuples need to be marked as needed or unneeded. Second, some of the unneeded tuples may need to be expired. The two problems are solved by performing the following procedure.

1. Insert $\triangle T$ and set *needed* attribute to $\texttt{false}$ for all inserted tuples.

2. For the T tuples in $\sigma_{needed=\texttt{false}}(T) \ltimes_{\texttt{Attrs}(T)} \texttt{Needed}(T, \mathcal{E})$, set *needed* attribute to `true`.
3. Expire T tuples in $\sigma_{\mathcal{P} \wedge needed=\texttt{false}}(T)$, where $LastReq(T) = \sigma_{\mathcal{P}}(T)$.

The first step assumes all $\triangle T$ tuples are unneeded and do not need to be expired. The second step marks the $\triangle T$ tuples that are needed. The last step expires unneeded $\triangle T$ according to $LastReq(T)$. The most expensive step is clearly the second one. However, only the maintenance expressions of views $\mathcal{V}$ that are defined on T need to be considered. Hence, the step is reasonably efficient since it is (only) as expensive as computing the insertions to the views in $\mathcal{V}$ based on $\triangle T$.

7 Related Work

One of the problems that our framework tackles is how to maintain a view when only parts of the underlying tables are accessible. Most work on view maintenance assumes that the complete underlying tables are accessible, for example, [BLT86, CW91, GL95, GMS93, Han87, QW91]. However, there has also been work on view maintenance that assumes otherwise. [BT88] and [GJM96] identified *self-maintainable* views that can be maintained without accessing underlying tables. [QGMW96], [HZ96] and [Qua97] tried to make a view self-maintainable by defining auxiliary views such that the view and the auxiliary views together are self-maintainable. The function $\texttt{Needed}(T, \mathcal{E})$ we introduce serves essentially the same purpose as an auxiliary view, although it does not have to be maintained as such. [HZ96] developed a framework wherein the attributes of the underlying tables may be inaccessible. In our framework, the tuples of a table can be made inaccessible. It will be important in future work to combine both approaches.

Our framework also takes advantage of the available constraints in order to reduce the size of $\texttt{Needed}(T, \mathcal{E})$ and increase the effectiveness of expiration. This is different from, but related to, the use of constraints in the area of *semantic query optimization* [CGM88, Kin81, SO89]. It is important to point out their connection since semantic query optimization has largely been ignored in view maintenance literature. Indeed, there has been some prior work in improving view maintenance using constraints; however, they all use special-case algorithms to take advantage of specific constraints. For instance, [QGMW96] used a specialized algorithm that exploits key and referential integrity constraints to eliminate maintenance expressions. [GJM96] used key constraints to rewrite maintenance expressions for a view to use itself. [JMS95] introduced *chronicles* that are updated in a special manner, and showed that views defined on chronicles can be maintained efficiently. In our approach, we can describe chronicles using constraints and automatically infer that the entire chronicles can be safely expired. In summary, the techniques we introduce generalize many special-case algorithms developed in the previous work. Furthermore, since we exploit a broader class of constraints, we improve on many of the algorithms.

Our framework also introduces tables whose real extensions are not complete when compared to their full extensions. There has been numerous work on incomplete databases. See [AHV95] for an overview. We are now investigating how previous work in the area can be used to solve some of the problems borne out of the framework. For instance, [Lev96]'s work on obtaining complete answers from an incomplete database is helpful in solving the fourth problem stated in Section 2.

Finally, the algorithms in [BCL89] for detecting irrelevant updates can be modified to detect unneeded tuples. This can be done by treating the maintenance expressions as views and treating a tuple $t \in T$ as if it were an insertion. However, the algorithms in [BCL89] do not work with constraints. Also, they require a satisfiability test for each tuple t. Our method is more "set-oriented" since it uses queries.

8 Conclusion

We have presented a framework for system-managed removal of warehouse data that avoids affecting the user-defined materialized views. Within it, the user or administrator can declaratively specify what he wants to expire and the system removes as much data as possible. The administrator can also input constraints (implied by the application) which the system uses to expire more data, as we illustrated using the TPC-D benchmark. We identified problems borne out of the framework and we solved the central problems by developing efficient algorithms. These problems included ones of a dynamic nature where the parameters of the framework may change.

References

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.

[BCL89] J. Blakely, N. Coburn, and P. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *TODS*, 14(3):369–400, September 1989.

[BLT86] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 61–71, May 1986.

[BT88] J. A. Blakeley and F. W. Tompa. Maintaining materialized views without accessing base data. *Information Systems*, 13(4):393–406, 1988.

[CGM88] U. Chakravarthy, J. Grant, and J. Minker. Foundations of semantic query optimization for deductive databases. In *Foundations of Deductive Databases and Logic Programming*, pages 243–273. Morgan Kaufman, 1988.

[Com] TPC Committee. Transaction Processing Council. Available on web at: `http://www.tpc.org/`.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 1991 International Conference on Very Large Data Bases*, September 1991.

[GJM96] A. Gupta, H. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Proceedings of the 1996 International Conference on Extending Database Technology*, March 1996.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 328–339, May 1995.

[GML97] H. Garcia-Molina and W. Labio. Expiration and partially materialized views. Technical report, Stanford University, 1997. available at http://www-db.stanford.edu/pub/papers/expire.ps.

[GMLY98] H. Garcia-Molina, W. Labio, and J. Yang. Expiring data from a warehouse. Technical report, Stanford University, 1998. available at http://www-db.stanford.edu/pub/papers/newexpire.ps.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 157–166, May 1993.

[Han87] E. Hanson. A performance analysis of view materialization strategies. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 440–453, May 1987.

[HZ96] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 481–492, June 1996.

[JMS95] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *Proceedings of the Fourteenth ACM Symposium on Principles of Database Systems*, pages 113–124, May 1995.

[Kin81] J. J. King. QUIST : A system for semantic query optimization in relational data bases. In *Proceedings of the 1981 International Conference On Very Large Data Bases*, pages 510–517, September 1981.

[Lev96] A. Y. Levy. Obtaining complete answers from incomplete databases. In *Proceedings of the 1996 International Conference on Very Large Data Bases*, pages 402–412, September 1996.

[NY82] M. Nicholas and K. Yazdanian. Integrity checking in deductive databases. In H. Galliere and J. Minker, editors, *Logic and Databases*, pages 325–346. Plenum Press, 1982.

[PMW90] B. Partee, A. Meulen, and R. Wall. *Mathematical Methods in Linguistics.* Kluwer Academic Publishers, 1990.

[QGMW96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proceedings of the 1996 International Conference on Parallel and Distributed Information Systems*, pages 158–169, December 1996.

