Making Views Self-Maintainable for Data Warehousing

Dallan Quass* Stanford University quass@cs.stanford.edu

Ashish Gupta IBM Almaden agupta@cs.stanford.edu

Inderpal Singh Mumick AT&T Research mumick@research.att.com

Jennifer Widom* Stanford University widom@cs.stanford.edu

Abstract

A data warehouse stores materialized views over data from one or more sources in order to provide fast access to the integrated data, regardless of the availability of the data sources. Warehouse views need to be maintained in response to changes to the base data in the sources. Except for very simple views, maintaining a warehouse view requires access to data that is not available in the view itself. Hence, to maintain the view, one either has to query the data sources or store auxiliary data in the warehouse. We show that by using key and referential integrity constraints, we often can maintain a select-project-join view when there are insertions, deletions, and updates to the base relations without going to the data sources or replicating the base relations in their entirety in the warehouse. We derive a set of auxiliary views such that the warehouse view and the auxiliary views together are *self-maintainable*—they can be maintained without going to the data sources or replicating all base data. In addition, our technique can be applied to simplify traditional materialized view maintenance by exploiting key and referential integrity constraints.

1 Introduction

The problem of *materialized view maintenance* has received increasing attention recently [GM95, GM96,Mum95], particularly due to its application to *data warehousing* [DEB95,ZGHW95]. A view is a derived relation defined in terms of base relations. A view is said to be materialized when it is stored in the database, rather than computed from the base relations in response to queries. The materialized view maintenance problem is the problem of keeping the contents of the stored view consistent with the contents of the base relations as the base relations are modified.

Data warehouses store materialized views in order to provide fast access to information that is integrated from several distributed *data sources* [DEB95]. The data sources may be heterogeneous and/or remote from the warehouse. Consequently, the problem of maintaining a materialized view in a data warehouse differs from the traditional view maintenance problem where the view and base data are stored in the same database. In particular, when changes are reported by one data

*This work was supported by Rome Laboratories under Air Force Contract F30602-94-C-023 and by equipment grants from Digital and IBM Corporations.

source it may be necessary to access base data from other data sources in order to maintain the view [HGW+95].

For any view involving a join, maintaining the view when base relations change may require accessing base data, even when *incremental view maintenance* techniques are used [GL95,GMS93]. For example, for a view $R \bowtie S$, when an insertion to relation R is reported it is usually necessary to query S in order to discover which tuples in S join with the insertion to R. In the warehousing scenario, accessing base data means either querying the data sources or replicating the base relations in the warehouse. The problems associated with querying the data sources are that the sources may periodically be unavailable, may be expensive or time-consuming to query, and inconsistencies can result at the warehouse unless care is taken to avoid them through the use of special maintenance algorithms [ZGHW95]. The problems associated with replicating base relations at the warehouse are the additional storage and maintenance costs incurred. In this paper we show that for many views, including views with joins, if key and referential integrity constraints are present then it is not necessary to replicate the base relations in their entirety at the warehouse in order to maintain a view. We give an algorithm for determining what extra information, called *auxiliary views*, can be stored at a warehouse in order to maintain a select-project-join view without accessing base data at the sources. The algorithm takes key and referential integrity constraints into account, which are often available in practice, to reduce the sizes of the auxiliary views. When a view together with a set of auxiliary views can be maintained at the warehouse without accessing base data, we say the views are *self-maintainable*.

Maintaining materialized views in this way is especially important for *data marts*—miniature data warehouses that contain a subset of data relevant to a particular domain of analysis or geographic region. As more and more data is collected into a centralized data warehouse it becomes increasingly important to distribute the data into localized data marts in order to reduce query bottlenecks at the central warehouse. When many data marts exist, the cost of replicating entire base relations (and their changes) at each data mart becomes especially prohibitive.

1.1 Motivating Example

We start with an example showing how the amount of extra information needed to maintain a view can be significantly reduced from replicating the base relations in their entirety. Here we present our results without explanation of how they are obtained. We will revisit the example throughout the paper.

Consider a database of sales data for a chain of department stores. The database has the following relations.

```
store(store_id, city, state, manager)
sale(sale_id, store_id, day, month, year)
line(line_id, sale_id, item_id, sales_price)
item(item_id, item_name, category, supplier_name)
```

The first (underlined) attribute of each relation is a key for the relation. The `store` relation contains the location and manager of each store. The `sale` relation has one record for each sale transaction, with the store and date of the sale. A sale may involve several items, one per line on a sales receipt, and these are stored in the `line` relation, with one tuple for every item sold in the transaction. The `item` relation contains information about each item that is stocked. We assume that the following referential integrity constraints hold: (1) from `sale.store_id` to `store.store_id`, (2) from `line.sale_id` to `sale.sale_id`, and (3) from `line.item_id` to `item.item_id`. A referential integrity constraint from $S.B$ to $R.A$ implies that for every tuple $s \in S$ there must be a tuple $r \in R$ such that $s.B = r.A$.

Suppose the manager responsible for toy sales in the state of California is interested in maintaining a view of this year's sales: "all toy items sold in California in 1996 along with the sales price, the month in which the sale was made, and the name of the manager of the store where the sale was made. Include the item id, the sale id, and the line id."

```
CREATE VIEW cal_toy_sales AS
SELECT store.manager, sale.sale_id, sale.month, item.item_id, item.item_name, line.line_id, line.sales_price
FROM   store, sale, line, item
WHERE  store.store_id = sale.store_id and
       sale.sale_id = line.sale_id and
       line.line_id = item.item_id and
       store.state = "CA" and
       sale.year = 1996 and
       item.category = "toy"
```

The question addressed in this paper is: Given a view such as the one above, what auxiliary views can be materialized at the warehouse so that the view and auxiliary views together are self-maintainable?

Figure 1 shows SQL expressions for a set of three auxiliary views that are sufficient to maintain view `cal_toy_sales` for insertions and deletions to each of the base relations, and are themselves self-maintainable. In this paper we give an algorithm for deriving such auxiliary views in the general case, along with incremental maintenance expressions for maintaining the original view and auxiliary views. Materializing the auxiliary views in Figure 1 represents a significant savings over materializing the base relations in their entirety, as illustrated in Table 1.

Suppose that each of the four base relations contain the number of tuples listed in the first column of Table 1. Assuming that the selectivity of `store.state="CA"` is .02, the selectivity of `sale.year=1996` is .25, the selectivity of `item.category="toy"` is .05, and that distributions are uniform, the number of tuples passing local selection conditions (selection conditions involving attributes from a single relation) are given in the second column of Table 1. A related proposal by Hull and Zhou [HZ96] achieves self-maintainability for base relation insertions by pushing down projections and local selection conditions on the base relations and storing at the warehouse only those tuples and attributes of the base relations that pass the selections and projections. Thus, their approach would require that the number of tuples appearing in the second column of Table 1 is stored at the warehouse to handle insertions.

We improve upon the approach in [HZ96] by also taking key and referential integrity constraints into account. For example, we don't need to materialize any tuples from `line`, because the key and

```
CREATE VIEW aux_store AS
SELECT store_id, manager
FROM   store
WHERE  state = "CA"

CREATE VIEW aux_sale AS
SELECT sale_id, store_id, month
FROM   sale
WHERE  year = 1996 and
       store_id IN (SELECT store_id FROM aux_store)

CREATE VIEW aux_item AS
SELECT item_id, item_name
FROM   item
WHERE  category = "toy"
```

Figure 1: Auxiliary Views for Maintaining the `cal_toy_sales` View

referential integrity constraints guarantee that existing tuples in `line` cannot join with insertions into the other relations. Likewise we can exclude tuples in `sale` that do not join with existing tuples in `store` whose state is California, because we are guaranteed that existing tuples in `sale` will never join with insertions to `store`. Using our approach can dramatically reduce the number of tuples in the auxiliary views over pushing down selections only. The number of tuples required by our approach to handle base relation insertions in our example appears in the third column of Table 1.

