Paper Number AR_299

Expiring Data from the Warehouse

Wilburt Juan Labio* Hector Garcia-Molina†

Abstract

Data warehouses are used to collect and analyze data from remote sources. The data collected often originate from transactional information and can become very large. This paper presents a framework for incrementally removing warehouse data (without a need to fully recompute views), offering two choices. One is to expunge data, in which case the result is as if the data had never existed. The second is to expire data, in which case views defined over the data are not necessarily affected. Within the framework, a user or administrator can specify what data to expire or expunge, what auxiliary data is to be kept for facilitating incremental view maintenance, what type of updates are expected from external sources, and how the system should compensate when data is expired or other parameters changed. We present algorithms for the various expiration and compensation actions, and we show how our framework can be implemented on top of a conventional RDBMS. **Keywords**: view maintenance, data warehouse

1 Introduction

The amount of data copied into a warehouse may be very large; for instance, [JMS95] cites a major telecommunications company that collects 75GB of data every day or 27TB a year. Even with cheap disks, in many cases it will be desirable to "remove" from the on-line warehouse some of the data that is no longer of interest or relevant. There are two basic methods for removing unneeded data: *expunction* and *expiration*. We illustrate the difference between these two methods in the following example.

EXAMPLE 1.1 A source contains a base relation on "shopping markets" of interest:

- $market(\underline{mID}, s, e)$. A tuple $\langle mID, s, e\rangle$ is in this relation if the market mID with an employee count of e is in state s (e.g., "CA"). The key of the view is mID.

The warehouse administrator (WHA) defines a materialized view so that this information can be easily accessible at the warehouse:

CREATE VIEW *MCopy* AS
  SELECT *
  FROM *market*

Furthermore, the WHA defines the following view to summarize employee averages by state:

CREATE VIEW *EmpAvg* AS
  SELECT s, $AVG(e)$
  FROM *MCopy*
  GROUPBY s

*Stanford University Department of Computer Science. e-mail: wilburt@cs.stanford.edu

†Stanford University Department of Computer Science. e-mail: hector@cs.stanford.edu

mID	s (state)	e (emp. count)
A	CA	100
B	CA	150
C	OR	26

Figure 1: *MCopy* View

s	avg
CA	125
OR	26

Figure 2: *EmpAvg* View

mID	s (state)	e (emp. count)
A	CA	100
B	CA	150

Figure 3: *MCopy* after Expiration of $\langle C, OR, 26\rangle$

s	cnt
OR	1

Figure 4: *EmpAvg* Auxiliary View after Expiration of $\langle C, OR, 26\rangle$

Figures 1 and 2 illustrate the views on a small example.

Next, assume that materialized view *MCopy* has become too large, and the WHA wants to "remove" (perhaps archive to tape) warehouse tuples that are no longer of interest. Say that the tuples to be removed are those from the state of Oregon. In Figure 1 this is tuple $\langle C, OR, 26\rangle$. If the tuple is *expired*, it still logically exists in *MCopy* except that it is unavailable. Thus, the view *EmpAvg* is unaffected by the expiration. Figure 3 shows the state of *MCopy* after the expiration, while Figure 2 still shows the state of *EmpAvg*. On the other hand, if tuple $\langle C, OR, 26\rangle$ is *expunged*, then its disappearance is visible to views defined on *MCopy*. In this case, the $\langle OR, 26\rangle$ tuple would also be removed from *EmpAvg*. In other words, expunction effectively changes the specification of view *MCopy* to exclude the Oregon data, while expiration does not.

Note that both expiration and expunction at the warehouse are different from *deletion* of base data. In the two examples of the previous paragraph, base relation *market* did not change: tuple $\langle C, OR, 26\rangle$ still exists there. We use the term deletion only for removal of base data. A deletion of tuple $\langle C, OR, 26\rangle$ would need to be propagated to the warehouse, and the impact would be similar to expunging the tuple from *MCopy*. However, in this case the specification of *MCopy* does not change: it is still a full copy of *market* (it is just that *market* has gotten smaller).

It is important to notice that expiration may complicate future maintenance of the warehouse. To illustrate, assume that after $\langle C, OR, 26\rangle$ is expired, a new tuple $\langle D, OR, 52\rangle$ is inserted into *market*. When the warehouse is informed of the insertion, it must compute a new average for Oregon tuples in *EmpAvg*. However, since tuple $\langle C, OR, 26\rangle$ is not available, it cannot readily compute the new average. This means that when data is expired, either it must be known that future updates like the one illustrated will not occur, or the warehouse needs to create an *auxiliary view* to help with future updates. In our example, the auxiliary view needs to save the count of Oregon tuples, as shown in Figure 4. (With the count, and the running average in *EmpAvg*, we can incrementally compute the new averages without having to access the actual tuples in *MCopy*.) Thus, the auxiliary view saves the "portion" of the expired data (in this case simply the tuple counts by state) that is still required for incremental view maintenance. □

Both expunction and expiration are useful in a warehouse. In our example, expunction is useful if say our company is no longer tracking Oregon markets because it is pulling out of the region. However, if we still wish to track the average employee counts, but no longer need direct access to the raw market data for Oregon, then expiration is appropriate.

In spite of the importance of expunction and expiration, as far as we know, current warehouse systems do not provide explicit support for them. It is up to the WHA to *manually* manipulate materialized views in order to achieve the desired effect. For example, say we wish to expire market data by timestamp (as opposed to by state). Each day the WHA can upload the file containing new market data, compute the averages for the day, append them to an averages view, and then remove the raw data at the end of the day. This assumes that the averages view is append only. If this is not the case, the WHA must define appropriate auxiliary views, and tell the system how to use them. Expirations based on non-timestamp attributes would be more complex.

In this paper we propose a framework for *system-managed* expunction and expiration. With it, the WHA, or users in some cases, can declare what data is expired or expunged, and the system automatically determines what auxiliary data is needed, and how to maintain views. The WHA can also declare in a very general way what types of base modifications are expected (e.g., append only data, increasing values for some attribute), and the system uses this knowledge to improve the efficiency of expiration and view maintenance. We allow changes in the expiration and modification specifications (with some limitations), and provide algorithms that the system uses to dynamically adjust. (Due to space limitations we do not discuss in detail how the system handles expunction; this is actually simpler than handling expirations.) Furthermore, we provide a number of additional *controls* on each view that specify how the system should react to expirations affecting the view.

We stress that many of the problems addressed in this paper have been individually studied before. For instance, there has been substantial work on incremental view maintenance, view adaptation, defining auxiliary views, using modification constraints to reduce maintenance work, and coping with incomplete information in views (expired data is a type of incomplete information). (All this related work is surveyed in Section 5.) The main contribution of this paper is in *generalizing* and *integrating* the strategies and algorithms earlier developed into a flexible framework. Furthermore, our work extends the prior strategies into a *dynamic* and more realistic setting where the specifications of what is expired and expunged, and of what modifications can occur, can change over time.

The rest of the paper proceeds as follows. Section 2 gives a more detailed overview of our framework and the controls it provides for expiration. That section also defines the notion of consistency among view extensions and the controls. Section 3 gives the algorithms for view initialization and for compensating for changes in controls. In Section 4, we briefly show how these algorithms can be integrated into a warehouse system that runs on top of a conventional RDBMS. Finally, we discuss related work in Section 5.

2 Framework

In this section we give a complete overview of our framework for incremental maintenance and removal of warehouse data. We do this through the example of Figure 5. The circles at the bottom of the figure represent the *base relations* on which the warehouse views are defined. The boxes represent views defined on the relations or on other views. Within each box we list the *controls* for that view: these are "commands" that tell the warehouse how to manage the view. For now, think of the relations as existing at remote sites,

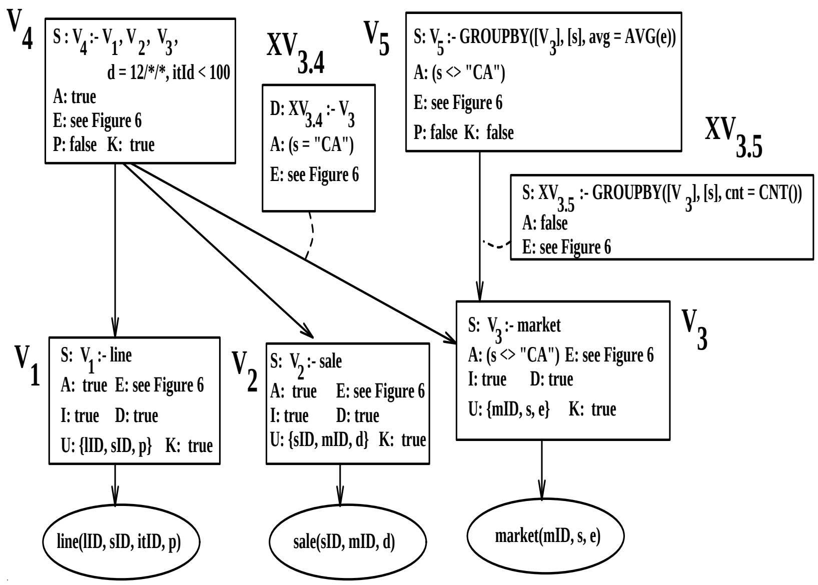

Figure 5: Representation of Views in the Framework

and the views at the warehouse. (This need not be the case in general.) The lines and arrows between boxes and circles are simply for visualizing some of the relationships that will be described in this section.

For our example, we have three base relations that serve as the "inputs" for our warehouse:

- *market(<u>mID</u>, s, e)*: This relation, introduced in Section 1, gives the state s and employee count e of every market. The underlined attribute, *mID*, is the key.
- *sale(<u>sID</u>, mID, d)*: A tuple in this relation represents a sale *sID* at market *mID* on date d. Each sale involves several items, as detailed in the next relation.
- *line(<u>lID</u>, <u>sID</u>, itID, p)*: Each tuple details a line *lID* of a particular sale *sID*, giving the item *itID* sold and the price p it sold at.

We assume that base relations are fully materialized (no expired tuples) but may be expensive to access (for querying or view maintenance) since they may reside in remote sources. To avoid this expense, *base views* in the warehouse are defined over the base relations. Hence, base relations are only accessed when the base views are initialized [1]. In our example, the base views are V_1, V_2 and V_3.

Let us now describe the controls of base view V_3 in Figure 5. $V_3[\mathcal{S}]$ gives the definition of the view using non-recursive Datalog [Ull89] rules. Since handling duplicates is important in evaluating aggregates, we assume the Datalog rules do not eliminate duplicates (as in [GMS93]). We could have used SQL or other bag-based languages, but it leads to more cumbersome expressions later on. For V_3, the specification is

$$V_3(mID, s, e) :- \ market(mID, s, e). \tag{1}$$

(Note that in Figure 5 this specification has been abbreviated to V_3 :− *market*.) This states that every tuple in relation *market* (with attributes *mID*, s, e) should be in V_3, modulo the portion that is expired as described by the *availability constraint* of V_3.

[1] Base views are either *self-maintainable* views ([GJM96]) or views that can be made self-maintainable ([QGMW96]).

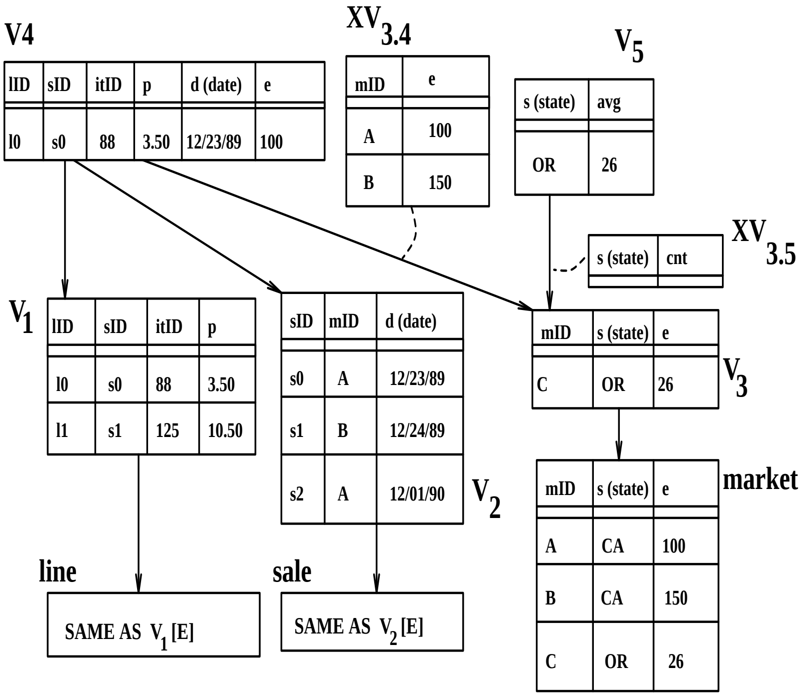

Figure 6: Current Extensions of Views

The availability constraint of V_3, $V_3[\mathcal{A}]$: $s \neq$ "CA", indicates what tuples are *not expired.* That is, if a *market* tuple has a state (s) value of "CA," then that tuple is expired and does not appear in V_3. If a *market* tuple is from a state other than California, then it is materialized in V_3. We use the notation $V_3[\mathcal{E}]$ to refer to the extension of V_3, a bag of materialized tuples. Figure 6 shows $V_3[\mathcal{E}]$ and the extensions for the other views, at a given point in time. (As expected, $V_3[\mathcal{E}]$ is missing the California tuples from relation *market.*) Strictly speaking, $V_3[\mathcal{E}]$ is not a control of V_3, but rather the "result" of the other controls. Given specification $V_j[\mathcal{S}]$ and availability constraint $V_j[\mathcal{A}]$, $V_j[\mathcal{E}]$ is the bag of tuples that results from

$$V_j[\mathcal{S}], V_j[\mathcal{A}]. \tag{2}$$

Notice that expunction is achieved through $V_j[\mathcal{S}]$. For example, if we add condition ($s \neq$ "CA") to $V_3[\mathcal{S}]$ (as opposed to having it in $V_3[\mathcal{A}]$), then California tuples would not be logically part of this view and should not be reflected in views defined over V_3. However, if condition ($s \neq$ "CA") is in $V_3[\mathcal{A}]$ (as in our example), it means that California tuples are expired, i.e., they are not currently materialized in $V_3[\mathcal{E}]$, but they could be reflected in higher level views. For instance, in Figure 6, the one tuple in $V_4[\mathcal{E}]$ contains an employee count e of 100, corresponding to an expired V_3 tuple. We define V_4 below and describe how this V_4 tuple arose.