[Qua96] D. Quass. Maintenance expressions for views with aggregation. In *In Workshop on Materialized Views: Techniques and Applications, in cooperation with ACM SIGMOD*, June 1996.

[Qua97] Dallan Quass. *Materialized Views in Data Warehouses.* PhD thesis, Stanford University, Stanford, CA 94305, 1997.

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

[SO89] S. Shenoy and Z. Ozsoyoglu. Design and implementation of a semantic query optimizer. *IEEE Transactions on Knowledge and Data Engineering*, (3):344–361, 1989.

[Ull89] J. D. Ullman. *Database and Knowledge-Base Systems, Vol.2.* Computer Science Press, 1989.

[YL87] H. Yang and P.-A. Larson. Query transformation for PSJ-queries. In *Proceedings of the 1987 International Conference On Very Large Data Bases*, pages 245–254, October 1987.

A Proof of Lemmas

Before we prove the lemmas, we define some notation. Given a maintenance expression $E = \pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R)$, we use $\texttt{Res}(E)$ to denote $\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R)$. That is, $\texttt{Res}(E)$ projects all the attributes of all the tables involved in the cross product. We use t_{res} to denote a tuple in $\texttt{Res}(E)$. Note that every t_{res} "manifests" itself in the result of E because we use bag semantics. Furthermore, since we require aggregate views to have the `COUNT` aggregate function, this observation holds for aggregate views. Similarly, we use t_{mapres} to denote a tuple in $\texttt{Res}(\texttt{Map}(E,T))$. Assuming $\mathcal{R}' \subseteq \mathcal{R}$, $t_{res}[\mathcal{R}']$ denotes the tuple resulting from t_{res} that includes only the attributes of the tables in $\mathcal{R}'$. We also use $t_{res}[\mathcal{A}]$ to denote the tuple resulting from t_{res} that includes only the attributes in $\mathcal{A}$. Given a maintenance expression $E = \pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R)$, we use $\mathcal{P}[t_{mapres}]$ to be the condition resulting from replacing each attribute with its value in t_{mapres}.

We now formalize the definition of when a tuple t is "needed" by E.

Definition A.1 **(needed)** Let $E = \pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R)$, and $T \in \mathcal{R}$. Let $\mathcal{D}$ be the delta relations in $\mathcal{R}$. Tuple $t \in T$ is *needed* by E if and only if for some extension of the delta relations, $\exists t_{res} \in \texttt{Res}(E)$ such that $t_{res}[\{T\}] = t$ and for all $R \in \mathcal{R}$ that is not a delta relation, $t_{res}[\{R\}] \in R$. □

Intuitively, the definition states that $t \in T$ is needed by $E = \pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R)$ if there is a tuple t_{res} in $\texttt{Res}(E)$ that t can "contribute" to. If t is removed from T, then $t_{res}[\mathcal{A}]$ is also removed from the result of E. (Note that this also holds even when t has duplicates because removing t would decrease the number of duplicates of $t_{res}[\mathcal{A}]$.)

Proof of Lemma 3.1 For the proof, we denote $\texttt{Map}(E,T)$ as $\pi_{\texttt{Attrs}(T)}\sigma_{\mathcal{P}'}(\times_{R\in(\mathcal{R}-\mathcal{D})}R)$, where $\mathcal{P}'$ is obtained from $\mathcal{P}$ using `Closure` and `Ignore`. $\mathcal{D}$ are the delta relations in $\mathcal{R}$.

Proof (Lemma 3.1):
(Necessity) Assume $t \in T$ is needed. By Definition A.1, there exist a tuple $t_{res} \in \texttt{Res}(E)$ so that $t_{res}[\{T\}] = t$. We can obtain $t_{mapres} \in \texttt{Res}(\texttt{Map}(E,T))$ as $t_{res}[\mathcal{R} - \mathcal{D}]$. This follows from the soundness of the `Closure` procedure ([Ull89]) and the definition of the `Ignore` procedure which guarantee that $\mathcal{P} \Rightarrow \mathcal{P}'$. It follows that $t = t_{mapres}[\{T\}](= t_{res}[\{T\}])$ since the attribute values were not changed in obtaining t_{mapres} from t_{res}. Hence $t \in \texttt{Map}(E,T)$ and $t \in \texttt{Needed}(T,\mathcal{E})$ (since $E \in \mathcal{E}$).
(Sufficiency) Assume $t \in \texttt{Needed}(T,\mathcal{E})$. Hence, for some $E \in \mathcal{E}$, $t \in \texttt{Map}(E,T)$. This implies that $t_{mapres} \in \texttt{Res}(\texttt{Map}(E,T))$ with $t_{mapres}[\{T\}] = t$. Since $\texttt{Map}(E,T)$ is not empty (because of the presence of t), $\mathcal{P}'$ does not have the atomic condition `false` ,*i.e.*, $\mathcal{P}'$ is satisfiable. Since `Ignore` does not remove `false`, it must be that $\mathcal{P}$ is also satisfiable. Furthermore, since the `Closure` procedure is complete ([Ull89]), we are guaranteed that $\mathcal{P}[t_{mapres}]$ is satisfiable.

To see this, there are five types of atomic conditions in $\mathcal{P}$. (R_i and R_j denote non-delta relations. D_k and D_l denote delta relations. K denotes a constant. θ is $=$, $>$ $\geq$, $\leq$, $<$, or $\neq$) They are: (1) $R_i.a\ \theta\ R_j.b$; (2) $R_i.a\ \theta\ K$; (3) $R_i.a\ \theta\ D_k.b$; (4) $D_k.a\ \theta\ D_l.b$; (5) $D_k.a\ \theta\ K$. Recall that $\mathcal{P}$ is satisfiable. Since $t_{mapres} \in \texttt{Res}(\texttt{Map}(E,T))$ it follows that $\mathcal{P}'[t_{mapres}] = \texttt{true}$. Since $\mathcal{P}'$ is obtained using `Closure` on $\mathcal{P}$ (then `Ignore`), $\mathcal{P}' \Rightarrow \texttt{Ignore}(\mathcal{P},\mathcal{D})$ and $\texttt{Ignore}(\mathcal{P},\mathcal{D})[t_{mapres}] = \texttt{true}$. Hence, $\mathcal{P}[t_{mapres}]$ is a conjunction of `true` and atomic conditions of type (3), (4) and (5). Type (3) atomic conditions become type (5) since the attribute references are replaced by the values in t_{mapres}. Since `Closure` is complete, any type (5) condition that implies a type (2) condition through type (3) and type (4) conditions were inferred. Since t_{mapres} satisfies these inferred type (2) conditions, it must be that the conjunction of type (4) and type (5) conditions in $\mathcal{P}[t_{mapres}]$ is satisfiable.