Base Relation	Tuples in Base Relation	Tuples Passing Local Selection Conditions	Tuples in Auxiliary Views of Figure 1
`store`	2,000	40	40
`sale`	80,000,000	20,000,000	400,000
`line`	800,000,000	800,000,000	0
`item`	1,000	50	50
Total	880,003,000	820,000,090	400,090

Table 1: Number of Tuples in Base Relations and Auxiliary Views

We can similarly use key constraints to handle deletions to the base relations without all the base relations being available. We can determine the effects of deletions from `sale`, `line`, and `item` without referencing any base relations because `cal_toy_sales` includes keys for these relations. We simply join the deleted tuples with `cal_toy_sales` on the appropriate key. Even though the view does not include a key for `store`, `store` is joined to `sale` on the key of `sale`, so the effect of deletions from `store` can be determined by joining the deleted tuples with `sale` and joining the result with `cal_toy_sales` on the key of `sale`.

Now consider updates. If all updates were treated as deletions followed by insertions, as is common in view maintenance, then the properties of key and referential integrity constraints that we use to reduce the size of auxiliary views would no longer be guaranteed to hold. Thus, updates are treated separately in our approach. Note that in data warehousing environments it is common for certain base relations not to be updated (e.g., relations `sale` and `line` may be append only). Even when base relations are updateable, it may be that not all attributes are updated (e.g., we don't expect to update the `state` of a `store`). If updates to the base relations in our example cannot change the values of attributes involved in selection conditions in the view, then the auxiliary views of Figure 1 are sufficient (even if attributes appearing in the view may be updated). If, on the other hand, updates to `sale` may change the year (for example), then an additional auxiliary view:

```
CREATE VIEW aux_line AS
SELECT line_id, sale_id, item_id, sales_price
FROM   line
WHERE item_id IN (SELECT item_id FROM aux_item)
```

would need to be materialized, which would have 40,000,000 tuples. That is, we would need to store all purchases of items whose category is "toy," in case the year of the corresponding sales record is changed later to 1996. Further, if updates may change the category of an item to "toy", we would need to keep all of the `line` relation in order to maintain the view.

In practice we have found that the attributes appearing in selection conditions in views tend to be attributes that are not updated, as in our example. As illustrated above and formalized later on, when such updates do not occur, much less auxiliary information is required for self-maintenance. Thus, exploiting knowledge of permitted updates is an important feature of our approach.

1.2 Self-Maintenance

Self maintenance is formally defined as follows. Consider a view V defined over a set of base relations $\mathcal{R}$. Changes, $\delta\mathcal{R}$, are made to the relations in $\mathcal{R}$ in response to which view V needs to be maintained. We want to compute δV, the changes to V, using as little extra information as possible. If δV can be computed using only the materialized view V and the set of changes $\delta\mathcal{R}$, then view V alone is self-maintainable. If view V is not self-maintainable, we are interested in finding a set of auxiliary views $\mathcal{A}$ defined on the same relations as V such that the set of views $\{V\}\cup\mathcal{A}$ is self-maintainable. Note that the set of base relations $\mathcal{R}$ forms one such set of auxiliary views. However, we want to find more "economical" auxiliary views that are much smaller than the base relations. The notion of a *minimal* set of auxiliary views sufficient to maintain view V is formalized in Section 3.

A more general problem is to make a set $\mathcal{V} = V_1, \ldots, V_n$ of views self-maintainable, i.e., find auxiliary views $\mathcal{A}$ such that $\mathcal{A}\cup\mathcal{V}$ is self-maintainable. Simply applying our algorithm to each view in $\mathcal{V}$ is not satisfactory, since opportunities to "share" information across original and auxiliary views will not be recognized. That is, the final set $\mathcal{A}\cup\mathcal{V}$ may not be minimal. We intend to investigate sets of views as future work.

1.3 Paper Outline

The paper proceeds as follows. Section 2 presents notation, terminology, and some assumptions. Section 3 presents an algorithm for choosing a set of auxiliary views to materialize that are sufficient for maintaining a view and are self-maintainable. Section 4 shows how the view is maintained using the auxiliary views. Section 5 explains how the auxiliary views are self-maintained. Related work and conclusions are presented in Section 6. Additional details and proofs are provided in the appendices.

2 Preliminaries

We consider *select-project-join* (SPJ) views; that is, views consisting of a single projection followed by a single selection followed by a single cross-product over a set of *base relations*. As usual, any combination of selections, projections, and joins can be represented in this form. We assume that all base relations have keys but that a view might contain duplicates due to the projection at the view. In this paper we assume single-attribute keys and conjunctions of selection conditions (no disjunctions) for simplicity, but our results carry over to multi-attribute keys and selection conditions with disjunctions. In Section 3 we will impose certain additional restrictions on the view but we explain how those restrictions can be lifted in Appendix B. We say that selection conditions involving attributes from a single relation are *local conditions*; otherwise they are *join conditions*. We say that attributes appearing in the final projection are *preserved* in the view.

In order to keep a materialized view up to date, changes to base relations must be propagated to the view. A *view maintenance expression* calculates the effects on the view of a certain type of change: insertions, deletions, or updates to a base relation. We use a *differential algorithm* as given in [GL95] to derive view maintenance expressions. For example, if view $V = R \bowtie S$, then the maintenance expression calculating the effect of insertions to R ($\triangle R$) is $\triangle V_R = \triangle R \bowtie S$, where $\triangle V_R$ represents the tuples to insert into V as a result of $\triangle R$.

Since in data warehousing environments updates to certain base relations may not occur, or may not change the values of certain attributes, we define each base relation R as having one of three types of updates, depending on how the updateable attributes are used in the view definition:

- If updates to R may change the values of attributes involved in selection conditions (local or join) in the view, then we say R has *exposed updates*.
- Otherwise, if updates to R will not change the values of attributes involved in selection conditions but may change the values of preserved attributes (attributes included in the final projection), then we say R has *protected updates*.
- Otherwise, if updates to R will not change the values of attributes involved in selection conditions or the values of preserved attributes, then we say R has *ignorable updates*.

Ignorable updates cannot have any affect on the view, so they do not need to be propagated. From now on we consider only exposed and protected updates. Exposed updates could cause new tuples to be inserted into the view or tuples to be deleted from the view, so we propagate them as deletions of tuples with the old values followed by insertions of tuples with the new values. For example, given a view $V = \sigma_{R.A=10} R \bowtie S$, if the value of $R.A$ for a tuple in R is changed from 9 to 10 then new tuples could be inserted into V as a result. Protected updates can only change the attribute values of existing tuples in the view; they cannot result in tuples being inserted into or deleted from the view. We therefore propagate protected updates separately. An alternate treatment of updates is considered in Section 4.2.2.

In addition to the usual select, project, and join symbols, we use $\ltimes$ to represent semijoin, $\uplus$ to represent union with bag semantics, and $\dot{-}$ to represent minus with bag semantics. We further assume that project (π) has bag semantics. The notation $\bowtie_X$ represents an equijoin on attribute X, while $\bowtie_{key(R)}$ represents an equijoin on the key attribute of R, assuming this attribute is in both of the joined relations. Insertions to a relation R are represented as $\triangle R$, deletions are represented as $\bigtriangledown R$, and protected updates are represented as μR. Tuples in μR have two attributes corresponding to each of the attributes of R: one containing the value before update and another containing the value after update. We use π^{old} to project the old attribute values and π^{new} to project the new attribute values.

3 Algorithm for Determining Auxiliary Views

We present an algorithm (Algorithm 3.1 below) that, given a view definition V, derives a set of auxiliary views $\mathcal{A}$ such that view V and the views in $\mathcal{A}$ taken together are self-maintainable; *i.e.*, can be maintained upon changes to the base relations without requiring access to any other data. Each auxiliary view $A_{R_i} \in \mathcal{A}$ is an expression of the form:

$$A_{R_i} = (\pi\sigma R_i) \ltimes A_{R_{j_1}} \ltimes A_{R_{j_2}} \ltimes \ldots \ltimes A_{R_{j_n}}$$

That is, each auxiliary view is a selection and a projection on relation R_i followed by zero or more semijoins with other auxiliary views. It can be seen that the number of tuples in each A_{R_i} is never larger than the number of tuples in R_i and, as we have illustrated in Section 1.1, may be much smaller. Auxiliary views of this form can easily be expressed in SQL, and they can be maintained efficiently as will be shown in Section 5. As an additional optimization, *change relevancy tests* [BCL89, LS93] could be used to determine which changes to the base relations are relevant to the auxiliary views, thereby reducing the number of changes sent to the warehouse. Change filtering is not discussed further in this paper.