The next three V_3 controls in Figure 5 are the *modification constraints* $V_3[\mathcal{I}]$, $V_3[\mathcal{D}]$ and $V_3[\mathcal{U}]$. The first two controls describe the insertions and deletions that can occur to V_3 (and hence to *market*). In the current example, both constraints are set to *true* which means any type of modification can occur. However, if for instance the WHA finds out that markets in Oregon have closed, he may set both constraints to ($s \neq$ "OR") to indicate that there will be no more modifications to the Oregon data. In this paper we assume that an update must satisfy both $V_3[\mathcal{I}]$ and $V_3[\mathcal{D}]$, as if it was a delete followed by an insert. Furthermore, control $V_3[\mathcal{U}]$ specifies what attributes may be updated. In the current example, $V_3[\mathcal{U}]$ is $\{mID, s, e\}$, indicating that all of V_3's attributes may be updated. Modification constraints provide semantic information about the underlying application; as we will see later, knowledge of these constraints can help reduce the size of the auxiliary views.

The final V_3 control, $V_3[\mathcal{K}]$, will be described later. Moving to other views in Figure 5, V_1 and V_2 are base views defined on relations *line* and *sale*. Their controls are similar to those of V_3. Notice that since $V_1[\mathcal{A}]$ and $V_2[\mathcal{A}]$ are both *true*, $V_1[\mathcal{E}]$ and $V_2[\mathcal{E}]$ are copies of the extension of the *line* and *sale* relations respectively.

Views V_4 and V_5 are non-base views whose specifications $V_4[\mathcal{S}]$ and $V_5[\mathcal{S}]$ are given by the rules

$$\begin{aligned} V_4(lID, sID, itID, p, d, e) \quad &:- \quad V_1(lID, sID, itID, p), V_2(sID, mID, d), \\ &\quad\;\; V_3(mID, s, e), d = 12/*/*, itID < 100 \end{aligned} \tag{3}$$

$$V_5(s, avg) \quad :- \quad GROUPBY([V_3(mID, s, e)], [s], avg = AVG(e)) \tag{4}$$

View V_4 is a natural join of V_1, V_2, and V_3, where we only want information for December months, and for certain items (with *itID* less than 100). View V_5 is the same aggregate view introduced in Section 1. Both rules are again expressed in non-recursive Datalog. For aggregates, we use the notation defined by [Mum91].

Notice that the availability control of V_4 is set to *true*, meaning that no tuples are expired, despite V_3 having expired some tuples. Clearly, the current $V_4[\mathcal{E}]$ cannot possibly be computed after the tuples in V_3 have been expired unless we read from the original *market* relation. In our framework we assume that for initializing a view we only have access to the extensions of the immediate views. Thus, to initialize V_4 we can only read from $V_1[\mathcal{E}]$, $V_2[\mathcal{E}]$, and $V_3[\mathcal{E}]$ at that time. This implies that to reach the state shown in Figures 5 and 6 the V_3 expiration must have occurred after the initialization of V_4.

Our framework can be extended to allow access to more than the immediate views. However, this means that the query that is used to initialize a view must try to make up for expired data at one level by reading "deeper" views. This is an expensive process, and in many practical cases will not be worthwhile because the deeper views may be at remote sites (like *market*) or will have even more expired data than the immediate views. Because of this, and to simplify our presentation, we restrict access to only the immediate views.

After $V_4[\mathcal{E}]$ is initialized, V_4 needs to be maintained when modifications to V_3 (V_1 and V_2 as well) occur. Since tuples are expired from V_3, the modifications to V_3 may not actually affect $V_3[\mathcal{E}]$! For instance, the deletion from *market* of $\langle A, CA, 100\rangle$, an expired V_3 tuple, does not affect $V_3[\mathcal{E}]$ but will cause the only tuple in $V_4[\mathcal{E}]$ to be deleted. Thus, whether a modification to V_3 affects $V_3[\mathcal{E}]$ or not, it may affect V_4 and must be propagated.

The expiration of the V_3 tuples also creates problems in calculating the modifications to V_4 due to modifications to V_1 and V_2. Suppose $\langle l0, s1, 90, 12.50\rangle$ is inserted into V_1. This tuple joins with the V_2 tuple with *sID* = "s1", which in turn joins with the California V_3 tuple with *mID* = "B". Unfortunately, that V_3 tuple is expired. This implies that to manage V_4 we need an *auxiliary view*, $XV_{3.4}$. This auxiliary view contains expired data from V_3 (the first subscript in $XV_{3.4}$) necessary for maintaining V_4 (the second subscript in $XV_{3.4}$). Auxiliary views were introduced in [QGMW96] and [HZ96].

The contents of $XV_{3.4}$ are defined by $XV_{3.4}[\mathcal{S}]$ and $XV_{3.4}[\mathcal{A}]$. We select $XV_{3.4}[\mathcal{S}]$ to identify those V_3

tuples that would be needed in the *worst case* that $V_3[\mathcal{A}]$ were *false* and $V_4[\mathcal{A}]$ were *true*. In that case, when all of V_3 is expired, we would need all V_3 tuples, in case they may join with modified tuples of V_1 and V_2. That is, $XV_{3.4}[\mathcal{S}]$ is the rule

$$XV_{3.4}(mID, e) :- \ V_3(mID, s, e).$$

Note that we do not need to store the s attribute in $XV_{3.4}$ since it is not used in V_4. The constraint in $XV_{3.4}[\mathcal{A}]$ then tells us which of those tuples are actually needed given the current $V_3[\mathcal{A}]$ and $V_4[\mathcal{A}]$. That is, we only need California tuples, since those are the unavailable ones. Hence, $XV_{3.4}[\mathcal{A}]$ is set to (s = "CA"). One can check that in Figure 6, only California tuples are indeed in $XV_{3.4}[\mathcal{E}]$.

We have not shown the auxiliary views $XV_{1.4}$ and $XV_{2.4}$ in Figures 5 and 6 to avoid clutter. Since both V_1 and V_2 currently have no expired tuples, both $XV_{1.4}$ and $XV_{2.4}$ have empty extensions (i.e., $XV_{1.4}[\mathcal{A}] = XV_{2.4}[\mathcal{A}] = false$).

The extension of $XV_{3.4}$ can be reduced further by considering the modification constraints. To illustrate, suppose for a moment that both $V_1[\mathcal{I}]$ and $V_2[\mathcal{I}]$ are set to *false* (i.e., there are no insertions/updates to V_1 nor V_2). In this case, the tuples in $XV_{3.4}$ will be of no use and $XV_{3.4}[\mathcal{A}]$ can be set to *false*. To see this, notice that $XV_{3.4}$ is only used in propagating V_1, V_2 insertions and updates onto V_4. (View $XV_{3.4}$ is not used for propagating deletions, since with keys deletions can be handled by just joining with V_4.) So, since there are no V_1, V_2 insertions and updates, we can set $XV_{3.4}[\mathcal{A}]$ to *false*. In Section 3 we derive the expressions for $XV_{k.j}[\mathcal{A}]$ which take into consideration availability and modification constraints.

The advantage of these $XV_{k.4}[\mathcal{S}]$ and $XV_{k.4}[\mathcal{A}]$ definitions is that the specification does not have to change as the availability constraint and modification constraints of V_1, V_2, V_3 and V_4 change. For instance, if Oregon tuples are also expired from V_3, then only $XV_{3.4}[\mathcal{A}]$ has to change to cover both California and Oregon tuples. This is the approach we follow in our framework: keep the specification of auxiliary views broad, and use the availability constraints to remove tuples that are not currently needed.

View V_5 is different from V_4 in that California tuples are expired (i.e., $V_5[\mathcal{A}]$ is ($s \neq$ "CA")). This means that V_5 can be initialized even after V_3 tuples have been expired. That is, the expired California V_3 tuples are irrelevant for computing the averages currently materialized in V_5. Strictly speaking we do not need an auxiliary view $XV_{3.5}$ to cope with modifications to V_3 tuples. For instance, if the V_3 tuple $\langle C, OR, 26\rangle$ is updated to $\langle C, OR, 27\rangle$, the new average can be computed easily: simply collect the V_3 tuples with (s = "OR") and compute the average of the employee count (e) attribute. However, in keeping with our general philosophy, we will say that there is an auxiliary view $XV_{3.5}$, except that at this time all of its contents are expired ($XV_{3.5}[\mathcal{A}] = false$). Thus, the specification for $XV_{3.5}$ is set to the rule

$$XV_{3.5}(s, cnt) :- \ GROUPBY([V_3(mID, s, e)], [s], cnt = COUNT()). \tag{5}$$

This defines the contents of $XV_{3.5}$ in the worst case that all of V_3 and none of V_5 were expired. That is, given the counts provided in this specification, we could incrementally maintain V_5 without having to access any V_3 information.

The "purge" Boolean controls, $V_4[\mathcal{P}]$ and $V_5[\mathcal{P}]$ tell the warehouse what to do to compensate for expiration of data these views depend on. For example, if $V_5[\mathcal{P}] = true$, and $V_3[\mathcal{A}]$ is changed to expire both California and Oregon tuples, then the warehouse system would respond by expiring from V_5 the tuple that refers to

Oregon (in addition to the California one already expired). If $V_5[\mathcal{P}] = false$ and the same change occurs, the warehouse would respond by keeping the Oregon tuple in V_5 and by adding the count of Oregon markets to $XV_{3.5}$, i.e., by changing $XV_{3.5}[\mathcal{A}]$ from *false* to (s = "OR"). The value of $V_5[\mathcal{P}]$ depends on the user that defined V_5 (*owner* of V_5). The owner may set $V_5[\mathcal{P}]$ to *true* if he wants the tuples in $V_5[\mathcal{E}]$ to be dependent on $V_3[\mathcal{E}]$. This may be because the owner of V_5 also owns V_3 and uses the availability constraints to express which tuples he is interested in. On the other hand, if he wants $V_5[\mathcal{E}]$ to be independent from $V_3[\mathcal{E}]$, $V_5[\mathcal{P}]$ is set to *false*. Purge controls are not applicable to base views, since the data they depend on never expires.

The last Boolean controls, $V_4[\mathcal{K}]$ and $V_5[\mathcal{K}]$ only apply to views like V_4 and V_5 that have no other views defined on them. The owner of V_4 may set $V_4[\mathcal{K}]$ to *false* if he wants to expire tuples from V_4's auxiliary views based on $V_4[\mathcal{A}]$. This cuts down on the space used by the auxiliary views. Although it is not the focus of the paper, by setting $V_4[\mathcal{K}]$ to *true*, more auxiliary view tuples are saved which can be used in "unexpiring" expired V_4 tuples. As just mentioned, once views are defined on V_4, $V_4[\mathcal{K}]$ must be set to *true* since modifications to V_4 tuples (even expired ones) must be computed to maintain the views defined on V_4.

That concludes the discussion of the controls for V_4 and V_5. Notice that unlike the base views, V_4 and V_5 do not have modification constraints. This is because the modification constraints of V_4 and V_5 depend entirely on the modification constraints of the base views. Thus it would be redundant to associate modification constraints to V_4 and V_5.

So far we have illustrated how the various controls impact what is stored in views, but we have not yet addressed how expiration affects answers to queries. Essentially, we face the same choice we faced when initializing views. To illustrate, consider the query $Q(stID)$:– $V_3(stID, s, e), e > 26$. We could retrieve only data found in V_3, or we could recursively explore the views and relations V_3 is defined on. In the first case, we would retrieve no matching tuples, since the tuples $\langle A, CA, 100\rangle$ and $\langle B, CA, 125\rangle$ are expired. In the last case we would find these two tuples in relation *market*.

For simplicity, in this paper we assume that user queries only read from the views that are directly mentioned in them. The answer may be incomplete, and should be interpreted in light of the respective availability constraints. Thus, when the user is given the answer to the above query, he must be reminded that California tuples are not included. One way to do this would be to associate to each query Q an availability constraint (denoted as $Q[\mathcal{A}]$) that describes the incompleteness of the answer. For this query, $Q[\mathcal{A}]$ is set to ($s \neq$ "CA") which tells the user that the answer shown is actually for the query $Q(stID)$:– $V_3(stID, s, e), e > 26, s \neq$ "CA". In [LGM97] we discuss how our framework can be extended to allow queries to search a given number of levels below the queried views. Also related is [Lev96], which discusses when complete answers can be obtained from an incomplete database.

To summarize, we have presented a framework where warehouse views have the following controls: view definition ($\mathcal{S}$); view extension ($\mathcal{E}$); availability constraint ($\mathcal{A}$); modification constraints ($\mathcal{I}$, $\mathcal{D}$, $\mathcal{U}$); purge ($\mathcal{P}$); and keep ($\mathcal{K}$). Queries only read from the views mentioned in the queries, and for incremental view maintenance of a view V_j we only use data in V_j, in auxiliary views $XV_{k.j}$, and in views that appear in $V_j[\mathcal{S}]$. We still need, however, to discuss one important issue to complete the overview of our framework. This is discussed in the following subsection.

2.1 Consistent State

The controls in a given warehouse are interrelated, and it is important that they be mutually consistent. For instance, the controls in Figure 5 and the extensions in Figure 6 were not chosen at random; on the contrary, they match each other well.

In particular, let the *state* of the warehouse refer to the all the controls of all views and the extensions of all the views. In a consistent state, the following conditions hold:

1. *Data Consistency:* The view extensions hold data that reflect the current base relations, modulo expired tuples. That is, for each view V_j, $V_j[\mathcal{E}]$ is identical to what we would obtain by augmenting $V_j[\mathcal{S}]$ with $V_j[\mathcal{A}]$, expanding $V_j[\mathcal{S}]$ to refer only to base relations, and evaluating it on the current contents of the base relations.

2. *Maintenance Consistency:* For each view V_j defined on views $\{V_k\}$, it is possible to incrementally maintain V_j by accessing only $V_j[\mathcal{E}]$, $\{V_k[\mathcal{E}]\}$, and $\{XV_{k.j}[\mathcal{E}]\}$.