Since $\mathcal{P}[t_{mapres}]$ is satisfiable, we can construct t_{res} as follows. For all attributes in $\mathcal{R} - \mathcal{D}$, copy the values from t_{mapres}. For attributes in $\mathcal{D}$, assign values so that $\mathcal{P}[t_{res}]$ is `true`. The existence of these values is guaranteed by the fact that $\mathcal{P}[t_{mapres}]$ is satisfiable. Since $\mathcal{P}[t_{res}]$ is `true`, $t_{res} \in \texttt{Res}(E)$. By Definition A.1, t is needed. ■

Proof of Lemma 4.1 Since we now prove Lemma 4.1 which deals with constraints, we assume a maintenance expression $E = \pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R)$ in quantifier representation. We also denote $\texttt{Map}_{\mathcal{C}}(E,T)$ as $\pi_{\texttt{Attrs}(T)}\sigma_{\mathcal{P}'}(\times_{R\in(\mathcal{R}-\mathcal{D})}R)$, where $\mathcal{P}'$ is obtained from $\mathcal{P}$ using $\texttt{Closure}_{\mathcal{C}}$ and `Ignore`. Before we prove Lemma 4.1, recall that the lemma makes three statements: (1) All the needed T tuples are in $\texttt{Needed}_{\mathcal{C}}(T,\mathcal{E}_{\mathcal{C}})$. (2) $\texttt{Needed}_{\mathcal{C}}(T,\mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T,\mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T,\mathcal{E})$. (3) Under certain restrictions on the `not exists` constraints in $\mathcal{C}$, only the needed T tuples are in $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$. We prove these statements in turn.

Proof (Statement (1), Lemma 4.1):
Assume $t \in T$ is needed. By Definition A.1, there exist a tuple $t_{res} \in \texttt{Res}(E)$ so that $t_{res}[\{T\}] = t$. We can obtain $t_{mapres} \in \texttt{Res}(\texttt{Map}_{\mathcal{C}}(E,T))$ as $t_{res}[\mathcal{R} - \mathcal{D}]$. This follows from the soundness of the $\texttt{Closure}_{\mathcal{C}}$ procedure and the definition of the `Ignore` procedure which guarantee that $\mathcal{P} \Rightarrow \mathcal{P}'$. ($\texttt{Closure}_{\mathcal{C}}$ is sound in that it only derives conditions that are implied by $\mathcal{P}$.) It follows that $t = t_{mapres}[\{T\}](= t_{res}[\{T\}])$ since the attribute values were not changed in obtaining t_{mapres} from t_{res}. Hence $t \in \texttt{Map}_{\mathcal{C}}(E,T)$ and $t \in \texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$ assuming $E \in \mathcal{E}_{\mathcal{C}}$. ∎

Proof (Statement (2), Lemma 4.1):
For each $E \in \mathcal{E}$, $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$ uses $\texttt{Map}(E_{\mathcal{C}}, T)$; $\texttt{Needed}(T, \mathcal{E}_{\mathcal{C}})$ uses $\texttt{Map}(E_{\mathcal{C}}, T)$; and $\texttt{Needed}(T, \mathcal{E})$ uses $\texttt{Map}(E,T)$. Let $E = \pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}} R)$. Let $E_{\mathcal{C}} = \pi_{\mathcal{A}}\sigma_{\mathcal{P}_q}(\times_{R\in\mathcal{R}} R)$. Since $\mathcal{P}_q \Rightarrow \mathcal{P}$ due to additional `exists` and `not exists` conditions, $\texttt{Map}(E_{\mathcal{C}}, T) \subseteq \texttt{Map}(E,T)$. It follows that $\texttt{Needed}(T, \mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T, \mathcal{E})$.

Let $\texttt{Map}(E_{\mathcal{C}}, T) = \pi_{\texttt{Attrs}(T)}\sigma_{\mathcal{P}'_q}(\times_{R\in\mathcal{R}} R)$, and $\texttt{Map}_{\mathcal{C}}(E_{\mathcal{C}}, T) = \pi_{\texttt{Attrs}(T)}\sigma_{\mathcal{P}''_q}(\times_{R\in\mathcal{R}} R)$. More conditions (implied by `exists` and `not exists` conditions) are added in $\mathcal{P}''_q$, and they are not in $\mathcal{P}'_q$. Therefore, $\mathcal{P}''_q \Rightarrow \mathcal{P}'_q$. It follows that $\texttt{Map}_{\mathcal{C}}(E_{\mathcal{C}}, T) \subseteq \texttt{Map}(E_{\mathcal{C}}, T)$. Hence, $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T, \mathcal{E}_{\mathcal{C}})$. ∎

Proof (Statement (3), Lemma 4.1):
Assume $t \in \texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$. For some $E \in \mathcal{E}_C$, it must be that $t \in \texttt{Map}_{\mathcal{C}}(E,T)$. This implies that $t_{mapres} \in \texttt{Res}(\texttt{Map}_{\mathcal{C}}(E,T))$ with $t_{mapres}[\{T\}] = t$. Since $\texttt{Map}_{\mathcal{C}}(E,T)$ is not empty, the selection condition expression of $\texttt{Map}_{\mathcal{C}}(E,T)$ denoted $\mathcal{P}'$ does not have the atomic condition `false` ,*i.e.*, $\mathcal{P}'$ is satisfiable. Since `Ignore` does not remove `false`, it must be that the selection condition expression of E, denoted $\mathcal{P}$, is also satisfiable. Furthermore, assuming $\texttt{Closure}_{\mathcal{C}}$ is complete, we are guaranteed that $\mathcal{P}[t_{mapres}]$ is satisfiable.

Since $\mathcal{P}[t_{mapres}]$ is satisfiable, we can construct t_{res} as follows. For all attributes in $\mathcal{R} - \mathcal{D}$, copy the values from t_{mapres}. For attributes in $\mathcal{D}$, assign values so that $\mathcal{P}[t_{res}]$ is `true`. The existence of these values is guaranteed by the fact that $\mathcal{P}[t_{mapres}]$ is satisfiable. Since $\mathcal{P}[t_{res}]$ is `true`, $t_{res} \in \texttt{Res}(E)$. By Definition A.1, t is needed. ∎

The proof of Statement (3) of Lemma 4.1 relies on having a complete $\texttt{Closure}_{\mathcal{C}}$ algorithm. Before we present the algorithm and prove its completeness, we introduce some notation. As before, we use S_i to denote an existentially quantified tuple variable over S, and S_j^{asj} to denote a universally quantified tuple variable over S. We use "S" and "S'" to denote either S_i or S_j^{asj}. We use X to denote a reference to some attribute $T.a$. Finally, we use θ to denote either $<$, $\leq$, $\neq$, $=$, $\geq$ or $>$. Since an atomic condition $T.b > S.a$ ($T.b \geq S.a$) can always be expressed as $S.a < T.b$ ($S.a \leq T.b$, respectively), we focus on the first four comparison operators.

