Expiring Data in a Warehouse

Hector Garcia-Molina, Wilburt Juan Labio, Jun Yang
{hector, wilburt, junyang}@cs.stanford.edu
Computer Science Dept., Stanford, CA 94305

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.

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 further illustrated next.

Proceedings of the 24th VLDB Conference
New York, USA, 1998

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.
- $Order(ordId, custId, clerk)$, 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 $l_1.cost < 99$. 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 $ordId$

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.

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. □

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 description of the available requested data.

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.

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 π, σ, and $\times$ bag operators.)

For instance, we can define a view *ClerkCust* to obtain the sum of the recent, expensive 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.

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

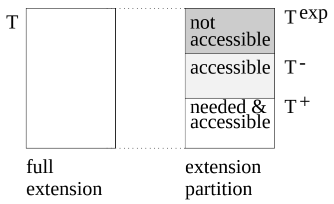

Figure 2: Extension Partition of T

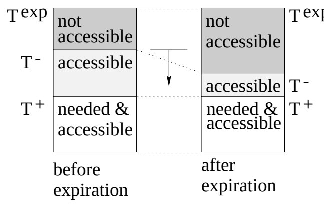

Figure 3: Effect of Expiration on T^- and T^{exp}

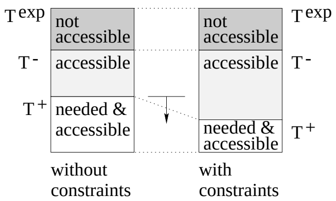

Figure 4: Effect of Constraints on T^+ and T^-

$$\sigma_{L.ordId=O.ordId \ \wedge \ O.custId=C.custId}$$
$$(C \times O \times L)$$

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 selection condition $L.cost > 99$, and join condition $L.ordId = O.ordId$. 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 changes 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 changes to table S are stored in *delta relation* tables $\triangle S$ and $\triangledown S$, where $\triangle S$ contains the new inserted and updated tuples, and $\triangledown S$ contains the old deleted and updated tuples. To incrementally maintain V, we compute $\triangle V$ and $\triangledown V$ using the *maintenance queries* shown below.

$$\sigma_{\triangle S.b=T.c}(\triangle S \times T) \ \cup \ \sigma_{\triangle S.b=\triangle T.c}(\triangle S \times \triangle T) \ \cup$$
$$\sigma_{S.b=\triangle T.c}(S \times \triangle T) \ \cup \ \sigma_{\triangledown S.b=\triangledown T.c}(\triangledown S \times \triangledown T) \quad (1)$$
$$\sigma_{\triangledown S.b=T.c}(\triangledown S \times T) \ \cup \ \sigma_{\triangledown S.b=\triangle T.c}(\triangledown S \times \triangle T) \ \cup$$
$$\sigma_{S.b=\triangledown T.c}(S \times \triangledown T) \ \cup \ \sigma_{\triangle S.b=\triangledown T.c}(\triangle S \times \triangledown T) \quad (2)$$

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.

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 keep the most recent expiration request $\sigma_{\mathcal{P}'}(T)$ on T in $LastReq(T)$. 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 $\sigma_{\mathcal{P} \vee \mathcal{P}'}(T)$. This is done because we do not "unexpire" any expired data in our framework.

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

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. 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.

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) which describe the contents of the delta relations $\triangle T$ and $\bigtriangledown T$ in a constraint specification language (Section 4). 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 and may set *Constraints*(T) to be empty.

Framework Summary: Table 2 gives a summary of the framework. Henceforth, we denote the set of all tables as $\mathcal{T}$, the set of all constraints as $\mathcal{C}$, and the set of all maintenance queries as $\mathcal{E}$. 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 these problems appear in [LGM97].

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 processed very efficiently by removing the tuples $\sigma_{\mathcal{P} \wedge needed=\text{false}}(T)$ from T's real extension.