3. *Modification Constraint Consistency:* The modification constraints of the base views describe the modifications to the base views caused by changes to the base relations in the "real world". More formally, if $\triangle V_b$ are the new updated/inserted tuples of some base view V_b, the following rules are guaranteed to produce the same bag of tuples.

$$\begin{aligned} \triangle T_1 \quad &:- \quad \triangle V_b, \\ \triangle T_2 \quad &:- \quad \triangle V_b, V_b[\mathcal{I}] \end{aligned}$$

   Similar rules hold for $\bigtriangledown V_b$ (old updated/deleted tuples) and $V_b[\mathcal{D}]$. As a result, in any rule that $\triangle V_b$ ($\bigtriangledown V_b$) appears in, it can be replaced by $\triangle V_b, V_b[\mathcal{I}]$ ($\bigtriangledown V_b, V_b[\mathcal{D}]$ respectively). Also, only the attributes in $V_b[\mathcal{U}]$ can get updated.

We now illustrate how the warehouse state given in Figures 5 and 6 satisfies these constraints. To check the data consistency of V_4, we rewrite the specification of V_4 by augmenting it with $V_4[\mathcal{A}]$ then recursively replacing the views it accesses by their specifications, until we only have base relations. We obtain the rule

$$V_4(lID, sID, itID, p, d, e) :- \ line, sale, market, d = 12/*/*, itID < 100, true$$

Given the contents of relations *line*, *sale* and *market* shown in Figure 6, the result of the above definition is $\{\langle l0, s0, 88, 3.50, 12/23/89, 100\rangle\}$ which is also $V_4[\mathcal{E}]$. The data consistency of the other views can be confirmed in a similar fashion [2].

Secondly, checking maintenance consistency involves ensuring that views can be properly maintained. For the specific example discussed previously, we have already argued that each view can be properly maintained. In our framework, this type of consistency is ensured by the algorithms we develop in Section 3.

Lastly, we do not need to check modification constraint consistency since we assume that the WHA faithfully describes the modifications to base views based on his knowledge of the "real world". Notice that the WHA does not need to know everything about the "real world" for modification constraint consistency

[2] $V_5[\mathcal{S}]$ is augmented as $V_5 :- \ GROUPBY([V_3, s \neq \text{“}CA\text{”}], [s], avg = AVG(e))$ since selections are done before the grouping.

to hold. For instance, he can set all the $\mathcal{I}$ and $\mathcal{D}$ constraints to *true* initially and make the constraints stronger as he learns more about the modifications to the base views.

3 Maintaining a Consistent State

There are two types of events that can possibly affect the consistency of the warehouse: (1) a new view is defined; (2) a control is changed. In this section, we give the algorithms that handle these two types of events and that guarantee that the warehouse is kept in a consistent state.

When a view V_j is defined, the owner provides the specification $V_j[\mathcal{S}]$ and the boolean values for $V_j[\mathcal{P}]$ and $V_j[\mathcal{K}]$. If V_j is a base view, the values for $V_j[\mathcal{I}]$, $V_j[\mathcal{D}]$ and $V_j[\mathcal{U}]$ are also provided. The view initialization algorithm must then initialize the $V_j[\mathcal{E}]$ and $V_j[\mathcal{A}]$. This portion of the algorithm is discussed in Section 3.1. In addition, the view initialization algorithm must initialize all the auxiliary views $\{XV_{k.j}\}$ necessary to maintain V_j and set their availability constraints $\{XV_{k.j}[\mathcal{A}]\}$ appropriately. This portion of the algorithm is discussed in Section 3.2.

After a view V_j is initialized, most of its controls can be changed and compensating actions are necessary to maintain a consistent state. We give the algorithms that implement these compensating actions in Section 3.3. One type of control change we do not cover in Section 3.3 are changes to $V_j[\mathcal{S}]$ which occur when V_j tuples are expunged. $V_j[\mathcal{S}]$ can be changed only by adding selection conditions or adding semi-joins with other views. The expunged tuples can be found using techniques outlined in [GMR95]. Once found, they are treated as deleted V_j tuples and propagated to the "higher level" views.

Before we discuss the algorithms, we consider the rule $V_8(X) :- V_7(X,Y), X > 7$ and introduce some notation. The variables that appear in the head predicate (i.e., X) are called *distinguished variables*. The body of this rule is composed of an *ordinary predicate* (i.e., $V_7(X,Y)$) and a *built-in predicate* (e.g., $=$, $>$). We call a predicate that represents either new updated/inserted tuples of V ($\triangle V$) or old updated/deleted tuples of V ($\bigtriangledown V$) a *delta predicate*. Also, we will often omit in the rules the variables used in the predicates when they are not needed. For instance, we would just use"V_7" instead of "$V_7(X,Y)$" in the rule above. We call the body of a rule its Right Hand Side (RHS). Moreover, given a set of rules R, we use $RHS(R)$ to denote the disjunction of the RHS's of the rules in R. We assume that all rules are *safe* ([Ull89]).

We need to introduce two more concepts. First, given two constraints $\mathcal{A}_1$ and $\mathcal{A}_2$, $\mathcal{A}_1$ is *stronger* than $\mathcal{A}_2$ iff $\mathcal{A}_1 \Rightarrow \mathcal{A}_2$. Second, we will need the concept of a *complete* V_j which is the bag of tuples that would result from $V_j[\mathcal{S}]$. In rules like the one in the last paragraph, we assume ordinary predicates such as V_7 refer to the complete V_7. $V_7[\mathcal{E}]$ must be used in the rules to refer to the materialized bag of tuples.

We now discuss in detail the various algorithms. Due to space constraints, the complete listing of the algorithms is found in Appendix A. Also, we only discuss the portion of the algorithms for SPJ views.

3.1 Initializing $V_j[\mathcal{E}]$ and $V_j[\mathcal{A}]$

Initializing a view V_j involves computing the extension $V_j[\mathcal{E}]$ and the constraint $V_j[\mathcal{A}]$. $V_j[\mathcal{E}]$ cannot be initialized using $V_j[\mathcal{S}]$ since the rule requires access to the complete underlying views. When a view is initialized, only the extensions of the underlying views are available. Thus the rule to use is

$$V_j[\mathcal{E}] :- RHS(V_j[\mathcal{S}])^{\mathcal{E}}, \tag{6}$$

where $RHS(V_j[\mathcal{S}])^{\mathcal{E}}$ is just $RHS(V_j[\mathcal{S}])$ but with each V_k replaced by $V_k[\mathcal{E}]$. For instance, the rule to initialize $V_4[\mathcal{E}]$ is

$$V_4[\mathcal{E}] \quad :- \quad V_1[\mathcal{E}], V_2[\mathcal{E}], V_3[\mathcal{E}], d = 12/*/*, itID < 100. \tag{7}$$

It is evident from Rule (6) that $V_j[\mathcal{E}]$ may have less tuples than the complete V_j because the extensions of the underlying views themselves may be incomplete. To maintain data consistency, $V_j[\mathcal{A}]$ must be initialized so that the rule $V_j[\mathcal{E}] :- RHS(V_j[\mathcal{S}]), V_j[\mathcal{A}]$ describes the extension of V_j. Since $V_j[\mathcal{E}]$ is given by Rule (6) as well, it is only natural to initialize $V_j[\mathcal{A}]$ based on this rule as illustrated next.

EXAMPLE 3.1 We modify the working example in Section 2 slightly. Assume that when V_4 is defined the availability constraints are as follows: (1) $V_1[\mathcal{A}] : p < 10$; (2) $V_2[\mathcal{A}] : true$; and (3) $V_3[\mathcal{A}] : s \neq$ "CA". Assuming the underlying views are data consistent, we can rewrite Rule (7) by replacing each $V_k[\mathcal{E}]$ by $RHS(V_k[\mathcal{S}]), V_k[\mathcal{A}]$. The rewritten rule in this case is

$$V_4[\mathcal{E}] \quad :- \quad line, p < 10, sale, true, market, s \neq \text{“}CA\text{”}, d = 12/*/*, itID < 100. \tag{8}$$

Since Rule (8) gives the tuples in $V_4[\mathcal{E}]$, its RHS is equivalent to $RHS(V_4[\mathcal{S}]), V_4[\mathcal{A}]$ and it can be used in deriving $V_4[\mathcal{A}]$. In fact, the RHS of Rule (8) is a valid $V_4[\mathcal{A}]$. However, there are obvious redundant predicates. For instance, the predicates ($itID < 100$) and ($d = 12/*/*$) are redundant since both are part of $V_4[\mathcal{S}]$ already. In this example, the predicates resulting from the RHS's of $V_1[\mathcal{S}]$, $V_2[\mathcal{S}]$ and $V_3[\mathcal{S}]$ are also redundant. As a result, $V_4[\mathcal{A}]$ is initialized to $(p < 10) \wedge (s \neq \text{“}CA\text{”})$. □

The example illustrated that finding a valid $V_j[\mathcal{A}]$ is easy. The harder problem is finding a $V_j[\mathcal{A}]$ with relatively few predicates. It is easy to see that this problem can be reduced to the general problem of minimizing conjunctive queries with built-in predicates ([Ull89]). However, since we are dealing with a special case, we use a more efficient algorithm which was illustrated in the example.

The algorithm for initializing $V_j[\mathcal{A}]$ has two steps. First, for each rule r in $V_j[\mathcal{E}] :- RHS(V_j[\mathcal{S}])^{\mathcal{E}}$ that an underlying view $V_k[\mathcal{E}]$ appears in, r is rewritten by substituting $RHS(V_k[\mathcal{S}]), V_k[\mathcal{A}]$ for each occurrence of $V_k[\mathcal{E}]$. (Notice that if $V_k[\mathcal{S}], V_k[\mathcal{A}]$ is more than one rule, r may be rewritten into multiple rules.) The first step is done for each underlying view V_k.

At this point, the disjunction of the RHS's of the rewritten $V_j[\mathcal{S}]$ rules is a valid $V_j[\mathcal{A}]$. The second step of the algorithm eliminates these redundant predicates by considering where the predicates originated from. A predicate can originate from three locations: (1) a $V_j[\mathcal{S}]$ rule (e.g., ($itID < 100$)); (2) $V_k[\mathcal{A}]$ (e.g., ($p < 10$)); or (3) $RHS(V_k[\mathcal{S}])$ (e.g., *market*). Predicates originating from $V_j[\mathcal{S}]$ can be eliminated while predicates from $V_k[\mathcal{A}]$ must be retained. Built-in predicates from $RHS(V_k[\mathcal{S}])$ can be eliminated since they have been applied when $V_k[\mathcal{E}]$ was computed. In most cases, ordinary predicates from $RHS(V_k[\mathcal{S}])$ can be eliminated unless they are required to guarantee safety. For instance, if $V_3[\mathcal{S}]$ was given by (s is projected out) $V_3(mID, e) :- market(mID, s, e)$, the predicate *market* needs to be retained in $V_4[\mathcal{A}]$. Otherwise,

$V_4[\mathcal{E}]$:− $RHS(V_4[\mathcal{S}]), V_4[\mathcal{A}]$ will not even be safe since it has a predicate ($s \neq$ "CA") but s does not appear in an ordinary predicate. The overall algorithm is called *Compute*$\mathcal{A}$ (Figure 14, Appendix A).

It turns out that *Compute*$\mathcal{A}$ can also be used to determine the availability constraint $Q[\mathcal{A}]$ of a query Q. This is because a query is nothing but a view specification whose extension is not materialized.

3.2 Initializing Auxiliary Views

The view maintenance rules that have been developed ([GL95], [GMS93]) assume the complete underlying views are available. An example of such a view maintenance rule is given below.

$$\triangle V_4 :- \triangle V_1, V_2, V_3, d = 12/*/*, itID < 100 \tag{9}$$

In [HZ96] and [QGMW96], they assume that the underlying views cannot be accessed in maintaining a view. Thus, they defined auxiliary views so that a view can be maintained by accessing just the modifications and the auxiliary views. For instance, instead of using Rule (9), the rule

$$\triangle V_4 :- \triangle V_1, XV_{2.4}, XV_{3.4} \tag{10}$$

would be used. Notice that none of the selection conditions need to be applied since these selections are "pushed" into $XV_{k.j}[\mathcal{S}]$ ([HZ96]). In [QGMW96], they assumed that key and referential integrity constraints hold and made the auxiliary views smaller by performing semi-joins in $XV_{k.j}[\mathcal{S}]$.

In this paper, we assume that we have the view maintenance rules that use just the auxiliary views (e.g., Rule (10)). We modify these rules so that they apply in our more general framework. Also, we start with an $XV_{k.j}[\mathcal{S}]$ as given by [HZ96]. We then express referential integrity constraints (and other constraints) used in [QGMW96] as modification constraints and expire the unnecessary tuples by setting $XV_{k.j}[\mathcal{A}]$ appropriately.

Before proceeding, we argue that not all the modifications produced by the conventional view maintenance rules are needed in our framework. For instance, Rule (10) might produce insertions to V_4 that do not satisfy $V_4[\mathcal{A}]$. From the point of view of V_4, these insertions are not needed since they do not affect $V_4[\mathcal{E}]$. However, if there is a view V_i defined on V_4, some of these insertions may be needed to maintain $V_i[\mathcal{E}]$. Since V_i can be defined only after V_4 itself is initialized, only the insertions to V_4 that satisfy $V_4[\mathcal{A}]_{@4}$ are needed. We use $V_k[\mathcal{A}]_{@j}$ to denote the value of $V_k[\mathcal{A}]$ when V_j was initialized. Thus, Rule (10) can be safely be modified to $\triangle V_4 :- \triangle V_1, XV_{2.4}, XV_{3.4}, V_4[\mathcal{A}]_{@4}$. Henceforth, we assume that the view maintenance rules for V_j will have $V_j[\mathcal{A}]_{@j}$ in their RHS's.

3.2.1 Initializing $XV_{k.j}[\mathcal{E}]$ and $XV_{k.j}[\mathcal{A}]$

Translated to our terminology, the algorithms in [HZ96] and [QGMW96] determine $XV_{k.j}[\mathcal{S}]$ assuming $V_k[\mathcal{A}]$ is *false* and $V_j[\mathcal{A}]$ is *true*. Intuitively, some of the tuples that would result from $XV_{k.j}[\mathcal{S}]$ are unnecessary since some can be derived from $V_k[\mathcal{E}]$ or some maintain only expired V_j tuples. We will contend shortly that when V_j is initialized, $XV_{k.j}[\mathcal{E}]$ can be empty and maintenance consistency still holds. Since initializing $XV_{k.j}[\mathcal{E}]$ is trivial, we focus on how $XV_{k.j}[\mathcal{A}]$ is initialized.