We now present in detail the $\texttt{Closure}_{\mathcal{C}}$ algorithm and prove that it is complete. The algorithm uses the following axioms to obtain all atomic conditions implied by $\mathcal{P}$.

The following 8 axioms are for inferring equalities from a conjunction of atomic conditions.

E1: $S.a = S.a$
E2: $S.a = T.b \Rightarrow T.b = S.a$
E3: $S.a = T.b \wedge T.b = U.c \Rightarrow S.a = U.c$
E4: $S.a = T.b \Rightarrow S.a \leq T.b$
E5: $S.a \leq T.b \wedge T.b \leq U.c \Rightarrow S.a \leq U.c$
E6: $S.a \leq T.b \wedge T.b \leq S.a \Rightarrow S.a = T.b$

E7*: $S_i^{asj}.a\ \theta\ T.b \Rightarrow S.a\ \theta\ T.b$
E8*: $S_i^{asj}.a = T_j.b \Rightarrow S_i^{asj}.a = S_k^{asj}.a$

Axioms E1–E6 are fairly standard and are clearly sound. Axioms E7 and E8 are the two additional axioms introduced in Section 4 and are also sound.

The following 9 axioms derive inequalities. The first 8 axioms are called Armstrong's axioms, and were proven in [Ull89] to be sound and complete when none of the tuple variables is universally quantified. Axiom I9 is identical to Axiom E7 and is also sound.

I1: $S.a \leq S.a$
I2: $S.a < T.b \;\Rightarrow\; S.a \leq T.b$
I3: $S.a < T.b \;\Rightarrow\; S.a \neq T.b$
I4: $S.a \leq T.b \wedge S.a \neq T.b \;\Rightarrow\; S.a < T.b$
I5: $S.a \neq T.b \;\Rightarrow\; T.b \neq S.a$
I6: $S.a < T.b \wedge T.b < U.c \;\Rightarrow\; S.a < U.c$
I7: $S.a \leq T.b \wedge T.b \leq U.c \;\Rightarrow\; S.a \leq U.c$
I8: $S.a \leq U.c \wedge U.c \leq T.b \wedge S.a \leq V.d \wedge V.d \leq T.b \wedge U.c \neq V.d \;\Rightarrow\; S.a \neq T.b$
I9*: $S_j^{asj}.a\ \theta\ X \;\Rightarrow\; S_i.a\ \theta\ X$.

We assume we have a procedure `CloseEqual` that fires Axioms E1–E8 and a procedure -`CloseInequal` that fires Axioms I1–I9 when given a conjunction of atomic conditions. `Closure`$_{\mathcal{C}}$ uses the two procedures in the following steps.

1. Use `CloseEqual` to obtain all equality atomic conditions. Using these equality atomic conditions, we place two attributes $S.a$ and $T.b$ in the same equivalence class C if $S.a = T.b$ results from `CloseEqual`.
2. For each equivalence class C, pick an attribute $S_i.a$ where S_i is an existentially quantified tuple variable. For each attribute $T_j.b$ (that is not $S_i.a$) in C, replace each atomic condition $T_j.b\ \theta\ X$ with $S_i.a\ \theta\ X$.
3. Use `CloseInequal` to obtain all the inequality atomic conditions.
4. Add additional atomic conditions by examining each equivalence class. For an equivalence class C, assume $S_i.a$ was the attribute picked in Step 2. For each attribute $T_j.b$ (that is not $S_i.a$) in C, introduce the atomic condition $T_j.b\ \theta\ X$ if $S_i.a\ \theta\ X$ is in the closure.

`Closure`$_{\mathcal{C}}$ is clearly sound since it does not derive any atomic condition that is not implied by $\mathcal{P}$.

Assuming both `CloseEqual` and `CloseInequal` are complete, it is not hard to show that `Closure`$_{\mathcal{C}}$ is complete. Suppose there is an atomic condition $T.b\ \theta\ X$ that is implied by the given conjunction of atomic conditions, but it is not derived by `Closure`$_{\mathcal{C}}$. It must be the case that θ is not = because otherwise `CloseEqual` would have produced it (in Step 1). It must also be the case that T is existentially quantified and $T.b$ is not an attribute that was picked in Step 2 of `Closure`$_{\mathcal{C}}$. Otherwise, `CloseInequal` would have produced $T.b\ \theta\ X$ in Step 3. However, it is guaranteed that Step 4 produces $T.b\ \theta\ X$. Otherwise, `CloseInequal` must have failed to produce $S_i.a\ \theta\ X$, where $S_i.a$ is the attribute that belongs to the same equivalence class as $T.b$ that was picked in Step 2. This is implies that `CloseInequal` is incomplete, contradicting our assumption. We now prove the completeness of `CloseEqual` and `CloseInequal`.

We prove the completeness of `CloseEqual` by proving the following lemma which states that given a conjunction of atomic conditions $\mathcal{P}$ input to `Closure`$_{\mathcal{C}}$, there is no $S.a = T.b$ that is implied by $\mathcal{P}$ but is not in $\mathcal{P}^+$ (the output of `CloseEqual`).

Lemma A.1 *Let $\mathcal{P}$ be a conjunction of atomic conditions input to* `Closure`$_{\mathcal{C}}$*, such that* `false` *is not implied by $\mathcal{P}$. Then every equality $S.a = T.b$ not in $\mathcal{P}^+$ has some assignment of a set of integers to each attribute used in $\mathcal{P}$ that makes all the atomic conditions in $\mathcal{P}^+$ true but $S.a\ \theta\ T.b$ false.* □

Proof (Lemma A.1 (Completeness of `CloseEqual`*)):*
After applying the set of axioms, we derive a set of equivalent classes where each equivalent class contains a set of attributes that are inferred to be equal (from Axioms E1–E3, E6–E8). We construct a graph as follows: Each node N in the graph corresponds to an equivalent class of attributes $N.attrs$. There is a directed edge from node N to node M iff $U.c < V.d$ or $U.c \leq V.d$ for some $U.c$ in $M.attrs$ and $V.d$ in $N.attrs$.

We now show that the graph is acyclic. Suppose that there is a directed cycle. This means that we have a chain of inequalities. Since we assume $\mathcal{P}$ is satisfiable, this chain must consist of only $\leq$. However, Axioms E5 and E6, we should have easily derived the fact that all attributes in these nodes are equal, i.e., all attributes belong to one equivalent class. This is a contradiction. Therefore, the graph is a DAG.