Before we present how the needed tuples are identified, we introduce *maintenance expressions*, which are subqueries of maintenance queries. 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 [GMLY98], we show that the maintenance queries of aggregate views (such as *ClerkCust*) can also be decomposed into maintenance expressions. 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

Table 1: Summary of Framework

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
$\mathcal{C}$	$\bigcup_{T \in \mathcal{T}} Constraints(T)$
$\mathcal{E}$	$\bigcup_{\text{view } V \in \mathcal{T}} Maint(V)$

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 `Closure`$(\mathcal{P})$ returns the closure of the input conjunctive condition ([Ull89]).

Function `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 `true`. For instance, if $\mathcal{P}$ is $R.a > S.b \wedge S.b > T.c$, `Ignore`$(\mathcal{P}, \{S\})$ is `true` $\wedge$ `true` or simply `true`. Notice that `Ignore(Closure`$(\mathcal{P})$`)`$, \{S\})$ is $R.a > T.c$.

Function `Map`(E, T) acts on a maintenance expression E and returns a query that identifies the T tuples needed by E.

Definition 3.1 **(Map)** Let E be $\pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R \in \mathcal{R}} R)$, and $\mathcal{D}$ be the delta relations in $\mathcal{R}$. `Map`(E, T) is $\{\}$ if $T \notin \mathcal{R}$. Otherwise, `Map`(E, T) is $\pi_{\text{Attrs}(T)}\sigma_{\text{Ignore}(\text{Closure}(\mathcal{P}),\mathcal{D}')}(\times_{R \in (\mathcal{R}-\mathcal{D}')} R)$, where $\mathcal{D}'$ is $\mathcal{D} - \{T\}$. □

That is, if T is not referenced in E, `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, `Map` returns a new expression obtained by first removing the delta relations in $\mathcal{D}$ from the cross product. 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. Finally, the projected attributes is changed to `Attrs`(T), the attributes of the table T.

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

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

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

Note that `Needed` may list a needed tuple $t \in T$ more times than t appears in T. Hence, the semijoin ($\ltimes$) operation, which is equivalent to an `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 [GMLY98]. 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.

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

Let us consider what L tuples are needed by E. We claim that `Map`(E, L), shown below, identifies all these L tuples.

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

Notice that `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 `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, `Map`(E, L) has the atomic condition $L.ordId > 1000$.

While $\mathtt{Map}(E, L)$ identifies all the needed L tuples, it may list an L tuple t_L more times than t_L appears in L. For instance, $\mathtt{Map}(E, L)$ performs a cross product between C and L without applying any conditions between them. Hence, $\mathtt{Map}(E, L)$ lists t_L as many times as there are C tuples. In report [GMLY98], we discuss how to make $\mathtt{Map}(E, L)$ more efficient by avoiding cross products. Thus, to obtain the correct bag of tuples, the query $L \ltimes_{\mathrm{Attrs}(L)} \mathtt{Map}(E, L)$ is used. □

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 $\mathtt{Needed}(T, \mathcal{E})$ needs to be modified to $\mathtt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$ 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 constraint language *CL* for specifying the constraints in $\mathcal{C}$. In Section 4.2, we give the algorithm that uses $\mathcal{C}$ to produce $\mathcal{E}_{\mathcal{C}}$ from $\mathcal{E}$. We present in Section 4.3 the function $\mathtt{Needed}_{\mathcal{C}}$ that acts on $\mathcal{E}_{\mathcal{C}}$.

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.

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

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` 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 as we illustrate next. (We refer the reader to [GMLY98] for a discussion of *CL*'s expressibility.) Furthermore, we will see that *CL*'s syntax is particularly well suited for modifying maintenance expressions.

EXAMPLE 4.1 Figure 5 gives 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. We now give the intuition behind each constraint.

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 ∇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. 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 ($\triangle O$ tuples). Constraints (6) and (7) are used to describe the insertions to L. That is, inserted L tuples that join with $\triangle O$ have *ordId* values greater than the maximum *ordId*. Inserted L tuples that join with O have *ordId* values equal to the maximum *ordId*.

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.

Referential integrity constraints: Given the schema introduced in Example 1.1, it is reasonable to assume that there is a referential integrity constraint from $O.custId$ to key $C.custId$. Constraints (11) to (13) express this assumption. Similar constraints are used to express a referential integrity constraint from attribute $L.ordId$ to key $O.ordId$.