The general expression for $XV_{k.j}[\mathcal{A}]$ is $\mathcal{C}_1 \wedge \mathcal{C}_2$. The subexpressions $\mathcal{C}_1$ and $\mathcal{C}_2$ specify the following conditions that must be satisfied by each $XV_{k.j}[\mathcal{E}]$ tuple.

1. $XV_{k.j}[\mathcal{E}]$ tuples are derived from expired V_k tuples. Furthermore, these V_k tuples were expired *after* V_j is initialized. $XV_{k.j}$ tuples derived from unexpired V_k tuples are obviously redundant. Also, $XV_{k.j}$ tuples derived from V_k tuples expired *before* V_j is initialized cannot possibly be used in maintaining V_j. This is because $V_j[\mathcal{E}]$ will not contain any tuples derived from these expired V_k tuples.

2. $XV_{k.j}[\mathcal{E}]$ tuples are used to propagate modifications to the underlying $\{V_k\}$ views onto V_j. An $XV_{k.j}$ tuple that is not used for propagating modifications is useless since $XV_{k.j}$ exists only to maintain V_j.

We now discuss how $\mathcal{C}_1$ and $\mathcal{C}_2$ specify the conditions above.

$\mathcal{C}_1$ is used to enforce the first condition. The general expression for $\mathcal{C}_1$ is

$$V_k[\mathcal{A}]_{@j} \wedge \neg V_k[\mathcal{A}]. \tag{11}$$

Intuitively, including $V_k[\mathcal{A}]_{@j}$ in $\mathcal{C}_1$ prevents tuples derived from V_k tuples expired *before* V_j is initialized from being saved. On the other hand, including $\neg V_k[\mathcal{A}]$ prevents tuples derived from currently unexpired V_k tuples from being saved. Notice that when V_j is initialized, $V_k[\mathcal{A}]_{@j}$ is equal to $V_k[\mathcal{A}]$. Thus initially, $XV_{k.j}[\mathcal{A}]$ is *false* and $XV_{k.j}[\mathcal{E}]$ is empty. This is reasonable since $V_k[\mathcal{E}]$ is sufficient in maintaining V_j initially. The next example illustrates the use of Expression (11).

EXAMPLE 3.2 We revisit the working example in Section 2 and focus on $XV_{3.4}$. Since $V_3[\mathcal{A}]$ was *true* when V_4 was initialized, $V_3[\mathcal{A}]_{@4}$ is *true*. The current state of the warehouse has $V_3[\mathcal{A}]$ set to $(s \neq \text{“}CA\text{”})$. Using Expression (11), $\mathcal{C}_1$ is computed to be $(s = \text{“}CA\text{”})$. Assuming $\mathcal{C}_2$ is *true*, $XV_{3.4}[\mathcal{A}]$ is computed to be $(s = \text{“}CA\text{”})$ as given in Section 2. □

$\mathcal{C}_1$ reduces $XV_{k.j}[\mathcal{E}]$ by expiring tuples that are derived from unexpired V_k tuples. $\mathcal{C}_2$ reduces $XV_{k.j}[\mathcal{E}]$ even further by expiring tuples that are not used in propagating modifications to V_j. The $\mathcal{C}_2$ for $XV_{k.j}[\mathcal{A}]$ is derived by considering the view maintenance rules for V_j that use only extensions. We call these view maintenance rules $VMR^{\mathcal{E}}$ to emphasize that only extensions are used. Because of the incompleteness of the underlying views, the conventional view maintenance rules cannot be used. We now illustrate how $VMR^{\mathcal{E}}$ is derived. The full algorithm called *DeriveVMR*$^{\mathcal{E}}$ is in Figure 16, Appendix A.

EXAMPLE 3.3 Consider a hypothetical view V_8 whose $V_8[\mathcal{S}]$ is $V_8 :- V_6, V_7, B_6$ where B_6 is a built-in predicate that only uses V_6's variables. Due to space constraints, we only consider the maintenance rules for propagating insertions onto V_8. We assume that the insertions are propagated separately. The conventional view maintenance rules that use the auxiliary views are as follows.

$$\triangle V_8 \quad :- \quad \triangle V_6, XV_{7.8}, V_8[\mathcal{A}]_{@8} \tag{12}$$

$$\triangle V_8 \quad :- \quad XV_{6.8}, \triangle V_7, V_8[\mathcal{A}]_{@8} \tag{13}$$

However, since these rules use the complete $XV_{6.8}$ and $XV_{7.8}$, they cannot be used in calculating $\triangle V_8$. Our goal then is to find a set of rules that only uses $\{V_k[\mathcal{E}]\}$, $\{XV_{k.j}[\mathcal{E}]\}$ and $\{\triangle V_k\}$ but results in the same bag of tuples as the one that results from Rules (12) and (13).

Since the complete auxiliary views such as $XV_{6.8}$ is not available, we now attempt to “construct” $XV_{6.8}$ from the available extensions. The complete $XV_{6.8}$ can be divided into three parts: (1) tuples that satisfy

$V_6[\mathcal{A}]_{@8} \wedge \neg V_6[\mathcal{A}]$; (2) tuples that satisfy $\neg V_6[\mathcal{A}]_{@8}$; and (3) tuples that satisfy $V_6[\mathcal{A}]$. It then follows that the following rules derive the complete $XV_{6.8}$.

$$XV_{6.8} \quad :- \quad RHS(XV_{6.8}[\mathcal{S}]), \neg V_6[\mathcal{A}]_{@8}$$

$$XV_{6.8} \quad :- \quad RHS(XV_{6.8}[\mathcal{S}]), V_6[\mathcal{A}]$$

$$XV_{6.8} \quad :- \quad RHS(XV_{6.8}[\mathcal{S}]), V_6[\mathcal{A}]_{@8}, \neg V_6[\mathcal{A}]$$

Rule (13) can be rewritten by substituting for $XV_{6.8}$ the RHS's of the rules above (with proper renaming of variables). This rewriting results in the following rules.

$$\triangle V_8 \quad :- \quad RHS(XV_{6.8}[\mathcal{S}]), \neg V_6[\mathcal{A}]_{@8}, \triangle V_7, V_8[\mathcal{A}]_{@8} \tag{14}$$

$$\triangle V_8 \quad :- \quad RHS(XV_{6.8}[\mathcal{S}]), V_6[\mathcal{A}], \triangle V_7, V_8[\mathcal{A}]_{@8} \tag{15}$$

$$\triangle V_8 \quad :- \quad RHS(XV_{6.8}[\mathcal{S}]), V_6[\mathcal{A}]_{@8}, \neg V_6[\mathcal{A}], \triangle V_7, V_8[\mathcal{A}]_{@8} \tag{16}$$

Focusing on Rule (14), its RHS evaluates to *false* because $\neg V_6[\mathcal{A}]_{@8}$ is certainly inconsistent with $V_8[\mathcal{A}]_{@8}$ (see Section 3.1). Thus, this rule is not needed.

Focusing on Rule (15), we argue that $RHS(V_6[\mathcal{S}]), B_6$ can substituted for $RHS(XV_{6.8}[\mathcal{S}])$. This is because the $XV_{6.8}[\mathcal{S}]$ as given by [HZ96] is an SP view over V_6 (Section 3.2). Thus, for every tuple $t_{6.8}$ that results from $RHS(XV_{6.8}[\mathcal{S}])$ (i.e., the V_6 tuples that pass the selection conditions), there must a tuple t_6, with possibly a superset of $t_{6.8}$'s attributes, that results from $RHS(V_6[\mathcal{S}])$ and derives $t_{6.8}$. Moreover, substituting $RHS(V_6[\mathcal{S}]), B_6$ for $RHS(XV_{6.8}[\mathcal{S}])$ does not result in more insertions to V_8 because any selection conditions in $XV_{6.8}[\mathcal{S}]$ are also applied in the rewritten rule as well. As a result, Rule (15) is equivalent to

$$\triangle V_8 \quad :- \quad V_6[\mathcal{E}], B_6, \triangle V_7, V_8[\mathcal{A}]_{@8}. \tag{17}$$

Focusing on Rule (16), since $XV_{6.8}[\mathcal{A}]$ is $\mathcal{C}_1 \wedge \mathcal{C}_2$ and $\mathcal{C}_1$ is $V_6[\mathcal{A}]_{@8} \wedge \neg V_6[\mathcal{A}]$, we argue that this rule can be rewritten as

$$\triangle V_8 \quad :- \quad XV_{6.8}[\mathcal{E}], \triangle V_7, V_8[\mathcal{A}]_{@8} \tag{18}$$

given that $\mathcal{C}_2$ is set properly. That is, this rewriting is only valid if $\mathcal{C}_2$ is set such that Rule (16) without $\mathcal{C}_2$ is equivalent to Rule (18) that implicitly has $\mathcal{C}_2$ as part of $XV_{6.8}[\mathcal{E}]$ (which includes $XV_{6.8}[\mathcal{A}]$). For instance, if $\mathcal{C}_2$ was *true*, this holds. Our strategy then is to start with $\mathcal{C}_2$ set to *true* so that Rule (16) can be rewritten to Rule (18). We then make $\mathcal{C}_2$ stronger if we are sure that such rewritings still hold.

In summary, we have argued that Rules (17) and (18) are equivalent to Rule (13). A similar argument can be made that the rules

$$\triangle V_8 \quad :- \quad \triangle V_6, XV_{7.8}[\mathcal{E}], V_8[\mathcal{A}]_{@8} \tag{19}$$

$$\triangle V_8 \quad :- \quad \triangle V_6, V_7[\mathcal{E}], V_8[\mathcal{A}]_{@8} \tag{20}$$

are equivalent to Rule (12). Thus, Rules (17) through (20) are equivalent to Rules (12) and (13). The difference between these two sets of rules is that Rules (17) through (20) do not rely on the availability of the complete auxiliary views. Hence, Rules (17) through (20) can be used to maintain V_8.

Notice that obtaining the $VMR^{\mathcal{E}}$ of V_8 only requires a simple rewriting of the original rules. That is, for each rule like Rule (13), we substitute for each $XV_{k.8}$ either $V_k[\mathcal{E}]$ or $XV_{k.8}[\mathcal{E}], B_k$ (where B_k are the built-in predicates in $XV_{k.8}[\mathcal{S}]$). All possible combinations of substitutions are made. □

In the previous example, we assumed that $\mathcal{C}_2$ was *true* when we proved that Rules (16) and (18) were equivalent. We now endeavor to make $\mathcal{C}_2$ stronger. In order to explain how $\mathcal{C}_2$ is made stronger, Rules (16) and (18) are written in their general form below. (P_i denotes a predicate that could be an ordinary, built-in or delta predicate. Also, we have replaced $XV_{k.j}[\mathcal{E}]$ with $RHS(XV_{k.j}[\mathcal{S}]), \mathcal{C}_1, \mathcal{C}_2$ in Rule (22).)

$$\triangle V_j \quad :- \quad RHS(XV_{k.j}[\mathcal{S}]), \mathcal{C}_1, P_1, \ldots, P_n \tag{21}$$

$$\triangle V_j \quad :- \quad RHS(XV_{k.j}[\mathcal{S}]), \mathcal{C}_1, \mathcal{C}_2, P_1, \ldots, P_n. \tag{22}$$

For the two rules to be equivalent, there must exist containment mappings from Rule (21) to Rule (22) and vice versa ([CM77]). The existence of a containment mapping from Rule (21) to Rule (22) is obvious. On the other hand, the existence of a containment mapping from Rule (22) to Rule (21) just requires $\mathcal{C}_2$ to be *true* or to be mapped to some of the predicates in $\{P_i\}$. Thus, the idea in making $\mathcal{C}_2$ stronger than *true* is to choose a subset of $\{P_i\}$ from Rule (21) (or equivalently Rule (22)) that can be "added" to $\mathcal{C}_2$. In the next section, we show that only some of the predicates in $\{P_i\}$ can be added to $\mathcal{C}_2$.

3.2.2 Modifications Constraints and $XV_{k.j}[\mathcal{A}]$

When modification constraint consistency holds, modification constraints are guaranteed to be satisfied by modifications to base views. Because of this guarantee, for any base view V, $\triangle V$ ($\bigtriangledown V$) can always be substituted by $\triangle V, V[\mathcal{I}]$ ($\bigtriangledown V, V[\mathcal{D}]$ respectively) in any rule that $\triangle V$ ($\bigtriangledown V$ respectively) appears in. This substitution helps make $\mathcal{C}_2$ stronger as we will illustrate. After the example, we discuss how to handle the delta predicates of non-base views.

EXAMPLE 3.4 We revisit the working example in Section 2. Due to space constraints, we only consider making $\mathcal{C}_2$ of $XV_{2.4}[\mathcal{A}]$ stronger using $\mathcal{I}$ constraints. Recall that $XV_{2.4}$ is an auxiliary view of V_4 that is defined on the view V_2. Since we are initializing $\mathcal{C}_2$ of $XV_{2.4}[\mathcal{A}]$, only the rules in $VMR^{\mathcal{E}}$ of V_4 that use $XV_{2.4}[\mathcal{E}]$ are needed. These rules are listed below.

$$\triangle V_4 \quad :- \quad \triangle V_1, XV_{2.4}[\mathcal{E}], XV_{3.4}[\mathcal{E}] \tag{23}$$

$$\triangle V_4 \quad :- \quad \triangle V_1, XV_{2.4}[\mathcal{E}], V_3[\mathcal{E}] \tag{24}$$

$$\triangle V_4 \quad :- \quad XV_{1.4}[\mathcal{E}], XV_{2.4}[\mathcal{E}], \triangle V_3 \tag{25}$$

$$\triangle V_4 \quad :- \quad V_1[\mathcal{E}], itID < 100, XV_{2.4}[\mathcal{E}], \triangle V_3 \tag{26}$$

Without considering modification constraints, none of the predicates can be added to $\mathcal{C}_2$. The predicate ($itID < 100$) cannot be added because $XV_{2.4}$ does not use the $itID$ variable. Delta predicates, such as $\triangle V_6$, cannot be added because they represent the *current* modifications. It is unreasonable to expire based on current modifications since the expired tuples may join with *future* modifications. Also, we cannot add ordinary predicates such as $V_1[\mathcal{E}]$. This is because, a tuple inserted into V_4 may be produced in three ways from the point of view of V_1. The tuple can be derived from a tuple in: (1) $\triangle V_1$; (2) $V_1[\mathcal{E}]$; or (3) $XV_{1.4}[\mathcal{E}]$. Unfortunately, we cannot set $\mathcal{C}_2$ to $\triangle V_1 \vee V_1[\mathcal{E}] \vee XV_{1.4}[\mathcal{E}]$ since we are adding a delta predicate in $\mathcal{C}_2$. Neither can we set $\mathcal{C}_2$ to $V_1[\mathcal{E}] \vee XV_{1.4}[\mathcal{E}]$ because some expired $XV_{2.4}$ tuple may join with some tuple in $\triangle V_1$. In short, neither $XV_{k.j}[\mathcal{E}]$ nor $V_k[\mathcal{E}]$ can be added as long as $\triangle V_k$ is in some of the view maintenance rule.