Intuitively, the first part of the auxiliary view expression, $(\pi\sigma R_i)$, results from pushing down projections and local selection conditions onto R_i. Tuples in R_i that do not pass local selection conditions cannot possibly contribute to tuples in the view; hence they are not needed for view maintenance and therefore need not be stored in A_{R_i} at the warehouse. The semijoins in the second part of the auxiliary view expression further reduce the number of tuples in A_{R_i} by restricting it to contain only those tuples joinable with certain other auxiliary views. In addition, we will show that in some cases the need for A_{R_i} can be eliminated altogether.

We first need to present a few definitions that are used in the algorithm.

Given a view V, let the *join graph* $G(V)$ of a view be a directed graph $\langle \mathcal{R}, \mathcal{E} \rangle$. $\mathcal{R}$ is the set of relations referenced in V, which form the vertices of the graph. There is a directed edge $e(R_i, R_j) \in \mathcal{E}$ from R_i to R_j if V contains a join condition $R_i.B = R_j.A$ and A is a key of R_j. The edge is annotated with *RI* if there is a referential integrity constraint from $R_i.B$ to $R_j.A$.

We assume for now that the graph is a forest (a set of trees). That is, each vertex has at most one edge leading into it and there are no cycles. This assumption still allows us to handle a broad class of views that occur in practice. For example, views involving *chain joins* (a sequence of relations $R_1, \ldots, R_n$ where the join conditions are between a foreign key of R_i and a key of $R_{i+1}, 1 <= i < n$) and *star joins* (one relation R_1, usually large, joined to a set of relations $R_2, \ldots, \mathcal{R}_n$, usually small, where the join conditions are between foreign keys in R_1 and the keys of $R_2, \ldots, R_n$) have tree graphs. In addition, we assume that there are no self-joins. We explain how each of these assumptions can be removed in Appendix B.

The following definition is used to determine the set of relations upon which a relation R_i depends—that is, the set of relations R_j in which (1) a foreign key in R_i is joined to a key of R_j, (2) there is a referential integrity constraint from R_i to R_j, and (3) R_j has protected updates.

$$Dep(R_i, G) = \{R_j \mid \exists e(R_i, R_j) \text{ in } G(V) \text{ annotated with } RI \text{ and } R_j \text{ does not have exposed updates}\}$$

$Dep(R_i, G)$ determines the set of auxiliary views to which R_i is semijoined in the definition of the auxiliary view A_{R_i} for R_i, given above. The reason for the semijoins is as follows. Let R_j be a member of $Dep(R_i, G)$. Due to the referential integrity constraint from R_i to R_j and the fact that the join between R_i and R_j is on a key of R_j, each tuple $t_i \in R_i$ must join with one and only one tuple $t_j \in R_j$. Suppose t_j does not pass the local selection conditions on R_j. Then t_j, and hence t_i, cannot contribute to tuples in the view. Because updates to R_j are protected (by the definition of $Dep(R_i, G)$), t_j, and hence t_i, will never contribute to tuples in the view, so it is not necessary to include t_i in A_{R_i} at the warehouse. It is sufficient to store only those tuples of R_i that pass the local selection conditions on R_i and join with a tuple in R_j that passes the local selection conditions on R_j (*i.e.*, $(\sigma R_i) \ltimes (\sigma R_j)$, where the semijoin condition is the same as the join condition between R_i and R_j in the view). That R_i can be semijoined with A_{R_j}, rather than σR_j, in the definition of A_{R_i} follows from a similar argument applied inductively.

The following definition is used to determine the set of relations upon which relation R_i transitively depends.

$$Dep^+(R_i, G) \text{ is the transitive closure of } Dep(R_i, G)$$

$Dep^+(R_i, G)$ is used to help determine whether it is necessary to store A_{R_i} at the warehouse in order to maintain the view or whether A_{R_i} can be eliminated altogether. Intuitively, if $Dep^+(R_i, G)$ includes all relations referenced in view V except R_i, then A_{R_i} is not needed for propagating insertions to any base relation onto V. The reason is that the key and referential integrity constraints guarantee that new insertions into the other base relations can join only with new insertions into R_i, and not with existing tuples in R_i. This behavior is explained further in Section 4.

The following definition is used to determine the set of relations with which relation R_i needs to join so that the key of one of the joining relations is preserved in the view (where all joins must be

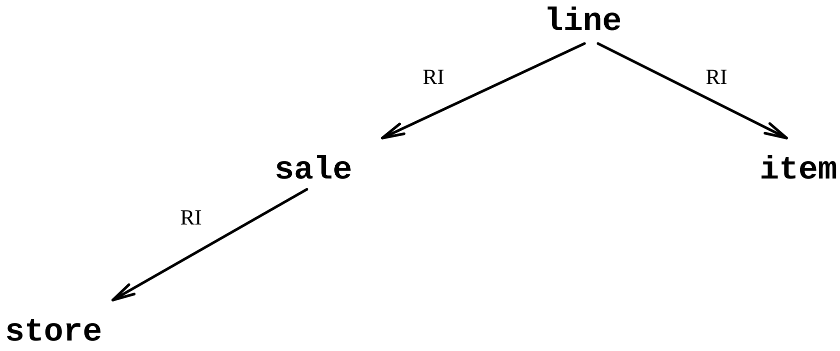

Figure 2: Join Graph G(`cal_toy_sales`)

from keys to foreign keys). If no such relation exists then $Need(R_i, G)$ includes all other relations in the view.

$$Need(R_i, G) = \begin{cases} \phi & \text{if the key of } R_i \text{ is preserved in } V, \\ \{R_j\} \cup Need(R_j, G) & \text{if the key of } R_i \text{ is not preserved in } V \text{ but there is an} \\ & R_j \text{ such that } e(R_j, R_i) \text{ in } G(V), \\ \mathcal{R} - \{R_i\} & \text{otherwise} \end{cases}$$

Note that because we restrict the graph to be a forest, there can be at most one R_j such that $e(R_j, R_i)$ is in $G(V)$. We redefine *Need* when this restriction is removed in Appendix B.

$Need(R_i, G)$ is also used to help determine whether it is necessary to store auxiliary views. In particular, an auxiliary view A_{R_j} is necessary if R_j appears in the *Need* set of some R_i. Intuitively, if the key of R_i is preserved in view V, then deletions and protected updates to R_i can be propagated to V by joining them directly with V on the key of R_i. Otherwise, if the key of R_i is not preserved in V but R_i is joined with another relation R_j on the key of R_i and V preserves the key of R_j, then deletions and protected updates to R_i can be propagated onto V by joining them first with R_j, then joining the result with V. In this case R_j is in the *Need* set of R_i, and hence A_{R_j} is necessary. More generally, if the key of R_i is not present in V but R_i joins with R_j on the key of R_i, then auxiliary views for R_j and each of the relations in $Need(R_j, G)$ are necessary for propagating deletions and protected updates to R_i. Finally, if none of the above conditions hold then auxiliary views for all relations referenced in V other than R_i are necessary.

To illustrate the above definitions we consider again the `cal_toy_sales` view of Section 1.1. Figure 2 shows the graph G(`cal_toy_sales`). The *Dep*, Dep^+, and *Need* functions for each of the base relations are given in Table 2. Assume for now that each base relation has protected updates.

Algorithm 3.1 appears in Figure 3. We will explain how the algorithm works on our running `cal_toy_sales` example. The auxiliary views generated by the algorithm are exactly those given in Figure 1 of Section 1.1. They are shown in relational algebra form in Table 3.