We assign values to attributes as follows: First find a topological sort of the graph such that M comes before N in the order if there is an edge from N to M. Assign a strictly increasing sequence of integers to nodes in the topological order. Attributes corresponding to the same node are assigned the same integer. (Note that the attribute of a universally quantified tuple will have only one integer value.)

We show that this assignment satisfies $\mathcal{P}$. If $U.c = V.d$ is in $\mathcal{P}^+$, then $U.c$ and $V.d$ belong to the same node in the graph, and hence are assigned to the same integer. If $U.c! = V.d$ is in $\mathcal{P}^+$, then $U.c$ and $V.d$ belong to different nodes (otherwise $\mathcal{P}$ is unsatisfiable), and hence are assigned to different integers. If $U.c \leq V.d$ is in $\mathcal{P}^+$ and $U.c$ and $V.d$ correspond to the same node, then $U.c \leq V.d$ is satisfied because they are assigned to the same integer. If $U.c \leq V.d$ is in $\mathcal{P}^+$ but $U.c$ and $V.d$ correspond to different nodes, there should be an edge from the node for $V.d$ to the node for $U.d$, so the assignment guarantees that $U.c < V.d$. If $U.c < V.d$ is in $\mathcal{P}^+$, then $U.c$ and $V.d$ must correspond to different nodes (otherwise $\mathcal{P}$ is unsatisfiable), and there should be an edge from the node for $V.d$ to the node for $U.c$, so the assignment guarantees that $U.c < V.d$. In conclusion, $\mathcal{P}$ is satisfied by this attribute assignment.

Now, suppose that $S.a = T.b$ cannot be derived from $\mathcal{P}$ using the set of axioms. $S.a$ and $T.b$ must correspond to different nodes in the graph, or else $S.a = T.b$ is already inferred. However, different nodes are assigned to different integers. Therefore $S.a = T.b$ does not hold under this assignment. ■

We prove the completeness of `CloseInequal` by proving the following lemma which states that given a conjunction of atomic conditions $\mathcal{P}$ produced by Steps 1–2 of `Closure`$_{\mathcal{C}}$, there is no inequality $S.a\ \theta\ T.b$ that is implied by $\mathcal{P}$ but is not in $\mathcal{P}^+$ (the output of `CloseInequal`). The proof extends the one presented in [Ull89] which only handled selection, join and semi-join conditions.

Lemma A.2 *Let $\mathcal{P}$ be a conjunction of atomic conditions produced by Steps 1–2 of* `Closure`$_{\mathcal{C}}$*, such that* `false` *is not implied by $\mathcal{P}$. Then every inequality $S.a\ \theta\ T.b$ not in $\mathcal{P}^+$ has some assignment of a set of integers to each attribute used in $\mathcal{P}$ that makes all the atomic conditions in $\mathcal{P}^+$ true but $S.a\ \theta\ T.b$ false.* □

Proof (Lemma A.2 (Completeness of `CloseInequal`*)):*
The inequality $S.a\ \theta\ T.b$ can be of three types. For now we assume that neither S nor T are universally quantified tuple variables.

Case 1: θ is $\leq$. We now construct an assignment that satisfies $\mathcal{P}^+$ but makes $S.a > T.b$. Let $\mathcal{A}$ be those attributes $U.c$ for which $S.a \leq U.c$ is in $\mathcal{P}^+$, and let $\mathcal{B}$ be those attributes $V.d$ for which $V.d \leq T.b$ is in $\mathcal{P}^+$. Let $\mathcal{C} = \mathcal{V} - \mathcal{A} - \mathcal{B}$, where $\mathcal{V}$ is the set of attributes used by $\mathcal{P}$. Note that for any attribute $U.c \in \mathcal{A}$, $V.d \in \mathcal{B}$, and $W.e \in \mathcal{C}$, it is possible that $V.d \leq W.e$ and/or $W.e \leq U.c$, but not $W.e \leq V.d$ (else by Axiom I7, $W.e$ would be in $\mathcal{B}$), nor $U.c \leq W.e$ (then, $W.e$ would be in $\mathcal{A}$). Also, $\mathcal{A}$ and $\mathcal{B}$ are disjoint since otherwise $S.a \leq T.b$ would be in $\mathcal{P}^+$ (by Axiom I7) contrary to our assumption. Since $\mathcal{C}$ is disjoint from $\mathcal{A}$ and $\mathcal{B}$, we conclude that all three attribute sets are disjoint.

We can now topologically sort the elements of each attribute set with respect to the order $\leq$. That is, $U.c \in \mathcal{A}$ comes before $U'.c' \in \mathcal{A}$ if $U.c \leq U'.c'$ is in $\mathcal{P}^+$ (and likewise for the elements of $\mathcal{B}$ and $\mathcal{C}$). There may be cycles in the order, i.e., we derive both $U.c \leq U'.c'$ and $U'.c' \leq U.c$. In this case, it is guaranteed that one of U and U' is universally quantified. U and U' cannot be both existentially quantified since Step 2 of $\texttt{Closure}_{\mathcal{C}}$ picks only one attribute of one existentially quantified variable from each equivalence class. Given that one of U and U' is universally quantified, we break the cycle arbitrarily by assuming $U.c$ comes before $U'.c'$.

We can then order the attributes in $\mathcal{V}$ as follows: (1) the elements in $\mathcal{B}$, in order; (2) the elements in $\mathcal{C}$, in order; and (3) the elements in $\mathcal{A}$, in order. We can then initially assign distinct integers 1,2,... to the attributes in order. If some attribute $R.a \in \mathcal{V}$ maps to integer n, we denote this as $\text{IntMap}(R.a) = \{n\}$. For an attribute $R_i.a \in \mathcal{V}$ (i.e., R_i is an existentially quantified tuple variable), this is the final IntMap assignment of $R_i.a$. For an attribute $R_j^{asj}.a$, its final IntMap assignment depends on the equivalence classes determined in Step 1 of $\texttt{Closure}_{\mathcal{C}}$. If there is some attribute $S_i.b$ (attribute of some existentially quantified variable S_i) that is in the same equivalence class as $R_j^{asj}.a$, we set $\text{IntMap}(R_j^{asj}.a)$ to $\text{IntMap}(S_i.b)$. If there is no such attribute $S_i.b$ but there is an attribute $S_i^{asj}.b$ (attribute of some universally quantified variable S_i^{asj}), we set $\text{IntMap}(R_j^{asj}.a)$ to $\text{IntMap}(S_i^{asj}.b)$. If $R_j^{asj}.a$'s equivalence class has no other elements other than $R_j^{asj}.a$, we assign a set of integers to $R_j^{asj}.a$ as follows: $\text{IntMap}(R_j^{asj}.a) = \bigcup_{\forall R.a \in \mathcal{V}} \text{IntMap}(R.a)$, where R is either existentially quantified or universally quantified.