Now, let us consider modification constraints and assume that $V_3[\mathcal{I}]$ is *false* (i.e., there are no insertions or updates to V_3). When $\triangle V_3, V_3[\mathcal{I}]$ is substituted for $\triangle V_3$, the RHS's of Rules (25) and (26) evaluates to *false* which guarantees that no insertions/updates are produced by the rules. Hence, when the two remaining rules are examined, $\mathcal{C}_2$ is set to $XV_{3.4}[\mathcal{E}] \vee V_3[\mathcal{E}]$. That is, tuples that do not join with any tuples in $XV_{3.4}[\mathcal{E}]$ nor in $V_3[\mathcal{E}]$ are also expired. Notice that when $V_3[\mathcal{A}]$ is *false* (i.e., $V_3[\mathcal{E}]$ is empty), the constraint specifies a semi-join with $XV_{3.4}$ which is included in the $XV_{k.j}[\mathcal{S}]$ proposed by [QGMW96]. In Appendix B, we show how more complex modification constraints that describe referential integrity constraints, "append-only" insertions and *protected* updates help in making $\mathcal{C}_2$ stronger.

Modification constraints do not have to prove that certain view maintenance rules do not produce any modifications in order to help make $\mathcal{C}_2$ stronger. For instance, assume that $V_3[\mathcal{I}]$ is $(mID > \text{"}C\text{"})$ and $V_1[\mathcal{I}]$ is $(sID > \text{"}s2\text{"})$ say. Although with these constraints, it cannot be deduced that certain rules do not produce any insertions, both $(mID > \text{"}C\text{"})$ and $(sID > \text{"}s2\text{"})$ can be added to $\mathcal{C}_2$. When Rules (23) through (26) are examined (with each $\triangle V$ replaced by $\triangle V, V[\mathcal{I}]$), $\mathcal{C}_2$ is set to $(mID > \text{"}C\text{"}) \vee (sID > \text{"}s2\text{"})$. □

The previous example replaced each $\triangle V$ with $\triangle V, V[\mathcal{I}]$ where V was a base view. When V is not a base view, we need to compute an "effective" $V[\mathcal{I}]$ and $V[\mathcal{D}]$ based on the modifications constraints of the base views. The algorithm (Figure 15, Appendix A) is simple and similar to the one that computes $V_j[\mathcal{A}]$.

3.2.3 $V_j[\mathcal{A}]$ and $XV_{k.j}[\mathcal{A}]$

In addition to modification constraints, $V_j[\mathcal{A}]$ can also be used to make $\mathcal{C}_2$ stronger. To take into account $V_j[\mathcal{A}]$ when $XV_{k.j}[\mathcal{A}]$ is initialized, the rules in $VMR^{\mathcal{E}}$ of V_j must be augmented with $V_j[\mathcal{A}]$. Augmenting the rules with $V_j[\mathcal{A}]$ has two consequences. First, more modifications are filtered since $V_j[\mathcal{A}]$ is at least as strong as $V_j[\mathcal{A}]_{@j}$. However, recall that filtering the modifications that do not satisfy $V_j[\mathcal{A}]$ is only feasible when there are no views defined on V_j. Second, once $V_j[\mathcal{A}]$ becomes stronger, it provides more predicates that can be added to $\mathcal{C}_2$ for some $XV_{k.j}[\mathcal{A}]$. Since $\mathcal{C}_2$ can only be made stronger when $V_j[\mathcal{A}]$ is made stronger, we revisit how $XV_{k.j}[\mathcal{A}]$ is affected by $V_j[\mathcal{A}]$ when control changes are discussed (Section 3.3.1).

Whether the rules in $VMR^{\mathcal{E}}$ are augmented or not is up to the owner of V_j to decide. When $V_j[\mathcal{K}]$ is *false*, the owner indicates that he wants to expire $XV_{k.j}$ tuples based on $V_j[\mathcal{A}]$ (i.e., the second consequence is desired). Recall that the owner is only allowed to set $V_j[\mathcal{K}]$ to *false* if there are no views defined on V_j (i.e., the first consequence is feasible). Thus, when $V_j[\mathcal{K}]$ is *false*, the rules are augmented with $V_j[\mathcal{A}]$.

3.2.4 Overall Algorithm

In summary, $XV_{k.j}[\mathcal{A}]$ is just $\mathcal{C}_1 \wedge \mathcal{C}_2$ where $\mathcal{C}_1$ is $V_k[\mathcal{A}]_{@j} \wedge \neg V_k[\mathcal{A}]$. On the other hand, $\mathcal{C}_2$ needs to be computed by first producing the $VMR^{\mathcal{E}}$ of V_j. Only the rules that use $XV_{k.j}[\mathcal{E}]$ are needed. These rules are then rewritten by replacing each $\triangle V$ ($\triangledown V$) by $\triangle V, V[\mathcal{I}]$ ($\triangledown V, V[\mathcal{D}]$ respectively) which may require computing an "effective" $V[\mathcal{I}]$ ($V[\mathcal{D}]$). The rules that are guaranteed not to produce any modifications are eliminated. Also, the rules are augmented with $V_j[\mathcal{A}]$ if possible. Lastly, a subset of the predicates in the rules are chosen to be added to $\mathcal{C}_2$. The overall algorithm *Compute*$\mathcal{C}_2$ is shown in Figure 16 (Appendix A).

3.3 Compensating for Control Changes

In this section, we discuss the necessary compensating actions to maintain a consistent state when controls are changed. The possible changes to controls and the corresponding algorithms that implement the compen-

Control change	Compensating Algorithm or Effect
$V_j[\mathcal{A}]$	*Compensate$\mathcal{A}$*
$V_j[\mathcal{I}]$, $V_j[\mathcal{D}]$, $V_j[\mathcal{U}]$	*Compensate$\mathcal{M}$*
$V_j[\mathcal{P}]$	affects *Compensate$\mathcal{A}$*
$V_j[\mathcal{K}]$	affects *Compensate$\mathcal{A}$* and *Compute$\mathcal{C}_2$*

Figure 7: Possible Control Changes and its Effects

sating actions are shown in Figure 7. We have omitted listing the control $XV_{k.j}[\mathcal{A}]$ since this control cannot be changed by the owner of V_j. $XV_{k.j}[\mathcal{A}]$ can only be changed within *Compensate$\mathcal{A}$* and *Compensate$\mathcal{M}$*. As we will show, the changes to $XV_{k.j}[\mathcal{A}]$ are compensated for appropriately. Also notice that changes to $V_j[\mathcal{P}]$ and $V_j[\mathcal{K}]$ only affect *Compensate$\mathcal{A}$* and *Compute$\mathcal{C}_2$* and do not require other compensating actions. Hence, we focus on the first two types of control change. In the rest of the section, we assume that $\{V_i\}$ are the views defined on V_j and $\{V_k\}$ are the views upon which V_j is defined on.

3.3.1 Changing $V_j[\mathcal{A}]$

Before delving into how changes to $V_j[\mathcal{A}]$ are compensated for, we first limit how $V_j[\mathcal{A}]$ can be changed. We assume that $V_j[\mathcal{A}]$ is expressed as $V_j[\mathcal{A}_s] \wedge V_j[\mathcal{A}_o]$. When we discussed how $V_j[\mathcal{A}]$ was initialized by the system (using *Compute$\mathcal{A}$*) in Section 3.1, we actually only referred to $V_j[\mathcal{A}_s]$ since $V_j[\mathcal{A}_o]$ is just initially *true*. $V_j[\mathcal{A}_s]$ can only be changed when some $V_k[\mathcal{A}]$ is changed and $V_j[\mathcal{P}]$ is *true* (to be discussed). On the other hand, $V_j[\mathcal{A}_o]$ can be changed by the owner in a way that conforms to the next two assumptions. Since our focus is on expiration of data, we assume $V_j[\mathcal{A}]$ can only be made stronger. Lastly, we assume that when the owner includes ordinary predicates in $V_j[\mathcal{A}_o]$, the predicates refer to extensions. Thus, if some $V_h[\mathcal{E}]$ is in $V_j[\mathcal{A}_o]$, $V_j[\mathcal{A}_o]$ is changed whenever $V_h[\mathcal{E}]$ changes (due to modifications to V_h or changes to $V_h[\mathcal{A}]$). We now discuss how such changes to $V_j[\mathcal{A}]$ are compensated for. In the discussion, $V_j[\mathcal{A}]$ has been changed to the new availability constraint $V_j[\mathcal{A}]'$. The new extension $V_j[\mathcal{E}]'$ needs to be computed given that $V_j[\mathcal{A}]$, $V_j[\mathcal{A}]'$ and the old extension $V_j[\mathcal{E}]$ are available.

After $V_j[\mathcal{A}]$ is changed, a compensating action that expires a subset of the V_j tuples (denoted as $\triangledown V_j$) is required to maintain data consistency. There are actually other compensating actions required. All the compensating actions are implemented by the *Compensate$\mathcal{A}$* algorithm (Figure 12, Appendix A) which is composed of four main parts. We discuss each part in turn but we ignore Part (4) since it is easy.

1. **Part (1):** $\triangledown V_j$ is determined.
2. **Part (2):** For each view V_i defined on V_j, either the V_i tuples derived from $\triangledown V_j$ are also expired when $V_i[\mathcal{P}]$ is *true* or parts of $\triangledown V_j$ are saved in the auxiliary view $XV_{j.i}$ when $V_i[\mathcal{P}]$ is *false*.
3. **Part (3):** If $V_j[\mathcal{K}]$ is set to *false*, expire tuples from the auxiliary views of V_j based on $V_j[\mathcal{A}]$.
4. **Part (4):** $\triangledown V_j$ is removed from the materialized view on disk.

Part (1) of *Compensate$\mathcal{A}$*: The rules that compute $\triangledown V_j$ are derived from the basic definition of $\triangledown V_j$. That is, given the current extension $V_j[\mathcal{E}]$ and *if* we know the new extension $V_j[\mathcal{E}]'$, $\triangledown V_j$ is simply the tuples that were in $V_j[\mathcal{E}]$ but not in $V_j[\mathcal{E}]'$ as given by the following rule.

$$\triangledown V_j :- V_j[\mathcal{E}], \neg V_j[\mathcal{E}]' \tag{27}$$

Since $RHS(V_j[\mathcal{S}]), V_j[\mathcal{A}_s]', V_j[\mathcal{A}_o]'$ describes $V_j[\mathcal{E}]'$ and $RHS(V_j[\mathcal{S}]), V_j[\mathcal{A}_s], V_j[\mathcal{A}_o]$ describes $V_j[\mathcal{E}]$, it is not hard to see that Rule (27) can be expanded to

$$\triangledown V_j \quad :- \quad RHS(V_j[\mathcal{S}]), V_j[\mathcal{A}_s], V_j[\mathcal{A}_o], \neg V_j[\mathcal{A}_o]' \tag{28}$$

$$\triangledown V_j \quad :- \quad RHS(V_j[\mathcal{S}]), V_j[\mathcal{A}_s], V_j[\mathcal{A}_o], \neg V_j[\mathcal{A}_s]' \tag{29}$$

For now, we assume that $V_j[\mathcal{A}_o]$ changed but $V_j[\mathcal{A}_s]$ has not. We show later that similar rules are used when $V_j[\mathcal{A}_s]$ is changed instead. With this assumption, the RHS of Rule (29) evaluates to *false*. Hence, Rule (28) is equivalent to Rule (27). Unfortunately, neither of these rules can be used in practice since they require access to the complete views (Rule (28)) or to an extension that is yet to be computed (Rule (27)). Our strategy then is to find rules that access current extensions that are equivalent to either of the two rules. We now show how this is done for three different cases that are described below.

1. **Case (1):** The variables used in $V_j[\mathcal{A}_o]'$ are distinguished variables in $V_j[\mathcal{S}]$ or distinguished variables of some ordinary predicate in $V_j[\mathcal{A}_o]'$.
2. **Case (2):** $\{V_k[\mathcal{A}]\}$ have not changed since V_j was defined.
3. **Case (3):** $V_j[\mathcal{P}]$ is set to *true*.

In Case (1), since $V_j[\mathcal{A}_o]'$ does not use any non-distinguished variables, Rule (28) can be simplified to

$$\triangledown V_j :- V_j[\mathcal{E}], \neg V_j[\mathcal{A}_o]'. \tag{30}$$

(If $V_j[\mathcal{A}_o]$ used non-distinguished variables, then the above rule becomes unsafe.) Notice that this rule can be run because it only accesses the current extension of V_j. We now illustrate how Rule (30) is used.

EXAMPLE 3.5 We revisit the working example in Section 2. Recall that in that example, $V_3[\mathcal{A}_o]$ was changed from *true* to ($s \neq$ "CA"). Notice that this scenario falls under Case (1) since only distinguished variables are used in $V_3[\mathcal{A}_o]'$. As a result, the rule (based on Rule (30)) $\triangledown V_3(mID, s, e) :- V_3[\mathcal{E}](mID, s, e), s =$ "CA" is used to find $\triangledown V_3$. In this specific example, the rule results in the tuples $\{\langle A, CA, 100\rangle, \langle B, CA, 150\rangle\}$ which were the tuples expired in the example in Section 2.

□

Case (1) constrains the owner of V_j to formulate $V_j[\mathcal{A}_o]$ using only the distinguished variables of views. (He can use the distinguished variables of a view V_h other than V_j as long as he includes $V_h[\mathcal{E}]$ in $V_j[\mathcal{A}_o]$.) Since, for any view V, we can think of the distinguished variables in $V[\mathcal{S}]$ as the schema of V, we expect Case (1) to be the common case. In fact, the only other variables that can be used in $V_j[\mathcal{A}]$ without jeopardizing safety are the non-distinguished variables in $V_j[\mathcal{S}]$.