For each relation R_i referenced in the view V, the algorithm checks whether $Dep^+(R_i, G)$ includes every other relation referenced in V and R_i is not in $Need(R_j, G)$ for any relation R_j referenced in V. If so, it is not necessary to store any part of R_i in order to maintain V. Relation `line` is an example where an auxiliary view for the relation is not needed.

$$
\begin{aligned}
Dep(\texttt{store}, G) &= \phi \\
Dep(\texttt{sale}, G) &= \{\texttt{store}\} \\
Dep(\texttt{item}, G) &= \phi \\
Dep(\texttt{line}, G) &= \{\texttt{sale}, \texttt{item}\} \\
\\
Dep^{+}(\texttt{store}, G) &= \phi \\
Dep^{+}(\texttt{sale}, G) &= \{\texttt{store}\} \\
Dep^{+}(\texttt{item}, G) &= \phi \\
Dep^{+}(\texttt{line}, G) &= \{\texttt{sale}, \texttt{item}, \texttt{store}\} \\
\\
Need(\texttt{store}, G) &= \{\texttt{sale}\} \\
Need(\texttt{sale}, G) &= \phi \\
Need(\texttt{item}, G) &= \phi \\
Need(\texttt{line}, G) &= \phi
\end{aligned}
$$

Table 2: *Dep* and *Need* Functions for Base Relations

$$
\begin{aligned}
A_{store} &= \pi_{store\_id,\ manager}\ \sigma_{state=CA}\ \texttt{store} \\
A_{sale} &= (\pi_{sale\_id,\ store\_id,\ month}\ \sigma_{year=1996}\ \texttt{sale}) \ltimes_{store\_id} A_{store} \\
A_{item} &= \pi_{item\_id,\ item\_name}\ \sigma_{category=toy}\ \texttt{item}
\end{aligned}
$$

Table 3: Auxiliary Views for Maintaining the `cal_toy_sales` View

Otherwise, two steps are taken to reduce the amount of data stored in the auxiliary view A_{R_i} for R_i. First, it is possible to push down on R_i local selection conditions (explicit or inferred) in the view so that tuples that don't pass the selection conditions don't need to be stored; it is also possible to project away all attributes from R_i except those that are involved in join conditions, preserved in V, or are a key of R_i. Second, if $Dep(R_i, G)$ is not empty, it is possible to further reduce the tuples stored in A_{R_i} to only those tuples of R_i that join with tuples in other auxiliary views A_{R_k} where R_k is in $Dep(R_i, G)$. The auxiliary view for `sale` is an example where both steps have been applied. A_{sale} is restricted by the semijoin with A_{store} to include only tuples that join with tuples passing the local selection conditions on `store`. The auxiliary views for `store` and `item` are examples where only selection and projection can be applied.

Although view definitions are small and running time is not crucial, we observe that the running time of Algorithm 3.1 is polynomial in the number of relations, and therefore is clearly acceptable. We now state a theorem about the correctness and minimality of the auxiliary views derived by Algorithm 3.1.

Theorem 3.1 *Let V be a view with a tree-structured join graph. The set of auxiliary views $\mathcal{A}$*

Algorithm 3.1
Input
View V.
Output
Set of auxiliary view definitions $\mathcal{A}$.
Method
Let $\mathcal{R}$ be the set of relations referenced in V
Construct graph $G(V)$
for every relation $R_i \in \mathcal{R}$
Construct $Dep(R_i, G)$, $Dep^+(R_i, G)$, and $Need(R_i, G)$
for every relation $R_i \in \mathcal{R}$
if $Dep^+(R_i, G) = \mathcal{R} - \{R_i\}$ and
$\nexists R_j \in \mathcal{R}$ such that $R_i \in Need(R_j, G)$,
then A_{R_i} is not needed
else $A_{R_i} = (\pi_P \sigma_S R_i) \ltimes_{C_1} A_{R_{k_1}} \ltimes_{C_2} A_{R_{k_2}} \ltimes_{C_3} \ldots \ltimes_{C_m} A_{R_{k_m}}$, where
P is the set of attributes in R_i that are preserved in V, appear in join conditions, or are a key of R_i,
S is the strictest set of local selection conditions possible on R_i,
C_l is the join condition $R_i.B = R_{k_l}.A$ with A a key of R_{k_l}, and
$Dep(R_i, G) = \{R_{k_1}, R_{k_2}, \ldots, R_{k_m}\}$
$\diamond$

Figure 3: Algorithm to Derive Auxiliary Views

produced by Algorithm 3.1 is the unique minimal set of views that can be added to V such that $\{V\} \cup \mathcal{A}$ is self-maintainable. $\square$

The proof of Theorem 3.1 is given in Appendix A. By *minimal* we mean that no auxiliary view can be removed from $\mathcal{A}$, and it is not possible to add an additional selection condition or semijoin to further reduce the number of tuples in any auxiliary view and still have $\{V\} \cup \mathcal{A}$ be self-maintainable. We show in Section 4 how V can be maintained using $\mathcal{A}$, and we show in Section 5 how to maintain $\mathcal{A}$ without referencing base relations.

3.1 Effect of Exposed Updates

Recall that so far in our example we have considered protected updates only. Suppose `sale` had exposed updates (i.e., updates could change the values of `year`, `sale_id`, or `store_id`). We note that the definition of *Dep* does not include any relation that has exposed updates. Thus, the *Dep* function for `line` will not include `sale`, and we get:

$$Dep(\texttt{line}, G) = \{\texttt{item}\} \qquad Dep^+(\texttt{line}, G) = \{\texttt{item}\}$$

in which case an auxiliary view for `line` would be created as:

$$A_{line} = \texttt{line} \ltimes_{item\_id} A_{item}$$

No selection or projection can be applied on `line` in A_{line} because there are no local selections on `line` in the view and all attributes of line are either preserved in the view or appear in join conditions. Section 4.2 explains why exposed updates have a different effect on the set of auxiliary views needed than protected updates.

4 Maintaining the View Using the Auxiliary Views

Recall that a view maintenance expression calculates the effects on the view of a certain type of change: insertions, deletions, or updates to a base relation. View maintenance expressions are usually written in terms of the changes and the base relations [GL95,CGL+96]. In this section we show that the set of auxiliary views chosen by Algorithm 3.1 is sufficient to maintain the view by showing how to transform the view maintenance expressions written in terms of the changes and the base relations to equivalent view maintenance expressions written in terms of the changes, the view, and the auxiliary views.

We give view maintenance expressions for each type of change (insertions, deletions, and updates) separately. In addition, for each type of change we apply the changes to each base relation separately by propagating the changes to the base relation onto the view and updating the base relation. The reason we give maintenance expressions of this form, rather than maintenance expressions propagating several types of changes at once, is that maintenance expressions of this form are easier to understand and they are sufficient for our purpose: showing that it is possible to maintain a view using the auxiliary views generated by Algorithm 3.1. View maintenance expressions for insertions are handled in Section 4.2, for deletions are handled in Section 4.3, and for protected updates are handled in Section 4.4. Since exposed updates are handled as deletions followed by insertions, they are treated within Sections 4.2 and 4.3.

4.1 Propagating Multiple Types of Changes

Before giving the view maintenance expressions, we digress briefly to explain the mechanism whereby insertions, deletions, and updates are propagated separately. A subtle point is involved that we feel is not widely understood.

In general, if a sequence $\mathcal{S}$ of intermixed insert, delete, and update operations to a set of base relations is to be propagated to a view V, it is not correct to simply propagate all the deletions, then all the insertions, then all the updates in $\mathcal{S}$ to V. For example, if a tuple t were inserted into and subsequently deleted from a base relation, by propagating insertions after deletions the effects of tuple t would be seen in the view.

In order to correctly propagate all operations of each type of change separately, it is necessary to first transform $\mathcal{S}$ into an equivalent sequence $\mathcal{S}'$ that represents the "net effect" [SP89] of the operations in $\mathcal{S}$. That is, if one operation in $\mathcal{S}$ is followed by another operation on the same tuple, the two operations are replaced by a single operation that represents the net effect of both operations on that tuple. Table 4 gives the transformation rules. (In the table, "update" refers to protected updates; recall that we model exposed updates as "delete t" followed by "insert t'," which is not transformed by the rules in the table.) The rules of the table may need to be applied multiple times. For example, if $\mathcal{S}$ contained three operations: update $t \rightarrow t'$, update $t' \rightarrow t''$, and update $t'' \rightarrow t'''$, then two applications of the last rule in the table would be necessary.