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 (Constraint (14)). □

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 $LHS(c)$

$$\nabla O \equiv \sigma_{\text{false}}(\nabla O) \tag{3}$$
$$\triangle O \equiv \triangle O \overline{\ltimes}_{\triangle O.ordId \leq O.ordId} O \tag{4}$$
$$\nabla L \equiv \sigma_{\text{false}}(\nabla L) \tag{5}$$
$$\sigma_{\triangle O.ordId=\triangle L.ordId}(\triangle O \times \triangle L) \equiv \sigma_{\triangle O.ordId=\triangle L.ordId}(\triangle O \times (\triangle L \overline{\ltimes}_{\triangle L.ordId \leq O.ordId} O)) \tag{6}$$
$$\sigma_{O.ordId=\triangle L.ordId}(O \times \triangle L) \equiv \sigma_{O.ordId=\triangle L.ordId}(O \times (\triangle L \overline{\ltimes}_{\triangle L.ordId < O.ordId} O)) \tag{7}$$
$$C \equiv C \overline{\ltimes}_{(C.custId=C'.custId) \wedge (C.info \neq C'.info)} \rho_{C'}(C) \tag{8}$$
$$\nabla C \equiv \nabla C \overline{\ltimes}_{(\nabla C.custId=\nabla C'.custId) \wedge (\nabla C.info \neq \nabla C'.info)} \rho_{\nabla C'}(\nabla C) \tag{9}$$
$$\triangle C \equiv \triangle C \overline{\ltimes}_{\triangle C.custId=C.custId} C \tag{10}$$
$$O \equiv O \ltimes_{O.custId=C.custId} C \tag{11}$$
$$\triangle O \equiv \triangle O \ltimes_{\triangle O.custId=C.custId} C \tag{12}$$
$$\nabla O \equiv \nabla O \ltimes_{\nabla O.custId=C.custId} C \tag{13}$$
$$\nabla C \equiv \nabla C \ltimes_{(\nabla C.custId=C.partId) \wedge (\nabla C.info=C.info)} C \tag{14}$$

Figure 5: Example *CL* Constraints

"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. The next example illustrates 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) simply requires finding occurrences of ∇O in a maintenance expression E and replacing it with $\sigma_{\text{false}}(\nabla O)$. To make the example more interesting, let us suppose constraint c is Constraint (7), and apply it to the following maintenance expression E of *ClerkCust*.

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

Maintenance expression E can be rewritten as

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

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

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

The previous example illustrated algorithm *Apply* (Algorithm 4.1, Figure 6) 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). This check suffices since we only handle view definitions with no self-joins. 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]. (It can be done in $O(n^3)$ time, where n is the number of distinct attributes in the conditions.) If both checks are passed, then $LHS(c)$ matches a subquery of E. For instance, suppose that E is $\pi_{\mathcal{A}} \sigma_{\mathcal{P}}(\times_{R \in \mathcal{R}} R) \ltimes S..\overline{\ltimes} T..$, 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 $\pi_{\mathcal{A}} \sigma_{\mathcal{P}}((\times_{R \in (\mathcal{R}-\mathcal{U})} R) \times \sigma_{\mathcal{P}_{LHS}}(\times_{U \in \mathcal{U}} U)) \ltimes S..\overline{\ltimes} T...$

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.

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 $\ltimes$ and $\overline{\ltimes}$ 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.

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: $\pi_A(\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R) \ltimes S..\overline{\ltimes}T)$
Let c be of the form:
  $\sigma_{\mathcal{P}_{LHS}}(\times_{U\in\mathcal{U}}U) \equiv \sigma_{\mathcal{P}_{RHS}}(\times_{U\in\mathcal{U}}U) \ltimes V$
1. If $\mathcal{U} \subseteq \mathcal{R}$
  2. If $\mathcal{P} \Rightarrow \mathcal{P}_{LHS}$
    3. Remove conditions in $\mathcal{P}$ implied by $\mathcal{P}_{RHS}$
    4. $E \leftarrow \pi_A(\sigma_{\mathcal{P}\wedge\mathcal{P}_{RHS}}(\times_{R\in\mathcal{R}}R))$
       $\ltimes S..\ltimes V..\overline{\ltimes}T$
    5. Return `true`
6. Return `false` $\diamond$

Algorithm 4.2 *Modify*
Input: maint. expression E, 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 6: Modifying a Maintenance Expression

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 ([GMLY98]). Algorithm 4.2 (Figure 6) 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 $\texttt{Needed}_{\mathcal{C}}$

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. We then develop a fairly efficient $\texttt{Needed}_{\mathcal{C}}$ function which handles `exists` and some `not exists` conditions. In the latter part of the section, we give a lemma that formally describes the properties of $\texttt{Needed}_{\mathcal{C}}$.