Although Case (1) is the common case, the owner of V_j may want to use the non-distinguished variables in $V_j[\mathcal{S}]$. The owner is given this leeway as long as the scenario falls under Case (2) or Case (3). Under either of these cases, it is guaranteed that

$$\triangledown V_j \quad :- \quad RHS(V_j[\mathcal{S}])^{\mathcal{E}}, V_j[\mathcal{A}_o], \neg V_j[\mathcal{A}_o]' \tag{31}$$

can be used in determining $\triangledown V_j$. (Recall that $RHS(V_j[\mathcal{S}])^{\mathcal{E}}$ is $RHS(V_j[\mathcal{S}])$ with each V_k replaced by $V_k[\mathcal{E}]$.) In [LGM97], Rule (31) is proven to be equivalent to Rule (28) under Cases (2) and (3) using containment mappings. The difference between these two rules is that Rule (31) can be used in practice since it only uses available extensions.

Apart from Cases (1) through (3), the only remaining case is when the non-distinguished variables from $V_j[\mathcal{S}]$ are used but neither the conditions of Case (2) nor Case (3) are satisfied. In this case, it is still possible to find $\triangledown V_j$ if the auxiliary views $\{XV_{k.j}\}$ are used. However there is no guarantee of success (see [LGM97]). In this paper, we do not handle this case and we disallow any changes to $V_j[\mathcal{A}_o]$ that fall under this case.

We have shown how Cases (1) through (3) are handled when $V_j[\mathcal{A}_o]$ is changed. If $V_j[\mathcal{A}_s]$ is changed instead, we can perform the same case analysis and we do not show it here. (Of course, when $V_j[\mathcal{A}_s]$ is changed, Case (1) constrains $V_j[\mathcal{A}_s]'$ not $V_j[\mathcal{A}_o]'$ to use just distinguished variables.) Case (1) uses Rule (32). Cases (2) and (3) use Rule (33). Notice that the only difference between these rules and the ones used when $V_j[\mathcal{A}_o]$ was changed is that $V_j[\mathcal{A}_s]'$ is used instead of $V_j[\mathcal{A}_o]'$.

$$\triangledown V_j \quad :- \quad V_j[\mathcal{E}], \neg V_j[\mathcal{A}_s]' \tag{32}$$

$$\triangledown V_j \quad :- \quad RHS(V_j[\mathcal{S}])^{\mathcal{E}}, V_j[\mathcal{A}_o], \neg V_j[\mathcal{A}_s]' \tag{33}$$

The overall algorithm for finding $\triangledown V_j$ is called *GetTuplesToExpire* (Figure 11, Appendix A). It just determines whether $V_j[\mathcal{A}_o]$ or $V_j[\mathcal{A}_s]$ changed. It also determines which of the three cases the specific change falls under in. Once these are determined, the appropriate rules are used to find $\triangledown V_j$.

Part (2) of ***Compensate$\mathcal{A}$*****:** Expiring $\triangledown V_j$ may affect the views $\{V_i\}$ defined on V_j in two ways.

1. When $V_i[\mathcal{P}]$ is *true*, the V_i tuples derived from $\triangledown V_j$ are also expired.
2. When $V_i[\mathcal{P}]$ is *false*, parts of $\triangledown V_j$ are saved in $XV_{j.i}$ so that V_i can still be maintained.

Let us consider the first case where $V_i[\mathcal{P}]$ is set to *true*. One way of computing the effect of expiring $\triangledown V_k$ on V_i is to just use the rules in $VMR^{\mathcal{E}}$ of V_i that respond to deletions to V_k (i.e., $\triangledown V_k$ are treated as deletions). We adopt a different method in order to "reuse" the algorithms in Part (1) of *Compensate$\mathcal{A}$*. We simply recompute $V_i[\mathcal{A}_s]$ using *Compute$\mathcal{A}$* (Section 3.1). Once $V_i[\mathcal{A}_s]$ is changed, *Compensate$\mathcal{A}$*(V_i) is invoked to perform the necessary compensating actions for this change. One of the actions will be to expire the necessary tuples from V_i. We now illustrate this in the next example.

EXAMPLE 3.6 In Example 3.5, we showed how $\triangledown V_3$ was determined when $V_3[\mathcal{A}]$ was changed from *true* to ($s \neq$ "CA"). Now assuming $V_4[\mathcal{P}]$ is set to *true*, $V_4[\mathcal{A}_s]$ is changed accordingly using *Compute$\mathcal{A}$*. Since $V_1[\mathcal{A}]$ and $V_2[\mathcal{A}]$ are both *true*, *Compute$\mathcal{A}$* computes $V_4[\mathcal{A}_s]$ to be ($s \neq$ "CA").

Since $V_4[\mathcal{A}]$ was changed, another set of compensating actions is initiated. One of the compensating actions is that *GetTuplesToExpire* finds the tuples to expire from V_4. In this case, the following rule based on Rule (33) is used to find $\triangledown V_4$.

$$\triangledown V_4 \quad :- \quad V_1[\mathcal{E}], V_2[\mathcal{E}], V_3[\mathcal{E}], d = 12/*/*, itID > 100, true, s = \text{"}CA\text{"}$$

Notice that $V_3[\mathcal{E}]$ still has the California tuples since $\triangledown V_3$ has not been expired at this point. Once $\triangledown V_4$ is expired, data consistency is maintained since $RHS(V_4[\mathcal{S}]), V_4[\mathcal{A}]$ yields $V_4[\mathcal{E}]$. □

On the other hand, if $V_i[\mathcal{P}]$ is *false*, the expiration of $\triangledown V_j$ must be made transparent to V_i. In order to maintain V_i despite the expiration of $\triangledown V_j$, parts of $\triangledown V_j$ must be saved in the auxiliary view $XV_{j.i}$. In [LGM97], we prove, using containment mappings, that when $XV_{j.i}$ is an auxiliary view of an SPJ V_i view, the rule to determine what to insert into $XV_{j.i}$ is almost identical to $XV_{j.i}[\mathcal{S}]$ but with V_j replaced by $\triangledown V_j$. Since data consistency needs to be maintained, $XV_{j.i}[\mathcal{A}]$ needs to be recomputed. Only $\mathcal{C}_1$ is recomputed because the change to $V_j[\mathcal{A}]$ does not affect $\mathcal{C}_2$ of $XV_{j.i}[\mathcal{A}]$. We now illustrate these compensating actions.

EXAMPLE 3.7 We now reconsider Example 3.6. $V_3[\mathcal{A}]$ is still changed from *true* to $(s \neq \text{"}CA\text{"})$ but now $V_4[\mathcal{P}]$ is set to *false*. We showed that $XV_{3.4}[\mathcal{S}]$ is $XV_{3.4}(mID, e)$:– $V_3(mID, s, e)$ in Section 2. Therefore, the rule that determines what to insert into $XV_{3.4}$ is $\triangle XV_{3.4}(mID, e)$:– $\triangledown V_3(mID, s, e)$. This results in the insertion of $\{\langle A, 100\rangle, \langle B, 150\rangle\}$ into $XV_{3.4}$ as expected.

$XV_{3.4}[\mathcal{A}]$ must also be recomputed to maintain data consistency. Since $V_3[\mathcal{A}]$ was *true* when V_4 was defined and $V_3[\mathcal{A}]$ is $(s \neq \text{"}CA\text{"})$, $\mathcal{C}_1$ is computed to be $(s = \text{"}CA\text{"})$. Assuming $\mathcal{C}_2$ is *true* for now, $XV_{3.4}[\mathcal{A}]$ is computed to be $(s = \text{"}CA\text{"})$ as desired. □

Part (3) of ***Compensate$\mathcal{A}$*****:** If $V_j[\mathcal{K}]$ is *false*, it indicates that the owner wants to expire tuples from the auxiliary views of V_j based on changes to $V_j[\mathcal{A}]$. Part (3) of *Compensate$\mathcal{A}$* attempts to do just that. Selecting the tuples to expire from an auxiliary view $XV_{k.j}$ based on changes to $V_j[\mathcal{A}]$ involves augmenting the rules in $VMR^{\mathcal{E}}$ of V_j with $V_j[\mathcal{A}]$ and running *Compute$\mathcal{C}_2$* over these rules. We discussed this in Section 3.2.3 but we argued that augmenting the rules with $V_j[\mathcal{A}]$ is of no use when V_j is just initialized. In the next example, we illustrate how this augmentation can be useful once $V_j[\mathcal{A}]$ is changed.

EXAMPLE 3.8 In this example, we assume that the owner of V_4 is no longer interested in sales on the first day of December and changes $V_4[\mathcal{A}_o]$ from *true* to $(d \neq 12/1/*)$. One of the view maintenance rules for V_4 is the rule $\triangle V_4$:– $\triangle V_1, XV_{2.4}[\mathcal{E}], V_3[\mathcal{E}], d = 12/*/*, itID > 100$. It follows that the augmented rule is

$$\triangle V_4 :- \triangle V_1, XV_{2.4}[\mathcal{E}], V_3[\mathcal{E}], d = 12/*/*, itID > 100, d \neq 12/1/*\,.$$

Moreover, $(d = 12/1/*)$ will be appended to all the view maintenance rules of V_4. Assuming the modification constraints are all *true*, when *Compute$\mathcal{C}_2$* is run over the augmented rules, $\mathcal{C}_2$ is set to $(d \neq 12/1/*)$ instead of *true* expiring more $XV_{2.4}$ tuples. □

In summary, when $V_j[\mathcal{A}]$ is changed, $\mathcal{C}_2$ of $XV_{k.j}[\mathcal{A}]$ is reevaluated using *Compute$\mathcal{C}_2$*. $\mathcal{C}_1$ need not be reevaluated since it is only affected by $V_k[\mathcal{A}]$. If $XV_{k.j}[\mathcal{A}]$ becomes stronger, some tuples need to be expired from $XV_{k.j}$. Finding the tuples to expire from $XV_{k.j}$ is the same problem as finding the tuples to expire from V_j when $V_j[\mathcal{A}]$ is made stronger. Therefore, *GetTuplesToExpire* can be used. (Appendix A discusses how *GetTuplesToExpire* is used for auxiliary views.)

3.3.2 Changing $V_j[\mathcal{I}]$, $V_j[\mathcal{D}]$, $V_j[\mathcal{U}]$

Modification constraints of base views are changed by the WHA when he learns more about what type of modifications are made to base views. Just like changes to $V_j[\mathcal{A}]$, we assume that modification constraints are only made stronger. Since modification constraints play a role in computing availability constraints of auxiliary views, it is to be expected that changes to modification constraints affect the availability constraints and extensions of auxiliary views.

The algorithm that implements the compensating actions is *Compensate*$\mathcal{M}$ (Figure 13, Appendix A). It first determines if the auxiliary views of a non-base view V_j are affected by the change to the modification constraint of a base view V_b. *Compensate*$\mathcal{M}$ does this by expanding each view specification until only base views are accessed. If the rewritten rules access V_b, the auxiliary views of V_j may be affected. Thus for each auxiliary view $XV_{k.j}$ of V_j, $XV_{k.j}[\mathcal{A}]$ is recomputed using *Compute*$\mathcal{C}_2$ (Section 3.2.4). There is no need to recompute $\mathcal{C}_1$ since it is only affected by changes to $V_k[\mathcal{A}]$. If $XV_{k.j}[\mathcal{A}]$ becomes stronger, *GetTuplesToExpire* is called to find the tuples that need to be expired.

4 Implementing the Framework

We now illustrate how our framework can be implemented on top of a conventional RDBMS. The main idea is to separate the data of the views (extensions and controls) from the logic for managing the views. The data of the views is stored in the database while the logic for managing the views are distributed among various objects (e.g., instances of C++ classes) which we now describe.

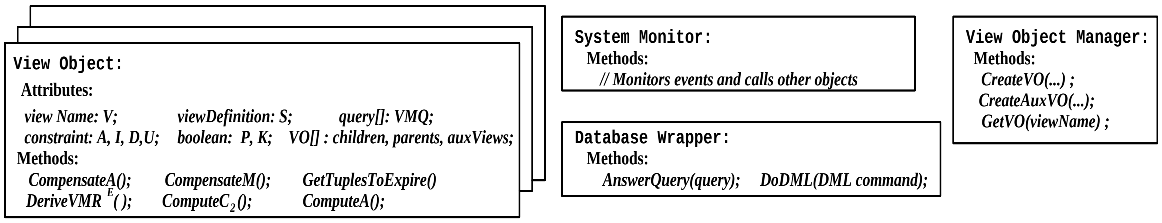

Figure 8: Interface of Various Object Types

Each view V_j has an instance of a `View Object` denoted by $\mathbf{VO_j}$. Object $\mathbf{VO_j}$ maintains V_j using the view maintenance queries determined by *DeriveVMR*$^{\mathcal{E}}$ (Section 3.2.1). Object $\mathbf{VO_j}$ also performs compensating actions (using the algorithms developed in Section 3.3) when the V_j controls are changed.

There is also an instance (denoted as **DW**) of a `Database Wrapper` that is used to communicate requests to the database. For example, a $\mathbf{VO_j}$ object may ask **DW** to create a relation for storing the extension of a view. A $\mathbf{VO_j}$ may also ask **DW** to answer an SQL query. The **DW** object issues these requests using the embedded SQL interface of the database.

There are two more types of objects that are needed: `System Monitor` and `View Object Manager`. We assume here a minimal configuration wherein there is only one instance of the `System Monitor` (denoted as **SM**) and `View Object Manager` (denoted as **VOM**). The **SM** detects events initiated by the user (e.g., view definitions and control changes) and calls other objects to handle these events. The **VOM** initializes `View Object` instances. The interfaces of the four object types are shown in Figure 8. Notice that each `View Object` instance will have in-memory copies of the view controls to enhance (read) performance.

This distributed object design allows for scalability and reliability. For example if we run objects under different processes, then failures or delays in one object do not necessarily affect other components. For instance, if the process associated with $\mathbf{VO_j}$ fails, the whole system does not fail since the logic for view maintenance and compensation of control changes are distributed among many instances of `View Objects`. Also, more than one instance of the `Database Wrapper`, `View Object Manager`, and `System Monitor`, can be created in order to distribute the processing load. Lastly, we have designed the system so that the RDBMS that is used has minimal requirements. The RDBMS is just needed to provide persistent storage and to answer SQL queries. View maintenance is not required and is done by the `View Object` instances.