Earlier operation	Later operation	Net operation
delete t	insert t	no operation
insert t	delete t	no operation
insert t	update $t \rightarrow t'$	insert t'
update $t \rightarrow t'$	delete t'	delete t
update $t \rightarrow t'$	update $t' \rightarrow t''$	update $t \rightarrow t''$

Table 4: Capturing Net Effect

The maintenance expressions used in this paper for propagating changes use the contents of the base relations and possibly the view before the changes have been applied. The changes are then applied to the view and the base relations before the next maintenance expression is evaluated. Furthermore, since we propagate exposed updates as deletions followed by insertions, deletions to a base relation must be propagated before insertions to that base relation so that key constraints are not violated during propagation.

4.2 Insertions

In this section we show how the effect on a view of insertions to base relations can be calculated using the auxiliary views chosen by Algorithm 3.1. The view maintenance expression for calculating the effects on an SPJ view V of insertions to a base relation R is obtained by substituting $\triangle R$ (insertions to R) for base relation R in the relational algebra expression for V. For example, the view maintenance expressions calculating the effects on our `cal_toy_sales` view (Section 1.1) of insertions to `store`, `sale`, `line`, and `item` appear in Table 5.
A few words of explanation about the table are in order.

- For convenience, in the table and hereafter we abbreviate `store`, `sale`, `line`, and `item` as St, Sa, L, and I, respectively.
- We abbreviate view `cal_toy_sales` as V.
- We have applied the general rule of "pushing selections and projections down" to the maintenance expressions.
- We use the notation $\triangle V_R$ to represent the insertions into view V due to insertions into base relation R. For example, $\triangle V_{St}$ represents insertions into V due to insertions into St.

$$
\begin{aligned}
\triangle V_{St} &= \pi_{Schema(V)}\ (\pi_{store\_id,manager}\ \sigma_{state=CA}\ \triangle St \\
&\qquad \bowtie_{store\_id}\ \pi_{sale\_id,store\_id,month}\ \sigma_{year=1996}\ Sa \\
&\qquad \bowtie_{sale\_id}\ L \\
&\qquad \bowtie_{item\_id}\ \pi_{item\_id,item\_name}\ \sigma_{category=toy}\ I) \\
\triangle V_{Sa} &= \pi_{Schema(V)}\ (\pi_{store\_id,manager}\ \sigma_{state=CA}\ St \\
&\qquad \bowtie_{store\_id}\ \pi_{sale\_id,store\_id,month}\ \sigma_{year=1996}\ \triangle Sa \\
&\qquad \bowtie_{sale\_id}\ L \\
&\qquad \bowtie_{item\_id}\ \pi_{item\_id,item\_name}\ \sigma_{category=toy}\ I) \\
\triangle V_{L} &= \pi_{Schema(V)}\ (\pi_{store\_id,manager}\ \sigma_{state=CA}\ St \\
&\qquad \bowtie_{store\_id}\ \pi_{sale\_id,store\_id,month}\ \sigma_{year=1996}\ Sa \\
&\qquad \bowtie_{sale\_id}\ \triangle L \\
&\qquad \bowtie_{item\_id}\ \pi_{item\_id,item\_name}\ \sigma_{category=toy}\ I) \\
\triangle V_{I} &= \pi_{Schema(V)}\ (\pi_{store\_id,manager}\ \sigma_{state=CA}\ St \\
&\qquad \bowtie_{store\_id}\ \pi_{sale\_id,store\_id,month}\ \sigma_{year=1996}\ Sa \\
&\qquad \bowtie_{sale\_id}\ L \\
&\qquad \bowtie_{item\_id}\ \pi_{item\_id,item\_name}\ \sigma_{category=toy}\ \triangle I)
\end{aligned}
$$

Table 5: Maintenance Expressions for Insertions

- Each of the maintenance expressions of Table 5 calculates the effect on view V of insertions to one of the base relations. We show in Section 4.2.3 that even if insertions to multiple base relations are propagated at once, the auxiliary views generated by Algorithm 3.1 are still sufficient.

From the expressions of Table 5 it would appear that beyond pushing down selections and projections, nothing can be done to reduce the base relation data required for evaluating the maintenance expressions. If there are no referential integrity constraints, that is indeed the case. However, referential integrity constraints allow certain of the maintenance expressions to be eliminated, requiring less base relation data in the auxiliary views. Maintenance expressions are eliminated due to the following property and corresponding rule.

Property 4.1 (Insertion Property for Foreign Keys) *If there is a referential integrity constraint from $R_j.B$ to $R_i.A$ ($R_j.B$ is the "foreign key"), A is a key of R_i, and R_i does not have exposed updates, then $R_i \bowtie_{\triangle R_i.A=R_j.B} R_j = \phi$. In general, if the above conditions hold then the following is true.*

$$\pi\sigma R_1 \bowtie \ldots \bowtie \pi\sigma\triangle R_i \bowtie_{\triangle R_i.A=R_j.B} \pi\sigma R_j \bowtie \ldots \pi\sigma R_n = \phi$$

$\odot$

Property 4.1 holds because the referential integrity constraint requires that each tuple in R_j join with an existing tuple in R_i, and because it joins on a key of R_i it cannot join with any of the tuples in $\triangle R_i$, so the join of $\triangle R_i$ with R_j must be empty.

Rule 4.1 (Insertion Rule for Foreign Keys) *Let $G(V)$ be the join graph for view V. The maintenance expression calculating the effect on a view V of insertions to a base relation R_i is guaranteed to be empty and thus can be eliminated if there is some relation R_j such that $R_i \in Dep(R_j, G)$.* $\odot$

Rule 4.1 is used to eliminate the maintenance expression that calculates the effect on a view V of insertions to a base relation R_i if there is another relation R_j in V such that $R_i \in Dep(R_j, G)$ where $G(V)$ is the join graph for V. The rule holds because, by the definition of Dep, R_i is in $Dep(R_j, G)$ when the view equates a foreign key of R_j to the key of R_i, there is a referential integrity constraint from the foreign key in R_j to the key of R_i, and R_i has protected updates (the effect of exposed updates is discussed in Section 4.2.2). Since the maintenance expression that calculates the effect of insertions to R_i includes a join between $\triangle R_i$ and R_j, it must be empty by Property 4.1 and therefore can be eliminated. Joins between keys and foreign keys are common in practice; in SQL, referential integrity constraints are always declared between keys and foreign keys, so the conditions of Rule 4.1 are often met.

Being able to eliminate certain maintenance expressions when calculating the effect on a view V of insertions to base relations can significantly reduce the cost of maintaining V. Although view maintenance expressions themselves are not the main theme of this paper, nevertheless this is an important stand-alone result.

4.2.1 Rewriting the Maintenance Expressions to Use Auxiliary Relations

Eliminating certain maintenance expressions using Rule 4.1 allows us to use the auxiliary views instead of base relations when propagating insertions. After applying Rule 4.1, the remaining maintenance expressions are rewritten using the auxiliary views generated by Algorithm 3.1 by replacing each $\pi\sigma R_i$ subexpression with the corresponding auxiliary view A_{R_i} for R_i.