For this IntMap assignment, $S.a$ is given a larger value than $T.b$, so $S.a \leq T.b$ does not hold. Now we must show that all the atomic conditions in $\mathcal{P}^+$ hold.

Consider $U.c \neq V.d$ in $\mathcal{P}^+$. This clearly holds if $U.c$ and $V.d$ are in different attribute sets (i.e., $\mathcal{A}$, $\mathcal{B}$ or $\mathcal{C}$), because IntMap assigns a disjoint set of integers to attributes belonging to different attribute sets. If $U.c$ and $V.d$ are in the same attribute set, they must be in different equivalence classes as determined in Step 1 of $\texttt{Closure}_{\mathcal{C}}$. Otherwise, $U.c = V.d$ would have been derived by `CloseEqual` implying that $\mathcal{P}$ is contradictory. Furthermore, it cannot be the case that $U.c$ is actually $R_i.c$, and $V.d$ is actually $R_i.c$ as well. Otherwise, Axiom E1 would have derived $U.c = V.d$. Finally, it cannot be the case that $U.c$ is actually $R_i.c$, and $V.d$ is actually $R_j^{asj}.c$. This is because Axiom I9 would derive $U.c \neq U.c$, which indicates that $\mathcal{P}$ is contradictory. $\text{IntMap}(U.c) = \text{IntMap}(V.d)$ only holds if the attributes belong to the same equivalence class. $\text{IntMap}(U.c) \subset \text{IntMap}(V.d)$ only holds if $U.c$ is actually $R_i.c$, and $V.d$ is actually $R_j^{asj}.c$. Similarly, $\text{IntMap}(V.d) \subset \text{IntMap}(U.c)$ only holds if $U.c$ is actually $R_j^{asj}.c$, and $V.d$ is actually $R_i.c$. These three cases are avoided by using `CloseEqual` and Axiom I9. Hence, any $U.c \neq V.d$ in $\mathcal{P}^+$ always holds.

Now consider $U.c \leq V.d$ in $\mathcal{P}^+$. Let us suppose that $U.c$ and $V.d$ are in the same set of attributes ($\mathcal{A}$, $\mathcal{B}$, or $\mathcal{C}$). If both U and V are existentially quantified variables, then $U.c \leq V.d$ since the topological order within each attribute set respects $\leq$. If U is a universally quantified variable and V is not, $U.c \leq V.d$ holds. To see this, Axiom I9 ensures that $U.c \leq V'.d$ is derived for each tuple variable V' that goes over the same table as V. Hence, IntMap assigns a set of values to $V.d$, where each value in the set is greater or equal to than $\text{IntMap}(U.c)$. On the other hand, if U is a universally quantified variable and V is not, $U.c \leq V.d$ holds. To see this, Axiom I9 ensures that $U'.c \leq V.d$ is derived for each tuple variable U' that goes over the same table as U. Hence, IntMap assigns a set of values to $U.c$, where each value in the set is less than or equal to IntMap($V.d$). Finally, if both U and V are universally quantified, Axiom I9 ensures that $U'.c \leq V'.d$ is derived for each pair of tuple variables U' and V'. Hence, IntMap assigns a set of values to $U.c$ and a set of values to $V.d$, such that each value in IntMap($V.d$) is greater than each value in IntMap($U.c$). Note that if $U.c$ is actually $T_i^{asj}.c$ and $V.d$ is actually $T_j^{asj}.c$, then IntMap assigns the same singleton set of integers to $U.c$ and $V.d$. Hence, $U.c \leq V.d$ still holds.

So far, we have shown that $U.c \leq V.d$ holds if both $U.c$ and $V.d$ are in the same attribute set. We now show that it holds even if $U.c$ and $V.d$ are in different attribute sets. Surely, if $V.d$ is in $\mathcal{C}$, or $U.c$ is in $\mathcal{A}$ and $V.d$ is in $\mathcal{B}$ or $\mathcal{C}$, $U.c \leq V.d$ holds. We are left with the possibility that $U.c$ is in $\mathcal{A}$ and $V.d$ is in $\mathcal{C}$ or $\mathcal{B}$, or $U.c$ is in $\mathcal{C}$ and $V.d$ is in $\mathcal{B}$. However, if $U.c$ is in $\mathcal{A}$ and $U.c \leq V.d$ is in $\mathcal{P}^+$, then $V.d$ would be in $\mathcal{A}$ by Axiom I7, and not in $\mathcal{B}$ nor $\mathcal{C}$. Similarly, if $V.d$ is in $\mathcal{B}$ then it is not possible that $U.c \leq V.d$ and $U.c$ is in $\mathcal{C}$, because $U.c$ would have to be in $\mathcal{B}$, by Axiom I7.

Finally, we must consider $U.c < V.d$ in $\mathcal{P}^+$. We can rule out the possibility that $U.c$ is actually $R.c$ and $V.d$ is actually $R'.c$, and one of R or R' is universally quantified. Otherwise, we can derive either $R.c < R.c$ or $R'.c < R'.c$ which implies that $\mathcal{P}$ is contradictory. With these possibilities ruled out, the argument that $U.c \leq V.d$ is true holds for $U.c < V.d$ as well.

Case 2: θ is $\neq$. We now construct an assignment that satisfies $\mathcal{P}^+$ but makes $S.a = T.b$. Once the construction is done, many of the arguments for Case 1 hold for the present case as well. Since the present case considers that $S.a$ and $T.b$ are not equal, let us suppose that $S.a$ is less than $T.b$. Let $\mathcal{D}$ be those attributes $W.e$ such that $S.a \leq W.e$ and $W.e \leq T.b$ are in $\mathcal{P}^+$, which includes $S.a$ and $T.b$ themselves. Let $\mathcal{A}$ be those attributes $U.c$ for which $X \leq U.c$ is in $\mathcal{P}^+$, for some $X \in \mathcal{D}$, but $U.c$ itself is not in $\mathcal{D}$. Let $\mathcal{B}$ be those attributes $V.d$ for which $V.d \leq X$ is in $\mathcal{P}^+$, for some $X \in \mathcal{D}$, but $V.d$ itself is not in $\mathcal{D}$. Let $\mathcal{C} = \mathcal{V} - \mathcal{A} - \mathcal{B} - \mathcal{D}$, where $\mathcal{V}$ is the set of attributes used in $\mathcal{P}$. As in Case 1, it can be easily shown that $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$ are disjoint based on the axioms.