We now present two examples to illustrate how the objects interact. Figure 9 shows the object interaction when a user defines V_4 over V_1, V_2 and V_3 (as in Section 2). As discussed in Section 3, the user provides the values for the controls $\mathcal{S}$, $\mathcal{P}$, $\mathcal{K}$, $\mathcal{I}$, $\mathcal{D}$, $\mathcal{U}$. The **SM** detects the view definition and records the input (Step 1). It then calls **VOM**.*CreateVO* to create $\mathbf{VO_4}$ (Step 2). Method **VOM**.*CreateVO* initializes $\mathbf{VO_4}$ by copying the user inputs into $\mathbf{VO_4}$'s attributes. Once created, $\mathbf{VO_4}$ initializes its availability constraint $\mathcal{A}$ and *VMQ* using its methods *Compute*$\mathcal{A}$ and *DeriveVMR*$^{\mathcal{E}}$ respectively. At this point, $\mathbf{VO_4}$ calls **DW**.*DoDML* to create the relation V_4-EXT to hold the extension of V_4. The attributes of V_4-EXT are just the projected attributes in $\mathbf{VO_4}.\mathcal{S}$ (Step 3). Object $\mathbf{VO_4}$ then calls **DW**.*DoDML* to insert the tuples that result from the query $\mathbf{VO_4}.\mathcal{S}$ into V_4-EXT. $\mathbf{VO_4}$ also asks **DW** to insert the V_4 controls (currently stored in memory as $\mathbf{VO_4}$'s attributes) into the database. Once $\mathbf{VO_4}$ is initialized, **SM** calls **VOM**.*CreateAuxVO* to create the `View Object` instances (i.e., $\{\mathbf{VO_{j.4}}\}$) of the auxiliary views (Step 4). (We only show $\mathbf{VO_{1.4}}$ in the figure to avoid clutter.) Method **VOM**.*CreateAuxVO* determines $\mathbf{VO_{j.4}}.\mathcal{S}$ (Section 3.2) and sets $\mathbf{VO_{j.4}}.\mathcal{A}$ to *false*. Once created, $\mathbf{VO_{j.4}}$ calls **DW**.*DoDML* to create the relation $XV_{j.4}$-EXT whose attributes are the projected attributes of $\mathbf{VO_{j.4}}.\mathcal{S}$ (Step 5). There is no need to insert any tuples into $XV_{j.4}$-EXT since the initial extension of any auxiliary view is empty (Section 3.2.1).

We now show how the objects interact when a view control is changed (Figure 10). Assume that the $\mathcal{A}$ control of V_1 is changed by the owner. The **SM** initiates the compensating actions by calling the method $\mathbf{VO_1}$.*Compensate*$\mathcal{A}$ (Step 1). Object $\mathbf{VO_1}$ creates the query to determine the tuples to expire by calling its *GetTuplesToExpire* method. Once the query is determined, $\mathbf{VO_1}$ asks **DW** to answer it (Step 2). Assuming the $\mathcal{P}$ control of V_4 is *true*, $\mathbf{VO_1}$ changes V_4's $\mathcal{A}$ control (Step 3). Object $\mathbf{VO_1}$ will then initiate the compensating actions by calling $\mathbf{VO_4}$.*Compensate*$\mathcal{A}$ (Step 4). Finally, $\mathbf{VO_1}$ calls **DW** to actually expire the tuples (Step 5) obtained from Step 2. The usual "delete" SQL command can be used to expire the tuples from V_1-EXT.

We have implemented an initial prototype ([WGL+96]) of our distributed multi-object warehouse architecture, using the Corba framework and the ILU implementation from Xerox PARC [CJS+94]. In the prototype, each view is managed by a separate object, as described above. However, the expiration functionality has not yet been incorporated into the prototype. We are currently in the process of doing this.

5 Related Work

One of the problems that our framework tackles is how to maintain a view when only parts of the underlying views are available. Most of the work on view maintenance have assumed that the complete underlying views are available ([BLT86], [BT88], [CW91], [GL95], [GMS93], [Han87], [QW91] and many others). However,

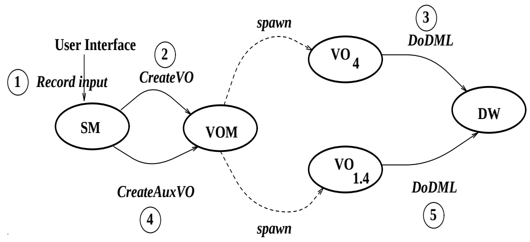

Figure 9: View Definition Object Interaction

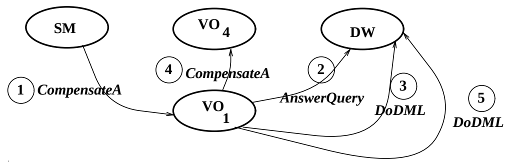

Figure 10: View Control Change Object Interaction

there has also been work on view maintenance that assume otherwise. [BT88] and [GJM96] endeavored to find *self-maintainable* views that can be maintained without accessing the unavailable underlying views. On the other hand, [QGMW96] tried to make a view self-maintainable by defining auxiliary views such that the view and the auxiliary views together are self-maintainable. [HZ96] also tried to define auxiliary views. (We showed in Section 3 how our auxiliary views relate with those of [HZ96] and [QGMW96].) In addition, [HZ96] developed a framework wherein attributes (i.e., columns) of the underlying views may be unavailable. In our framework, the tuples (i.e., rows) of a view or a relation can be made unavailable. We did not take [HZ96]'s approach because "expiring" attributes necessitates redefining the view and affecting all the views defined on that view. [JMS95] endeavored to find views that can be maintained when only a subset of the underlying views are available. More specifically, the large "chronicles" (i.e., sequences) were assumed to be unavailable and [JMS95] tried to limit the view definition language so that "chronicles" do not need to be accessed during view maintenance. In our framework, we can describe chronicles as append-only base views (Appendix B). If the view definition language is limited in a similar fashion, we believe that the auxiliary views of these base views will also be empty ([LGM97]).

Another problem that our framework tackled is describing the incompleteness of a single view. There have been numerous work in describing incomplete data in the area of incomplete databases. See [AHV95] for an overview. In particular, [IL84] has a more general representation of an incomplete view than ours in that they can associate a condition with each tuple t that uses variables that represent tuple attributes (not necessarily t's). In our framework, we can only associate a condition over a whole view that uses variables that represent the view attributes. [Gra84] augmented the conditional tables proposed by [IL84] with global conditions for each relation similar to our availability constraints. [Lev96] also used "local constraints" which are roughly equivalent to our availability constraints. [Dyr96] focused on describing incomplete datacubes (a set of aggregate views). They then used this description to describe query results (similar to our $Q[\mathcal{A}]$ of a query Q).

Another problem that our framework tackled is coping with control changes. In our framework, what is available for view maintenance and querying changes with the controls. We have developed algorithms that cope with control changes. The algorithms in [GMR95] can also be used to find $\bigtriangledown V_j$. Apart from [GMR95] and this paper, there has been no work on compensating for control changes.

6 Conclusions

We have presented a framework and design for system-managed removal of warehouse data. Within it, the warehouse administrator (WHA) provides specifications for what is expired or expunged from each view, and the system automatically takes appropriate action. Furthermore, the WHA can dynamically change the specifications, and the algorithms we have presented compensate as necessary.

We believe that expiration and expunction are extremely useful concepts in a warehouse. They make it possible to *gracefully* control the size of the warehouse, without requiring complete removal of entire materialized views (which is the common approach in today's systems). Furthermore, the availability and modification constraints provide an *explicit* record of what is contained in the warehouse and how it might change. This makes it possible for users to understand what they get from the warehouse, without having to rely on unwritten "community knowledge." The same explicit record also lets the system efficiently manage the materialized extensions.

References

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

[BLT86] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, pages 61–71, 1986.

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

[CJS+94] A. Courtney, W. Janssen, D. Severson, M. Spreitzer, and F. Wymore. Inter-language unification, release 1.5. Technical Report ISTL-CSA-94-01-01, Xerox PARC, May 1994.

[CM77] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of 1977 Ninth Annual ACM Symp. on the Theory of Computing*, pages 77–90, 1977.

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

[Dyr96] C. Dyreson. Information retrieval from an incomplete data cube. In *Proceedings of International Conference on Very Large Data Bases*, pages 532–543, 1996.

[GHM95] A. Gupta, H., and I. Mumick. Maintenance of materialized views: Problems, techniques, and applications. In *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, pages 3–19, 1995.

[GJM96] A. Gupta, H. Jagadish, and I. Mumick. Data integration using self-maintainable views. In *Proceedings of 1996 EDBT*, 1996.

[GL95] L. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of 1995 ACM SIGMOD*, pages 328–339, 1995.

[GMR95] A. Gupta, I. Mumick, and K. Ross. Adapting materialized views after redefinitions. In *Proceedings of 1995 ACM SIGMOD*, pages 211–222, 1995.

[GMS93] A. Gupta, I. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. In *Proceedings of 1993 ACM SIGMOD*, pages 157–166, 1993.

[Gra84] G. Grahne. Dependency satisfaction in databases with incomplete information. In *Proceedings of International Conference on Very Large Data Bases*, 1984.

[Han87] E. Hanson. A performance analysis of view materialization strategies. In *Proceedings of 1987 ACM SIGMOD*, pages 440–453, 1987.

[HZ96] R. Hull and G. Zhou. Framework for supporting data integration using the materialized and virtual approaches. In *Proceedings of 1996 ACM SIGMOD*, pages 481–492, 1996.

[IL84] T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.

Algorithm A.1 *GetTuplesToExpire*
Input V_j // view to be expired from
$\quad V_j[\mathcal{A}_{o\ or\ s}]'$ // the changed $V_j[\mathcal{A}_o]$ or $V_j[\mathcal{A}_s]$
Output ∇V_j // tuples to be expired from V_j
Method
(1) **if** Case (1) holds **then**
(2) $\nabla V_j :- V_j[\mathcal{E}], \neg V_j[\mathcal{A}_{o\ or\ s}]'$
(3) **else if** Case (2) or Case (3) **then**
(4) $\nabla V_j \leftarrow RHS(V_j[\mathcal{S}])^{\mathcal{E}}, V_j[\mathcal{A}_o], \neg V_j[\mathcal{A}_{o\ or\ s}]'$
(5) **else** // Case (4)
(6) $\nabla V_j \leftarrow \phi$ and refuse to change $V_j[\mathcal{A}_{o\ or\ s}]$ // not possible to expire!
(7) return ∇V_j

Figure 11: Algorithm for Obtaining the Tuples to Expire

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

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

[LGM97] W. Labio and H. Garcia-Molina. Expiration of data. Technical report, Stanford University, 1997. Available by anonymous ftp from db.stanford.edu in /pub/labio/1997.

[Mum91] Inderpal Singh Mumick. *Query Optimization in Deductive and Relational Databases.* PhD thesis, Stanford University, Stanford, CA 94305, 1991.

[QGMW96] D. Quass, A. Gupta, I.S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. Technical report, Stanford University, 1996. Available by anonymous ftp from db.stanford.edu in /pub/quass/1996.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *ACM TKDE*, 1991.

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

[WGL+96] J. Wiener, H. Gupta, W. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom. A system prototype for warehouse view maintenance. In *Workshop on Materialized Views: Techniques and Applications*, pages 26–33, 1996.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of 1995 ACM SIGMOD*, 1995.

A Algorithms

In this appendix, we discuss each algorithm in turn. Note that the steps of the algorithms are at a high level.

GetTuplesToExpire (Figure 11) determines the tuples to expire from V_j when either $V_j[\mathcal{A}_s]$ or $V_j[\mathcal{A}_o]$ is changed. It checks which case holds (see Section 3.3.1) and uses the appropriate rule. When *GetTuplesToExpire* is used to find tuples to expire from an auxiliary view $XV_{k.j}$, only Case (1) is used because *Compute*$\mathcal{C}_2$ ensures that only distinguished variables are used in $\mathcal{C}_2$. Changes to $\mathcal{C}_1$ do not result in expiring $XV_{k.j}$ tuples (Section 3.3.1) and *GetTuplesToExpire* is not used. Also, $XV_{k.j}[\mathcal{A}]$ is not separated into two portions unlike $V_j[\mathcal{A}]$.

Compensate$\mathcal{A}$ (Figure 12) is used to compensate for changes to $V_j[\mathcal{A}]$. It first finds the tuples to expire ∇V_j (Lines (1)-(4)). For each view V_i defined on V_j, it either expires the tuples that are derived from ∇V_j (Lines (6)-(8)) or saves parts of ∇V_j in $XV_{i.j}$ (Lines (9)-(12)). If $V_j[\mathcal{K}]$ is *false*, it tries to expire tuples from $XV_{k.j}$ (Lines (14)-(16)). Finally, it removes ∇V_j from the database.