Problem with `Needed`: Strictly speaking, `Needed` was not defined to work with maintenance expressions with `exists` and `not exists` conditions. Nevertheless, function $\texttt{Needed}(T, \mathcal{E}_{\mathcal{C}})$ can be adapted to apply to $\mathcal{E}_{\mathcal{C}}$ by modifying $\texttt{Map}(E,T)$. That is, for each $E = \pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R) \ltimes S..\overline{\ltimes}U..$ in $\mathcal{E}_{\mathcal{C}}$, $\texttt{Map}(E,T)$ returns the following query.

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

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.

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 constraints to

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

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:

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

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

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

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 $E_{\mathcal{C}}$.

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

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

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

This query has an empty answer because the `exists` condition on O is contradictory! Hence, $\texttt{Map}_{\mathcal{C}}(E_{\mathcal{C}}, O)$

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. For instance, given the maintenance expression $E_{\mathcal{C}}$ as shown Example 4.3, its quantifier representation is

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

$\mathcal{P}'$ in this case is

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

We assign the tuple variables mechanically as follows. For a table T appearing in the cross product (*e.g.*, $\triangle C$), we assign the tuple variable T_0 (*e.g.*, $\triangle C_0$). 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}). 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 $\exists R_i..\exists S_j..\forall T_k^{asj}..\forall U_l^{asj}(\mathcal{P}')$, where $\mathcal{P}'$ is a quantifier-free condition.

Assuming $\mathcal{P}'$ is conjunctive for now, $\texttt{Closure}_{\mathcal{C}}$ simply derives new atomic conditions from ones 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 (*e.g.*, transitivity) to derive atomic conditions. $\texttt{Closure}_{\mathcal{C}}$ adds the following two axioms to derive additional atomic conditions from ones that use universally quantified variables.

1. Let θ be $=, \neq, \leq, <, \geq$, or $>$. $S_i^{asj}.a \ \theta \ T.b \Rightarrow S.a \ \theta \ T.b$.
2. $S_i^{asj}.a = T_j.b \Rightarrow S_i^{asj}.a = S_j^{asj}.a$.

The first axiom states that if $S_i^{asj}.a \ \theta \ T.b$ holds, it means that 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. We now illustrate $\texttt{Closure}_{\mathcal{C}}$.

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). 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. 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 $O_0.custId \neq C_1.custId$ are derived. Hence, $\texttt{Map}(O, E)$ is guaranteed to return an empty answer which is consistent with Example 4.3. □

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 $\mathcal{P}'$ based on the axioms used by `Closure` plus the two additional axioms for universally quantified tuple variables.
3. Return $\exists..\exists..\forall..\forall..(\texttt{Closure}(\mathcal{P}''))$ ◇

Figure 7: $\texttt{Closure}_{\mathcal{C}}$

The example illustrated $\texttt{Closure}_{\mathcal{C}}$ (Algorithm 4.3, Figure 7) 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 is a single atomic condition or a disjunction of atomic conditions. Any `not exists` condition that does not conform to the previous restriction is ignored (replaced with `true`) when computing the closure. Using the axioms used by

$\texttt{Closure}$ plus the two additional axioms introduced, $\texttt{Closure}_{\mathcal{C}}$ derives the atomic conditions implied by $\mathcal{P}'$ (Line 2). We refer the reader to [GMLY98] for more details on how the axioms are applied.

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 $\texttt{Map}_{\mathcal{C}}$. The next lemma formally describes the properties of $\texttt{Needed}_{\mathcal{C}}$. (See [GMLY98] for the proof.)