We then topologically sort the elements in each attribute set w.r.t. $\leq$. We then combine the attributes into one sequence with the attributes in $\mathcal{B}$ first, $\mathcal{C}$ second, $\mathcal{D}$ third and $\mathcal{A}$ last. We initially assign increasing distinct integers to each attribute except for the attributes in $\mathcal{D}$, where the same integer is assigned. The IntMap function introduced in Case 1 is used to give the final assignments to each attribute.

Clearly, $S.a = T.b$ since both $S.a$ and $T.b$ are in $\mathcal{D}$. We now show that all the atomic conditions in $\mathcal{P}^+$ hold.

Consider $U.c \neq V.d$ in $\mathcal{P}^+$. As in Case 1, if $U.c$ and $V.d$ are in different attribute sets (i.e., $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$), then $U.c \neq V.d$ holds. If $U.c$ and $V.d$ belong to the same attribute set, the argument given in Case 1 that $U.c \neq V.d$ holds for attribute sets $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. Although the argument does not hold for $\mathcal{D}$, $U.c$ and $V.d$ cannot be in $\mathcal{D}$. If they were, Axiom I8 derives that $S.a \neq T.b$ contradicting our assumption that $S.a \neq T.b$ is not in $\mathcal{P}^+$.

Consider $U.c \leq V.d$. If $U.c$ and $V.d$ are in the same attribute set, the argument given in Case 1 holds if $U.c$ and $V.d$ are either in $\mathcal{A}$, $\mathcal{B}$ or $\mathcal{C}$. Since all the attributes in $\mathcal{D}$ are assigned the same integer, $U.c \leq V.d$ holds if $U.c, V.d \in \mathcal{D}$.

So far, we have shown that $U.c \leq V.d$ holds if both $U.c$ and $V.d$ are in the same attribute set. We now show that it holds even if $U.c$ and $V.d$ are in different attribute sets. Clearly, there are many cases where $U.c$ and $V.d$ reside in different attributes sets, and $U.c \leq V.d$ still holds by virtue of the ordering imposed on the sets (i.e., $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{A}$). For instance, if $U.c$ is in $\mathcal{D}$ and $V.c$ is in $\mathcal{A}$, then surely $U.c \leq V.d$ holds. We now consider the following possibility – $U.c$ is in $\mathcal{A}$ and $V.d$ is in some other attribute set. Hence, $X \leq U.c$, where $X \in \mathcal{D}$, must be in $\mathcal{P}^+$. However, $X \leq V.d$ is derived by Axiom I7 and $V.d$ must also be in $\mathcal{A}$. We now consider the following possibility – $V.d$ is in $\mathcal{B}$ and $U.c$ is either in $\mathcal{C}$ or $\mathcal{D}$. In this case, Axiom I7 enforces that $U.c$ must also be in $\mathcal{B}$. We are left with the possibility that $U.c$ is in $\mathcal{D}$, and $V.d$ is in $\mathcal{B}$. Since $U.c \leq V.d$ is in $\mathcal{P}^+$ and $U.c \in \mathcal{D}$, by definition $V.d$ must be in $\mathcal{A}$. Because the sets are ordered as $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, then $\mathcal{A}$, $U.c \leq V.d$ must hold.

Finally, we must consider $U.c < V.d$ in $\mathcal{P}^+$. As in Case 1, we can rule out the possibility that $U.c$ is actually $R.c$ and $V.d$ is actually $R'.c$, and one of R or R' is universally quantified. Otherwise, we can derive either $R.c < R.c$ or $R'.c < R'.c$ which implies that $\mathcal{P}$ is contradictory. With these possibilities ruled out, the argument that $U.c \leq V.d$ is true holds for $U.c < V.d$ as well.

Case 3: θ is $<$. If $S.a \leq T.b$ is not in $\mathcal{P}^+$, then use the construction of Case 1 where IntMap makes all the atomic conditions in $\mathcal{P}^+$ true but makes $S.a < T.b$ false (i.e., $S.a > T.b$ is true). If $S.a \neq T.b$ is not in $\mathcal{P}^+$, then use the construction of Case 2 where IntMap makes all atomic conditions in $\mathcal{P}^+$ true but makes $S.a \neq T.b$ false (i.e., $S.a = T.b$ is true). If both $S.a \leq T.b$ and $S.a \neq T.b$ is in $\mathcal{P}^+$, then by Axiom I4, $S.a < T.b$ is in $\mathcal{P}^+$ as well contrary to our assumption.

So far, we have assumed that neither S nor T in $S.a\ \theta\ T.b$ is universally quantified. Given this assumption, we have proved that if $S.a\ \theta\ T.b$ is not in $\mathcal{P}^+$, then there is an assignment that makes all the atomic conditions in $\mathcal{P}^+$ true but not $S.a\ \theta\ T.b$. We note that the IntMap assignment that makes $S.a\ \theta\ T.b$ false can be used if S (or T) is universally quantified. Suppose S_i^{asj} is a universally quantified tuple variable going over the same table as S. Since $S_i^{asj}.a\ \theta\ T.b$ implies $S.a\ \theta\ T.b$, the IntMap assignment that makes $S.a\ \theta\ T.b$ false also makes $S_i^{asj}.a\ \theta\ T.b$ false. ∎

B Aggregates

We now show how the maintenance queries of aggregate views can be decomposed into maintenance expressions of the form $\pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R)$, expected by our algorithms. To illustrate, let us begin with a view V_{sum} defined as

$$\pi_{a,\texttt{SUM}(b)\ \texttt{as}\ sum,\texttt{COUNT}()\ \texttt{as}\ cnt}(T).$$

To compute the deletions and insertions to V_{sum}, the following maintenance queries are used.

$$\bigtriangledown V_{sum} \leftarrow \pi_{\texttt{Distinct(Attrs(}V_{sum}\texttt{))}}(\sigma_{V_{sum}.a=\triangle T.a}(V_{sum}\times\triangle T) \cup \sigma_{V_{sum}.a=\bigtriangledown T.a}(V_{sum}\times\bigtriangledown T)) \quad (16)$$

$$\triangle V_{sum} \leftarrow \sigma_{cnt>0}\pi_{a,\texttt{SUM}(sum)\ \texttt{as}\ sum,\texttt{SUM}(cnt)\ \texttt{as}\ cnt} (\bigtriangledown V_{sum} \cup \pi_{a,\texttt{SUM}(b)\ \texttt{as}\ sum,+1\ \texttt{as}\ cnt}(\triangle T) \cup \pi_{a,-\texttt{SUM}(b)\ \texttt{as}\ sum,-1\ \texttt{as}\ cnt}(\bigtriangledown T)) \quad (17)$$