Algorithm A.2 *Compensate$\mathcal{A}$* // when this is called, $V_j[\mathcal{A}]$ has changed
Input V_j // view whose $V_j[\mathcal{A}]$ changed
Method
(1) **if** $V_j[\mathcal{A}_o]$ changed **then** // get tuples to be expired
(2) $\triangledown V_j \leftarrow GetTuplesToExpire(V_j, V_j[\mathcal{A}_o]')$
(3) **else** // $V_j[\mathcal{A}_s]$ changed
(4) $\triangledown V_j \leftarrow GetTuplesToExpire(V_j, V_j[\mathcal{A}_s]')$

(5) **for** each V_i defined on V_j **do**
(6) **if** $V_i[\mathcal{P}] = true$ **then** // treat $\triangledown V_j$ as deleted tuples
(7) Recompute $V_i[\mathcal{A}_s]$ using *Compute$\mathcal{A}$* // change $V_i[\mathcal{A}]$
(8) *Compensate$\mathcal{A}$*(V_i) // compensate change to $V_i[\mathcal{A}]$
(9) **else** // treat $\triangledown V_j$ as expired tuples
(10) Recompute $XV_{j.i}[\mathcal{A}]$ // reevaluate expression $\mathcal{C}_1$ and change $XV_{j.i}[\mathcal{A}]$
(11) Let R be the RHS of $XV_{j.i}[\mathcal{S}]$ but with V_j replaced by $\triangledown V_j$
(12) Insert into $XV_{j.i}$ the result of $\triangle XV_{j.i}$:– R // compensate change to $XV_{j.i}[\mathcal{A}]$

(13) **if** $V_j[\mathcal{K}] = false$ **then** // do not keep supporting auxiliary view tuples
(14) **for** each V_k that V_j is defined on **then**
(15) Recompute $XV_{k.j}[\mathcal{A}]$ // reevaluate expression $\mathcal{C}_2$ and change $XV_{k.j}[\mathcal{A}]$
(16) Expire result of $GetTuplesToExpire(XV_{k.j}, XV_{k.j}[\mathcal{A}]')$ // compensate change to $XV_{j.i}[\mathcal{A}]$

(17) Ask DB to remove $\triangledown V_j$ from disk // actually remove tuples

Figure 12: Algorithm to Compensate for Changes to $V_j[\mathcal{A}]$

Algorithm A.3 *Compensate$\mathcal{M}$*
Input V_b // the base view whose $V_b[\mathcal{I}]$ or $V_b[\mathcal{D}]$ changed
Method
(1) **for** each non base view V_j **do**
(2) $\{R\} \leftarrow$ expand $V_j[\mathcal{S}]$ until only base views are used
(3) **if** any rule in $\{R\}$ uses V_b **then**
(4) **for** each auxiliary view $XV_{k.j}$ of V_j **do**
(5) Reevaluate $XV_{k.j}[\mathcal{A}]$ // recompute $\mathcal{C}_2$ using *Compute$\mathcal{C}_2$*
(6) Expire result of $GetTuplesToExpire(XV_{k.j}, XV_{k.j}[\mathcal{A}]')$

Figure 13: Algorithm to Compensate for Changes to $V_b[\mathcal{I}]$, $V_b[\mathcal{D}]$, $V_b[\mathcal{U}]$

Algorithm A.4 *Compute$\mathcal{A}$*
Input V_j // the view whose $V_j[\mathcal{A}]$ is being computed
$\{V_k\}$ // the underlying views of V_j
Output $V_j[\mathcal{A}]$ // the availability constraint of V_j
Method
(1) **for** each rule r in $V_j[\mathcal{E}] :- RHS(V_j[\mathcal{S}])^{\mathcal{E}}$ **do** // *Step 1*
(2) **for** each ordinary predicate $V_k[\mathcal{E}]$ in r **do**
(3) Replace $V_k[\mathcal{E}]$ by the RHS of $RHS(V_k[\mathcal{S}]), V_k[\mathcal{A}]$ renaming variables as necessary
(4) **for** each rewritten rule r **do** // *Step 2*
(5) **for** each predicate s in r **do**
(6) **if** s originated from $V_j[\mathcal{S}]$ **then**
(7) eliminate s
(8) **else if** s is a built-in predicate from some $RHS(V_k[\mathcal{S}])$ **or**
s is an ordinary predicate that is not required for safety **then**
(9) eliminate s
(10) Return the disjunction of the RHS's of the rewritten and reduced rules

Figure 14: Algorithm for Computing $V_j[\mathcal{A}]$

Compensate$\mathcal{M}$ (Figure 13) is used to compensate for changes to modification constraints. For each non-base view V_j, it determines if its auxiliary views are affected by the changes (Lines (2)-(3)). If so, it uses *Compute$\mathcal{C}_2$* (Figure 16) to recompute $\mathcal{A}$ for each auxiliary view $XV_{k.j}$ of V_j (Lines (4)-(6)). *GetTuplesToExpire* is used to find the tuples to expire from $XV_{k.j}$.

Compute$\mathcal{A}$ (Figure 14) computes $V_j[\mathcal{A}_s]$. *Compute$\mathcal{A}$* rewrites each rule r in $V_j[\mathcal{E}] :- RHS(V_j[\mathcal{S}])^{\mathcal{E}}$ by replacing each underlying view extension $V_k[\mathcal{E}]$ with $RHS(V_k[\mathcal{S}]), V_k[\mathcal{A}]$ (Lines (1)-(3)). It then eliminates the redundant predicates and then returns the disjunction of the RHS's of the rewritten rules (Lines (4)-(9)).

Compute$\mathcal{I}$ (Figure 15) is used to compute $V_j[\mathcal{I}]$ for some non-base view V_j. *Compute$\mathcal{I}$* rewrites the $VMR^{\mathcal{E}}$ of V_j by expanding each $\triangle V_k$ (V_k is a non-base view) with the $VMR^{\mathcal{E}}$ of V_k (Lines (1)-(4)). Once the rules only access delta predicates of base views, *Compute$\mathcal{I}$* returns the disjunction of the $V_b[\mathcal{I}]$'s for each base view delta predicate $\triangle V_b$ accessed (Lines (5)-(11)). Since $\mathcal{C}_2$ is a "peephole" optimization, we are conservative in computing $\mathcal{I}$. Once $V_b[\mathcal{I}]$ uses a non-distinguished variables of $V_j[\mathcal{S}]$, *Compute$\mathcal{I}$* returns *true*. *Compute$\mathcal{D}$* is very similar. $V_j[\mathcal{U}]$ is computed by taking the union of $V_k[\mathcal{U}]$ for each underlying view V_k but disregarding non-distinguished variables of V_j.

DeriveVMR$^{\mathcal{E}}$ (Figure 16) is used to compute $VMR^{\mathcal{E}}$ of a view V_j. It takes as input the conventional view maintenance rules of V_j that uses just the auxiliary views. It then replaces each $XV_{k.j}$ by either $XV_{k.j}[\mathcal{E}]$ or $V_k[\mathcal{E}], B_k$ (B_k are the selection conditions in $XV_{k.j}[\mathcal{S}]$) making all possible substitution combination.

Compute$\mathcal{C}_2$ (Figure 16) is used to compute $\mathcal{C}_2$ for some $XV_{k.j}[\mathcal{A}]$. It first rewrites the conventional view maintenance rules using *DeriveVMR$^{\mathcal{E}}$*. It then eliminates the rules in $VMR^{\mathcal{E}}$ of V_j by considering modification constraints. The remaining rules are augmented with $V_j[\mathcal{A}]$ if possible. Finally, it chooses the rules that use $XV_{k.j}[\mathcal{E}]$. All this rewriting is done in Lines (1)-(6). It examines the rewritten rules and selects predicates that can be added to $\mathcal{C}_2$ as discussed in Section 3.2.2 (Lines (7)-(20)).

B Other Modification Constraints

In this appendix, we show how the auxiliary views of V_4 (introduced in Section 2) can be made smaller by considering modification constraints that express "append-only" insertions and referential integrity constraints.

Algorithm A.5 *Compute$\mathcal{I}$* // *Compute$\mathcal{D}$* is similar
Input V_j // the non-base view whose $V_j[\mathcal{I}]$ is being computed
Output $V_j[\mathcal{I}]$ // the effective insertion constraint of V_j
Method
(1) $\{R\} \leftarrow VMR^{\mathcal{E}}$ of V_j
(2) **for** each $R \in \{R\}$ **do**
(3) **if** the delta predicate $\triangle V$ is in the rule **and** V is not a base view **then**
(4) Replace $\triangle V$ by the $RHS(VMR^{\mathcal{E}})$ of V
(5) $V_j[\mathcal{I}] \leftarrow$ *false* // Initialize $V_j[\mathcal{I}]$
(6) **for** each $R \in \{R\}$ **do** // go through the rewritten rules
Let the delta predicate in R be $\triangle V_b$
(7) **if** $V_b[\mathcal{I}]$ uses a non-distinguished variable of $V_j[\mathcal{S}]$
(8) Return *true*
(9) $V_j[\mathcal{I}] \leftarrow V_j[\mathcal{I}] \vee V_b[\mathcal{I}]$
(10) Return $V_j[\mathcal{I}]$

Figure 15: Algorithm for Computing $V_j[\mathcal{I}]$

Among the views in our working example, it is reasonable to assume that V_1 (which is a copy of *line*) is append-only. That is, there are only insertions to V_1 and furthermore, they only refer to the last sale or the newest sale. More specifically, the sID's of the insertions are equal to either the current maximum sID value in V_1 or one more than that value. This constraint can be expressed by setting $V_1[\mathcal{I}]$ to $V_1MAXsID[\mathcal{E}](MAXsID) \wedge (sID \geq MAXsID)$. The WHA defines $V_1MAXsID$ to contain a single tuple that records the current maximum sID of V_1. Since there is only one tuple in $V_1MAXsID$, the WHA sets $V_1MAXsID[\mathcal{A}]$ to *true*. Revisiting Rules (23) through (26) (Section 3.2.2), and assuming Rules (25) and (26) are eliminated using referential integrity constraints (shown next), $\mathcal{C}_2$ of $XV_{2.4}[\mathcal{A}]$ is set to

$$V_1MAXsID[\mathcal{E}](MAXsID) \wedge (sID \geq MAXsID) \wedge (XV_{3.4}[\mathcal{E}] \vee V_3[\mathcal{E}]).$$

This results in a significant savings in space because $XV_{2.4}$ only has to save tuples derived from the most recent sales tuples.

We now show how Rules (25) and (26) are eliminated using referential integrity constraints ignoring updates for now. We assume that there is a referential integrity constraint from V_2 to V_3 on the mID attribute. From the point of view of V_3, this means that if a tuple t_3 is inserted into V_3 with an mID value of "mID_3", there is no V_2 tuple with an mID value of "mID_3". This can be expressed by setting $V_3[\mathcal{I}]$ to $\neg V_2(X, mID, Y)$. Notice that this $V_3[\mathcal{I}]$ implies that mID is the key of V_3 as well. If $\triangle V_3$, $V_3[\mathcal{I}]$ is substituted for $\triangle V_3$ in Rules (25) and (26), the RHS's of both rules evaluate to *false* and the rules can be ignored.

Lastly, it is also important to know when it is possible to propagate updates to V_k directly to a view V_j as opposed to propagating them as deletions followed by insertions. This is possible when the updates never modify key V_k attributes (which are distinguished in $V_j[\mathcal{S}]$) or attributes involved in selection conditions or join conditions of $V_j[\mathcal{S}]$. These V_k updates were called *protected* updates in [QGMW96]. Whether V_k updates are protected (w.r.t. some view V_j) or not can be checked using the $V_k[\mathcal{U}]$ modification constraint. If updates are not protected, the updates must be described in $V_k[\mathcal{I}]$. Thus, a $V_k[\mathcal{I}]$ has two disjuncts: $\mathcal{I}_{ins}$ that describes insertions and $\mathcal{I}_{upd}$ that describes the new updated tuples in case there are unprotected updates. In the previous example, we assumed $\mathcal{I}_{upd}$ of $V_3[\mathcal{I}]$ was *false*. Since V_k updates are protected w.r.t. some view V_j, the insertion constraint must be associated with both V_k and V_j (i.e., $V_{k.j}[\mathcal{I}]$).

Algorithm A.6 *DeriveVMR*$^{\mathcal{E}}$
Input VMR of V_j // view maintenance rules that use auxiliary views and modifications
$\{XV_{k.j}\}$ // the auxiliary views of V_j
Output $VMR^{\mathcal{E}}$ of V_j // view maintenance rules that use extensions
Method
(1) $VMR^{\mathcal{E}} = \{\}$
(2) **for** each rule R in VMR **do**
(3) $VMR^{\mathcal{E}} \leftarrow VMR^{\mathcal{E}} \cup$
For each $XV_{k.j}$ in R, substitute either $XV_{k.j}[\mathcal{E}]$, B_k or $V_k[\mathcal{E}]$.
Make all possible combinations of substitutions.
(B_k are the built-in predicates in $XV_{k.j}[\mathcal{S}]$.)
(4) Return $VMR^{\mathcal{E}}$

Algorithm A.7 *Compute*$\mathcal{C}_2$
Input $XV_{k.j}$ // the auxiliary view whose $XV_{k.j}[\mathcal{A}]$ is being calculated
$\{R\}$ // view maintenance rules that use auxiliary views from [QGMW96]
V_j // the view being maintained
Output $\mathcal{C}_2$ of $XV_{k.j}[\mathcal{A}]$
Method
(1) $\{R\} \leftarrow DeriveVMR^{\mathcal{E}}(\{R\})$
(2) $\{R\} \leftarrow$ rewrite the rules by replacing $\triangle V$ with $\triangle V, V[\mathcal{I}]$ and $\bigtriangledown V$ with $\bigtriangledown V, V[\mathcal{D}]$
(Use *Compute*$\mathcal{I}$ or *Compute*$\mathcal{D}$ if V is a not a base view.)
(3) eliminate rules in $\{R\}$ that produce no modifications
(4) **if** $V_j[\mathcal{K}] = false$ **then**
(5) augment $\{R\}$ with $V_j[\mathcal{A}]$
(6) $\{R'\} \leftarrow$ subset of $\{R\}$ that uses the predicate $XV_{k.j}[\mathcal{E}]$
(7) $\mathcal{C}_2 \leftarrow false$ // will contain expression to be returned
(8) **for** each rule $R' \in \{R'\}$ **do**
(9) $c \leftarrow true$ // will contain conjunction R' predicates included in $\mathcal{C}_2$
(10) $\{s\} \leftarrow$ the predicates of R'
(11) $VARS \leftarrow$ Vars(predicate $XV_{k.j}[\mathcal{E}]$ in R') // initialize to vars used by $XV_{k.j}[\mathcal{E}]$
(12) **while** there is an $s_0 \in \{s\}$ that is not $XV_{k.j}[\mathcal{E}]$ nor a delta predicate **and**
$Var(s_0) \cap VARS \neq \phi$
(13) **if** s_0 represents some $XV_{l.j}[\mathcal{E}]$ or $V_l[\mathcal{E}]$ **and** // If $\triangle V_l$ is in some rule,
there exists $R'' \in \{R'\}$ with $\triangle V_l$ **then** // cannot add $XV_{l.j}[\mathcal{E}]$ nor $V_l[\mathcal{E}]$
pass
(14) **else**
(15) $c \leftarrow c \wedge s_0$ // s_0 passes criteria and is included in $\mathcal{C}_2$
(16) **if** s_0 is an ordinary predicate **then**
(17) $VARS \leftarrow VARS \cup Var(s_0)$
(18) $\{s\} \leftarrow \{s\} - \{s_0\}$
(19) $\mathcal{C}_2 \leftarrow \mathcal{C}_2 \vee c$
(20) Return $\mathcal{C}_2$

Figure 16: Algorithm for Computing $\mathcal{C}_2$