Lemma 4.1 *Given a table T and a set of maintenance expression $\mathcal{E}_{\mathcal{C}}$ obtained by applying constraints $\mathcal{C}$ on $\mathcal{E}$, 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 $\sigma_{\mathcal{P}_{LHS}}(\times_{R \in \mathcal{R}} R) \equiv \sigma_{\mathcal{P}_{RHS}}(\times_{R \in \mathcal{R}} R) \overline{\ltimes}_p T$ where p is a disjunction of atomic conditions, the query $T \ltimes_{\mathrm{Attrs}(T)} \texttt{Needed}(T, \mathcal{E})$ 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 $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T, \mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T, \mathcal{E})$.* □

5 Discussion

Although Lemma 4.1 itself does not guarantee that $\texttt{Needed}_{\mathcal{C}}$ always returns strictly fewer tuples than $\texttt{Needed}$, 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 $\mathcal{E}$. $\mathcal{C}$ are the various in Example 4.1. Table 5 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 5 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. The third row of Table 5 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 5 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 $\mathcal{E}$. Finally, the set of constraints $\mathcal{C}$ we consider is based on the TPC-D "update model" specification.

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 5 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 5 shows similar results assuming the maintenance expressions of view V_5 comprise $\mathcal{E}$. Note 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 appropriate constraints, 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.

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. 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, an expiration request $\sigma_{\mathcal{P}}(T)$ is satisfied by removing the tuples in $\sigma_{\mathcal{P} \wedge needed=\mathrm{false}}(T)$.

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 ex-

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	$\{\}$	$\pi_{\mathrm{Attrs}(L)} \sigma_{L.cost>99 \wedge L.ordId>1000}(L)$
O	$\pi_{\mathrm{Attrs}(O)} \sigma_{O.custId<500 \wedge O.ordId>1000}$ $(O \ltimes_{O.ordId<O'.ordId} \rho_{O'} O)$	$\pi_{\mathrm{Attrs}(O)} \sigma_{O.custId<500 \wedge O.ordId>1000}(O)$
C	$\pi_{\mathrm{Attrs}(C)} \sigma_{C.custId<500}(C)$	$\pi_{\mathrm{Attrs}(C)} \sigma_{C.custId<500}(C)$

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	20%	20%
P	0%	0%

pressions 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 $\sigma_{needed=\mathrm{false}}(T) \ltimes_{\mathrm{Attrs}(T)} \texttt{Needed}(T, \mathcal{E}_V)$, where $\mathcal{E}_V$ are the maintenance expressions of V. This query identifies the T tuples that are now 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 $LHS(c)$ or added to $RHS(c)$. To update the extension markings, for each table T, we use the query $\sigma_{needed=\mathrm{true}}(T) \overline{\ltimes}_{\mathrm{Attrs}(T)} \texttt{Needed}_{\mathcal{C}}(T, \mathcal{E})$, to identify the T tuples that were previously deemed needed, but must now be marked as unneeded. 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 if the administrator inputs all constraints. If this assumption does not hold, we show in [GMLY98] how to identify the tables whose extension marking may be modified.

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 `false` for all inserted tuples.

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

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%

2. Set the *needed* = `true` for the T tuples in $\sigma_{needed=\mathrm{false}}(T) \ltimes_{\mathrm{Attrs}(T)} \texttt{Needed}(T, \mathcal{E})$.
3. Expire T tuples in $\sigma_{\mathcal{P} \wedge needed=\mathrm{false}}(T)$, 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).

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, GL95,GMS93,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] and [HZ96] 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 a table 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]. 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 infer that the entire chronicles can be safely expired. In summary, the techniques we introduce generalize special-case algorithms. Furthermore, since we exploit a broader class of constraints, we improve on many of the algorithms.

Our framework also introduces "incomplete" tables. There has been numerous work on incomplete databases ([AHV95]). 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.

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.

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 at: http://www.tpc.org/.

[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.

[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.

[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.

[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.

[LGM97] W. Labio and H. Garcia-Molina. Expiration and partially materialized views. Technical report, Stanford University, 1997. Available at http:-//www-db.stanford.edu/pub/papers/expire.ps.

[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.

[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.

[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.