These maintenance queries are derived from [Qua96]. The `Distinct` function in Query (16) is for duplicate elimination. We consider the following subqueries as the maintenance expressions for $\bigtriangledown V_{sum}$: $\sigma_{V_{sum}.a=\triangle T.a}(V_{sum}\times\triangle T)$ and $\sigma_{V_{sum}.a=\bigtriangledown T.a}(V_{sum}\times\bigtriangledown T)$. The maintenance expressions for $\triangle V_{sum}$ are: $\bigtriangledown V_{sum}$, $\pi_{a,\texttt{SUM}(b)\ \texttt{as}\ sum,+1\ \texttt{as}\ cnt}(\triangle T)$, and $\pi_{a,-\texttt{SUM}(b)\ \texttt{as}\ sum,-1\ \texttt{as}\ cnt}(\bigtriangledown T)$. Given these maintenance expressions as "building blocks", we can compute the original maintenance queries. In other words, by keeping data "needed" for these maintenance expressions, we will never miss any data needed for the maintenance queries. In general, we do in fact have to keep all the data needed for the maintenance expressions in order to compute the the maintenance queries. However, if we augment our constraint language to allow the use of operator $\pi_{\mathcal{A}}$, where $\mathcal{A}$ contains aggregates, more optimization might become possible. We plan to investigate this problem in the future.

Next, suppose we have a view V_{avg} defined as $\pi_{a,\texttt{AVG}(b)\ \texttt{as}\ avg,\texttt{SUM}(b)\ \texttt{as}\ sum,\texttt{COUNT}()\ \texttt{as}\ cnt}(T)$. The maintenance query for computing $\bigtriangledown V_{avg}$ is identical to Query (16) except that references to V_{sum} are replaced by V_{avg}. The maintenance query for computing $\triangle V_{avg}$ is changed to

$$\triangle V_{avg} \leftarrow \sigma_{cnt>0}\pi_{a,\texttt{SUM}(sum)/\texttt{SUM}(cnt)\ \texttt{as}\ avg,\texttt{SUM}(sum)\ \texttt{as}\ sum,\texttt{SUM}(cnt)\ \texttt{as}\ cnt} (\bigtriangledown V_{avg} \cup \pi_{a,\texttt{SUM}(b)\ \texttt{as}\ sum,+1\ \texttt{as}\ cnt}(\triangle T) \cup \pi_{a,-\texttt{SUM}(b)\ \texttt{as}\ sum,-1\ \texttt{as}\ cnt}(\bigtriangledown T)). \quad (18)$$

It is not hard to see that the maintenance queries of V_{avg} can be decomposed in a similar fashion as the maintenance queries of V_{sum}.

Note that for aggregates over a single table T with only the AVG, SUM and COUNT aggregate functions, T itself is not referred to by the maintenance queries. These views are often called *self-maintainable* views ([QGMW96]). Our function $\texttt{Map}(E,T)$ will return $\{\}$ for all these maintenance expressions.

This is not true for the maintenance expressions of an aggregate view that uses MAX or MIN. This is because in general, such aggregate views are not even incrementally maintainable in the presence of deletions. For instance, suppose V_{max} is defined as $\pi_{a,\texttt{MAX}(b) \texttt{ as } sum,\texttt{COUNT}() \texttt{ as } cnt}(T)$. Feasible maintenance queries for V_{max} is $\bigtriangledown V_{max} \leftarrow V_{max}$ and $\triangle V_{max} \leftarrow \pi_{a,\texttt{MAX}(b) \texttt{ as } sum,\texttt{COUNT}() \texttt{ as } cnt}(T \cup \triangle T - \bigtriangledown T)$. That is, V_{max} is simply recomputed. In this case, the maintenance expressions of V_{max} are V_{max}, T, $\triangle T$ and $\bigtriangledown T$. The maintenance queries can be improved to only recompute a group when the actual maximum value of the group is deleted from T ([Qua96]). The improved maintenance queries will lead to maintenance expressions that do not access the entire table T.

In summary, for aggregate views over a single table T, their maintenance queries can be easily decomposed into maintenance expressions. In particular, aggregate views with just the AVG, SUM and COUNT functions can be easily be made self-maintainable, and their maintenance expressions do not even need to be considered in $\texttt{Needed}(T,\mathcal{E})$! For views with MAX and MIN, obtaining their maintenance expressions is also straightforward. Since their maintenance expressions do access T, it is beneficial to start with the improved maintenance queries given in [Qua96].

So far we have discussed how to obtain the maintenance expressions of an aggregate view defined over a single table T. Now let us suppose that V_{sum2} is defined as

$$\pi_{R.a,\texttt{SUM}(S.b) \texttt{ as } sum,\texttt{COUNT}() \texttt{ as } cnt}\sigma_{R.a=S.c}(R \times S). \tag{19}$$

To handle these views, the maintenance queries ([GL95],[Qua96]) are derived by treating V_{sum2} as if it was defined as $\pi_{V_{temp}.a,\texttt{SUM}(V_{temp}.b) \texttt{ as } sum,\texttt{COUNT}() \texttt{ as } cnt}(V_{temp})$. The "virtual" view V_{temp} is defined as $\pi_{R.a,S.b}\sigma_{R.a=S.c}(R \times S)$, and it is maintained before V_{sum2}. The following standard maintenance queries are used to compute the insertions and deletions to a virtual view V_{temp}.

$$\sigma_{\triangle R.a=S.c}(\triangle R \times S) \cup \sigma_{R.a=\triangle S.c}(R \times \triangle S) \cup \sigma_{\triangle R.a=\triangle S.c}(\triangle R \times \triangle S) \cup \sigma_{\bigtriangledown R.a=\bigtriangledown S.c}(\bigtriangledown R \times \bigtriangledown S)$$
$$\sigma_{\bigtriangledown R.a=S.c}(\bigtriangledown R \times S) \cup \sigma_{R.a=\bigtriangledown S.c}(R \times \bigtriangledown S) \cup \sigma_{\bigtriangledown R.a=\triangle S.c}(\bigtriangledown R \times \triangle S) \cup \sigma_{\triangle R.a=\bigtriangledown S.c}(\triangle R \times \bigtriangledown S)$$

The maintenance queries of V_{sum2} is then similar to the maintenance queries of V_{sum} above except the references to $\triangle T$ and $\bigtriangledown T$ are replaced with $\triangle V_{temp}$ and $\bigtriangledown V_{temp}$ respectively. Hence, the maintenance expressions of V_{sum2} are simply the maintenance expressions of V_{temp}.