For example, assuming for now that the base relations have protected updates, the maintenance expressions for $\triangle V_{St}$, $\triangle V_{Sa}$, and $\triangle V_I$, in Table 5 can be eliminated by Rule 4.1 due to the referential integrity constraints between `sale.store_id` and `store.store_id`, `line.sale_id` and `sale.sale_id`, and `line.item_id` and `item.item_id`, respectively. Only $\triangle V_L$, the expression calculating the effect of insertions to L, is not guaranteed to be empty. The maintenance expression $\triangle V_L$ is rewritten using auxiliary views as follows. Recall that the auxiliary views are shown in Table 3.

$$\triangle V_L = \pi_{Schema(V)}\ (A_{St} \bowtie_{sale\_id} A_{Sa} \bowtie_{store\_id} \triangle L \bowtie_{item\_id} A_I)$$

Notice that the base relation L is never referenced in the above maintenance expression, so an auxiliary view for L is not needed. In addition, Sa is joined with St in the maintenance expression, which is why it is acceptable to store only the tuples in Sa that join with existing tuples in St—tuples in Sa that don't join with existing tuples in St won't contribute to the result. We show how to handle insertions to Sa and St simultaneously in Section 4.2.3. A proof that the auxiliary views are sufficient in general to evaluate the (reduced) maintenance expressions for insertions appears in Appendix A.

4.2.2 Effect of Exposed Updates

Suppose the view contains a join condition $R_j.B = R_i.A$, A is a key of R_i, there is a referential integrity constraint from $R_j.B$ to $R_i.A$, but R_i has exposed, rather than protected, updates. $Dep(R_j, G)$ thus does not contain R_i. Recall that exposed updates can change the values of attributes involved in selection conditions (local or join). We handle exposed updates as deletions of tuples with the old attribute values followed by insertions of tuples with the new attribute values, since exposed updates may result in deletions or insertions in the view. Thus, if R_i has exposed updates then $\triangle R_i$ may include tuples representing the new values of exposed updates. Because these tuples can join with existing tuples in R_j (without violating the referential integrity or key constraints), Property 4.1 does not hold and Rule 4.1 cannot be used to eliminate the maintenance expression propagating insertions to R_i.

For example, suppose updates may occur to the `year` attribute of Sa. Then an auxiliary view for L would be created as $A_L = L \ltimes_{item\_id} A_I$ as shown in Section 3.1. We cannot semijoin L with A_{Sa} in the auxiliary view for L because new values of updated tuples in Sa could join with existing tuples in L, where the old values of the updated tuples didn't pass the local selection conditions on Sa and hence weren't in A_{Sa}. That is, suppose the year of some sale tuple t was changed from 1995 to 1996. Although the old value of t doesn't pass the selection criteria `year=1996` and therefore wouldn't appear in A_{Sa}, the new value of t would, and since it could join with existing tuples in L we cannot restrict A_L to include only those tuples that join with existing tuples in A_{Sa}.

In this paper we assume that it is known in advance whether each relation of a view V has exposed or protected updates. If a relation has exposed updates, we may need to store more information in the auxiliary views in order to maintain V than if the relation had protected updates. For example, we had to create an auxiliary view for L when Sa had exposed updates, where the auxiliary view for L wasn't needed when Sa had protected updates.

An alternate way to consider updates, which doesn't require advance knowledge of protected versus exposed, is to assume that every base relation has protected updates. Then, before propagating updates, the updates to each base relation are divided into two classes: updates that do not modify attributes involved in selection conditions, and those that do. The first class of updates can be propagated as protected updates using the expressions of Section 4.4. Assuming the second class of updates is relatively small, updates in the second class could be propagated by issuing queries back to the data sources.

4.2.3 Propagating Insertions to Multiple Relations at Once

When using maintenance expressions of the form in Table 5, if we wish to obtain the effect on V of propagating insertions to all four base relations at once, it is incorrect to evaluate the four maintenance expressions and take the union: new view tuples resulting from joining tuples in more than one $\triangle$ relation would not be created. It is instead necessary when using Table 5 to assume that insertions to multiple base relations are propagated to the view one base relation at a time, so that when insertions to base relation R_i are propagated, the insertions to base relations $R_j (j < i)$, have already been applied to the base relations.

Table 6 shows for our example a single maintenance expression propagating insertions to all base relations at once, obtained using the method in [GMS93]. For simplicity we have omitted the selections and projections.

$$\begin{aligned} \triangle V \;=\; & (\triangle St \bowtie Sa \bowtie L \bowtie I) \\ & \uplus ((\triangle St \uplus St) \bowtie \triangle Sa \bowtie L \bowtie I) \\ & \uplus ((\triangle St \uplus St) \bowtie (\triangle Sa \uplus Sa) \bowtie \triangle L \bowtie I) \\ & \uplus ((\triangle St \uplus St) \bowtie (\triangle Sa \uplus Sa) \bowtie \triangle L \bowtie I) \\ & \uplus ((\triangle St \uplus St) \bowtie (\triangle Sa \uplus Sa) \bowtie (\triangle L \uplus L) \bowtie \triangle I) \end{aligned}$$

Table 6: Propagating Insertions to Multiple Base Relations at Once

It is possible to reduce the maintenance expression in Table 6 to the following maintenance expression by eliminating the joins that are guaranteed to be empty by Property 4.1.

$$\triangle V = (((\triangle St \uplus St) \bowtie \triangle Sa) \uplus (St \bowtie Sa)) \bowtie \triangle L \bowtie (\triangle I \uplus I)$$

Just as for the case of propagating insertions one relation at a time, this maintenance expression can be rewritten to use auxiliary views. An auxiliary view for L is not needed, and Sa can be semijoined with A_{St} in the auxiliary view for Sa.

$$\triangle V = (((\triangle St \uplus A_{St}) \bowtie \triangle Sa) \uplus (A_{St} \bowtie A_{Sa})) \bowtie \triangle L \bowtie (\triangle I \uplus A_I)$$

The amount of data needed in the auxiliary views is the same whether insertions are propagated one relation at a time or all at once. The same holds true for propagating deletions and updates to multiple base relations at once, although due to space constraints details are not given in this paper.

4.3 Deletions

In this section we show how the effect on a view of deletions to base relations can be calculated using the auxiliary views. The view maintenance expression for calculating the effects on an SPJ view V of deletions to a base relation R is obtained similarly to the expression for calculating the effects of insertions: we substitute $\bigtriangledown R$ (deletions to R) for base relation R in the relational algebra expression for V. For example, the view maintenance expressions for calculating the effects on our `cal_toy_sales` view of deletions to `store`, `sale`, `line`, and `item` appear respectively as $\bigtriangledown V_{St}$, $\bigtriangledown V_{Sa}$, $\bigtriangledown V_L$, and $\bigtriangledown V_I$ in Table 7. We use the notation $\bigtriangledown V_R$ to represent the deletions from view V due to deletions from base relation R.

Often we can simplify maintenance expressions for deletions to use the contents of the view itself if keys are preserved in the view. We do this using the following properties and rule for deletions in the presence of keys.

$$
\begin{aligned}
\bigtriangledown V_{St} &= \pi_{Schema(V)}\,(\pi_{store\_id,manager}\ \sigma_{state=CA}\ \bigtriangledown St \\
&\qquad \bowtie_{store\_id}\ \pi_{sale\_id,store\_id,month}\ \sigma_{year=1996}\ Sa \\
&\qquad \bowtie_{sale\_id}\ L \\
&\qquad \bowtie_{item\_id}\ \pi_{item\_id,item\_name}\ \sigma_{category=toy}\ I) \\
\bigtriangledown V_{Sa} &= \pi_{Schema(V)}\,(\pi_{store\_id,manager}\ \sigma_{state=CA}\ St \\
&\qquad \bowtie_{store\_id}\ \pi_{sale\_id,store\_id,month}\ \sigma_{year=1996}\ \bigtriangledown Sa \\
&\qquad \bowtie_{sale\_id}\ L \\
&\qquad \bowtie_{item\_id}\ \pi_{item\_id,item\_name}\ \sigma_{category=toy}\ I) \\
\bigtriangledown V_{L} &= \pi_{Schema(V)}\,(\pi_{store\_id,manager}\ \sigma_{state=CA}\ St \\
&\qquad \bowtie_{store\_id}\ \pi_{sale\_id,store\_id,month}\ \sigma_{year=1996}\ Sa \\
&\qquad \bowtie_{sale\_id}\ \bigtriangledown L \\
&\qquad \bowtie_{item\_id}\ \pi_{item\_id,item\_name}\ \sigma_{category=toy}\ I) \\
\bigtriangledown V_{I} &= \pi_{Schema(V)}\,(\pi_{store\_id,manager}\ \sigma_{state=CA}\ St \\
&\qquad \bowtie_{store\_id}\ \pi_{sale\_id,store\_id,month}\ \sigma_{year=1996}\ Sa \\
&\qquad \bowtie_{sale\_id}\ L \\
&\qquad \bowtie_{item\_id}\ \pi_{item\_id,item\_name}\ \sigma_{category=toy}\ \bigtriangledown I)
\end{aligned}
$$

Table 7: Maintenance Expressions for Deletions

Property 4.2 (Deletion Property for Keys) *Given view* $V = \pi_{Schema(V)}(\pi\sigma R_1 \bowtie \ldots \bowtie \pi\sigma R_n)$, *if the key of a relation* R_i *is preserved in* V *then the following equivalence holds:*

$$\pi_{Schema(V)}(\pi\sigma R_1 \bowtie \ldots \bowtie \pi\sigma R_{i-1} \bowtie \pi\sigma \bigtriangledown R_i \bowtie \pi\sigma R_{i+1} \bowtie \ldots \bowtie \pi\sigma R_n) \equiv \pi_{Schema(V)}(V \bowtie_{key(R_i)} \bigtriangledown R_i)$$

$\odot$

Consider the join graph $G(V)$ of view V. Property 4.2 says that if V preserves the key of some relation R_i (i.e., $Need(R_i, G) = \phi$), then we can calculate the effect on V of deletions to R_i by joining V with $\bigtriangledown R_i$ on the key of R_i. The property holds because each tuple in V with the same value for the key of R_i as a tuple t in $\bigtriangledown R_i$ must have been derived from t. Conversely, all tuples in V that were derived from tuple t in $\bigtriangledown R_i$ must have the same value as t for the key of R_i. Therefore, the set of tuples in V that join with t on the key of R_i is exactly the set of tuples in V that should be deleted when t is deleted from R_i. A similar property holds if the key of R_i is not preserved in V, but is equated by a selection condition in V to an attribute C that is preserved in V. In this case the effect of deletions from R_i can be obtained by joining V with $\bigtriangledown R_i$ using the join condition $V.C = key(R_i)$.

Property 4.2 is used in [GJM96] to determine when a view is self-maintainable with respect to deletions from a base relation. We extend their result with Property 4.3.

Property 4.3 (Deletion Property for Key Joins) *Given a view* $V = \pi_{Schema(V)}(\pi\sigma R_1 \bowtie \ldots \bowtie \pi\sigma R_n)$ *satisfying the following conditions:*

1. *V contains join conditions $R_i.A = R_{i+1}.B$, $R_{i+1}.A = R_{i+2}.B$, ..., $R_{i+k-1}.A = R_{i+k}.B$*

2. *attribute A is a key for $R_{i+j}(0 <= j <= k)$, and*

3. *$R_{i+k}.A$ is preserved in V,*

then the following equivalence holds (even without referential integrity constraints):

$$\pi_{Schema(V)}(\pi\sigma R_1 \bowtie \ldots \bowtie \pi\sigma R_{i-1} \bowtie \pi\sigma \nabla R_i \bowtie \pi\sigma R_{i+1} \bowtie \ldots \bowtie \pi\sigma R_n) \equiv$$
$$\pi_{Schema(V)}(V \bowtie_{R_{i+k}.A} \pi\sigma R_{i+k} \bowtie_{R_{i+k}.B=R_{i+k-1}.A} \pi\sigma R_{i+k-1} \bowtie \ldots \bowtie_{R_{i+1}.B=R_i.A} \nabla R_i)$$

$\odot$

Let $G(V)$ be the join graph for view V. Property 4.3 generalizes Property 4.2 to say that if V preserves the key of some relation R_{i+k} and R_i joins to R_{i+k} along keys (that is, $Need(R_i, G) = \{R_{i+1}, \ldots, R_{i+k}\}$ and does not include all the base relations of V), then we can calculate the effect on V of deletions to R_i by joining ∇R_i with the sequence of relations up to R_{i+k} and then joining R_{i+k} with V. The property holds because tuples in V with the same value for the key of R_{i+k} as a tuple t in R_{i+k} must have been derived from t as explained in Property 4.2. Furthermore, since the joins between R_{i+k} and R_i are all along keys, each tuple in R_{i+k} can join with at most one tuple t' in R_i, which means that tuples in V that are derived from tuple t in R_{i+k} must also be derived from tuple t' in R_i. Conversely, if a tuple in V is derived from t' in R_i, then it must have the same value for the key of R_{i+k} as some tuple t in R_{i+k} that t' joins with. Therefore, the set of tuples in V that join on the key of R_{i+k} with some tuple t in R_{i+k} that joins along keys with a tuple t' in R_i is exactly the set of tuples in V that should be deleted when t' is deleted from R_i. As before, a similar property also holds if a key of R_{i+k} is not preserved in V but is equated by a selection condition in V to an attribute C that is preserved in V. In this case the effect of deletions from R_i can be obtained by joining V with R_{i+k} using the join condition $V.C = key(R_{i+k})$.

Rule 4.2 (Deletion Rule) *Let V be a view with a tree structured join graph $G(V)$, and let $Need(R_i, G) = \{R_{i+1}, \ldots, R_{i+k}\}$, where $k \geq 0$. The maintenance expression calculating the effect on a view V of deletions to a base relation R_i may be simplified according to Property 4.3 to reference V unless $Need(R_i, G)$ includes all the base relations of V except R_i.* $\odot$

Rule 4.2 is used to simplify maintenance expressions for deletions to use the contents of the view and fewer base relations. This allows us to rewrite the maintenance expressions for deletions to use the auxiliary views instead of base relations.

4.3.1 Rewriting the Maintenance Expressions to Use Auxiliary Relations

After simplifying the maintenance expressions according to Rule 4.2, the simplified expressions are rewritten to use the auxiliary views generated by Algorithm 3.1 by replacing each $\pi\sigma R_i$ subexpression in the maintenance expression with the corresponding auxiliary view A_{R_i} for R_i. This rewriting is similar to the rewriting for insertions.

Consider again our running example. The maintenance expressions of Table 7 are simplified using Rule 4.2 as follows.

$$
\begin{aligned}
\bigtriangledown V_{St} &= \pi_{Schema(V)} \; (V \bowtie_{sale\_id} \pi_{sale\_id,store\_id} \sigma_{year=1996} Sa \bowtie_{store\_id} \bigtriangledown St) \\
\bigtriangledown V_{Sa} &= \pi_{Schema(V)} \; (V \bowtie_{sale\_id} \bigtriangledown Sa) \\
\bigtriangledown V_{L} &= \pi_{Schema(V)} \; (V \bowtie_{line\_id} \bigtriangledown L) \\
\bigtriangledown V_{I} &= \pi_{Schema(V)} \; (V \bowtie_{item\_id} \bigtriangledown I)
\end{aligned}
$$

The first maintenance expression is rewritten to use the auxiliary view A_{Sa} in place of base relation Sa. The other maintenance expressions do not reference any base relations so they are not rewritten to use auxiliary views.

A proof that the auxiliary views are sufficient in general to evaluate the (simplified) maintenance expressions for deletions appears in Appendix A. For this example note that A_{Sa} is the only auxiliary view referenced in the maintenance expressions for deletions. Because all tuples in Sa that could contribute to tuples in the view are also in A_{Sa}, when calculating the effect on the view of deletions to St it is sufficient to join $\bigtriangledown St$ with tuples in A_{Sa}.

4.4 Protected Updates

In this section we show how the effect on a view of protected updates to base relations can be calculated using the auxiliary views. (Recall that exposed updates are treated separately as deletions followed by insertions.) We give two maintenance expressions for calculating the effect on a view V of protected updates to a base relation R: one returning the tuples to delete from the view (denoted as $\overline{\bigtriangledown} V_R$) and another returning the tuples to insert into the view (denoted as $\underline{\bigtriangleup} V_R$). In practice, these pairs of maintenance expressions usually can be combined into a single SQL update statement.

The view maintenance expression for calculating the tuples to delete from an SPJ view V due to protected updates to a base relation R is obtained by substituting $\pi^{old}\mu R$ (the old attribute values of the updated tuples in R) for base relation R in the relational algebra expression for V. (Recall that μR, π^{old}, and π^{new} were defined in Section 2.) The view maintenance expression for calculating the tuples to insert is obtained similarly by substituting $\pi^{new}\mu R$ (the new attribute values of the updated tuples in R) for base relation R in the relational algebra expression for V. For example, the view maintenance expressions calculating the tuples to delete from our `cal_toy_sales` view due to protected updates to each of the base relations are given in Table 8. Expressions calculating the tuples to insert into the view `cal_toy_sales` are not shown but can be obtained by substituting π^{new} for π^{old} in the expressions of Table 8. Note that the Table 8 expressions are similar to the deletion expressions of Table 7.

We simplify the maintenance expressions for protected updates similarly to the way we simplify the maintenance expressions for deletions, by using the contents of the view itself if keys are preserved in the view. We give the following properties and rule for updates in the presence of preserved keys. In the following let $P(\mu R_i) = (Schema(\mu R_i) \cap Schema(V)) \cup Schema(V)$. We use $\pi^{old}_{P(\mu R_i)}$ and $\pi^{new}_{P(\mu R_i)}$ to project the old and new attribute values respectively of preserved attributes

$$
\begin{aligned}
\nabla V_{St} &= \pi_{Schema(V)}\,(\pi^{old}_{store\_id,manager}\ \sigma_{state=CA}\ \mu St \\
&\qquad \bowtie_{store\_id}\ \pi_{sale\_id,store\_id,month}\ \sigma_{year=1996}\ Sa \\
&\qquad \bowtie_{sale\_id}\ L \\
&\qquad \bowtie_{item\_id}\ \pi_{item\_id,item\_name}\ \sigma_{category=toy}\ I) \\
\nabla V_{Sa} &= \pi_{Schema(V)}\,(\pi_{store\_id,manager}\ \sigma_{state=CA}\ St \\
&\qquad \bowtie_{store\_id}\ \pi^{old}_{sale\_id,store\_id,month}\ \sigma_{year=1996}\ \mu Sa \\
&\qquad \bowtie_{sale\_id}\ L \\
&\qquad \bowtie_{item\_id}\ \pi_{item\_id,item\_name}\ \sigma_{category=toy}\ I) \\
\nabla V_{L} &= \pi_{Schema(V)}\,(\pi_{store\_id,manager}\ \sigma_{state=CA}\ St \\
&\qquad \bowtie_{store\_id}\ \pi_{sale\_id,store\_id,month}\ \sigma_{year=1996}\ Sa \\
&\qquad \bowtie_{sale\_id}\ \pi^{old}_{line\_id,sale\_id,item\_id,sales\_price}\ \mu L \\
&\qquad \bowtie_{item\_id}\ \pi_{item\_id,item\_name}\ \sigma_{category=toy}\ I) \\
\nabla V_{I} &= \pi_{Schema(V)}\,(\pi_{store\_id,manager}\ \sigma_{state=CA}\ St \\
&\qquad \bowtie_{store\_id}\ \pi_{sale\_id,store\_id,month}\ \sigma_{year=1996}\ Sa \\
&\qquad \bowtie_{sale\_id}\ L \\
&\qquad \bowtie_{item\_id}\ \pi^{old}_{item\_id,item\_name}\ \sigma_{category=toy}\ \mu I)
\end{aligned}
$$

Table 8: Maintenance Expressions for Removing Old Updates

in μR_i and the (regular) attribute values for preserved attributes of other relations in V. We use $\bowtie oldkey(R_i)$ to denote joining on the attribute in which the key value before the update is held.

Property 4.4 (Protected Update Property for Keys) *Given a view $V = \pi_{Schema(V)}(\pi\sigma R_1 \bowtie \ldots \bowtie \pi\sigma R_n)$ where the key of a relation R_i is preserved in V, then the following equivalences hold:*

$$
\pi_{Schema(V)}(\pi\sigma R_1 \bowtie \ldots \bowtie \pi\sigma R_{i-1} \bowtie \pi^{old}\sigma\mu R_i \bowtie \pi\sigma R_{i+1} \bowtie \ldots \bowtie \pi\sigma R_n) \equiv \pi^{old}_{P(\mu R_i)}(V \bowtie_{oldkey(R_i)} \mu R_i)
$$
$$
\pi_{Schema(V)}(\pi\sigma R_1 \bowtie \ldots \bowtie \pi\sigma R_{i-1} \bowtie \pi^{new}\sigma\mu R_i \bowtie \pi\sigma R_{i+1} \bowtie \ldots \bowtie \pi\sigma R_n) \equiv \pi^{new}_{P(\mu R_i)}(V \bowtie_{oldkey(R_i)} \mu R_i)
$$

$\odot$

Property 4.5 (Protected Update Property for Key Joins) *Given a view $V = \pi_{Schema(V)}(\pi\sigma R_1 \bowtie \ldots \bowtie \pi\sigma R_n)$ satisfying the following conditions:*

1. *view V contains join conditions $R_i.A = R_{i+1}.B, R_{i+1}.A = R_{i+2}.B, \ldots, R_{i+k-1}.A = R_{i+k}.B$*
2. *attribute A is a key for $R_{i+j} (0 <= j <= k)$, and*
3. *$R_{i+k}.A$ is preserved in V,*

then the following equivalences hold (even without referential integrity constraints):

$$\pi_{Schema(V)}(\pi\sigma R_1 \bowtie \ldots \bowtie \pi\sigma R_{i-1} \bowtie \pi^{old}\sigma\mu R_i \bowtie \pi\sigma R_{i+1} \bowtie \ldots \bowtie \pi\sigma R_n) \equiv$$
$$\pi^{old}_{P(\mu R_i)}(V \bowtie_{R_{i+k}.A} R_{i+k} \bowtie_{R_{i+k}.B=R_{i+k-1}.A} R_{i+k-1} \bowtie \ldots \bowtie_{R_{i+1}.B=R_i.A} \mu R_i)$$
$$\pi_{Schema(V)}(\pi\sigma R_1 \bowtie \ldots \bowtie \pi\sigma R_{i-1} \bowtie \pi^{new}\sigma\mu R_i \bowtie \pi\sigma R_{i+1} \bowtie \ldots \bowtie \pi\sigma R_n) \equiv$$
$$\pi^{new}_{P(\mu R_i)}(V \bowtie_{R_{i+k}.A} R_{i+k} \bowtie_{R_{i+k}.B=R_{i+k-1}.A} R_{i+k-1} \bowtie \ldots \bowtie_{R_{i+1}.B=R_i.A} \mu R_i)$$

$\odot$

Properties 4.4 and 4.5 are similar to the corresponding properties for deletions. Attributes of R_i that are involved in selection conditions are guaranteed not to be updated, so it does not matter whether we test the old or new value in selection conditions. Property 4.4 is used in [GJM96] to determine when a view is self-maintainable for base relation updates.

Consider the join graph $G(V)$ of view V. Property 4.5 generalizes Property 4.4; Property 4.5 says that if V preserves the key of some relation R_{i+k} and R_i joins to R_{i+k} along keys (that is, $Need(R_i, G) = \{R_{i+1}, \ldots, R_{i+k}\}$ and does not include all the base relations of V), then we can calculate the effect on V of protected updates to R_i by joining μR_i with the sequence of relations up to R_{i+k} and then joining R_{i+k} with V. As for deletions, a similar property also holds if a key of R_{i+k} is not preserved in V but is equated by a selection condition in V to an attribute C that is preserved in V. In this case the effect of updates to R_i can be obtained by joining V with R_{i+k} using the join condition $V.C = key(R_{i+k})$.

Rule 4.3 (Protected Update Rule) *Let V be a view with a tree structured join graph $G(V)$, and let $Need(R_i, G) = \{R_{i+1}, \ldots, R_{i+k}\}$, where $k \geq 0$. The maintenance expressions calculating the effect on a view V of protected updates to a base relation R_i can be simplified according to Property 4.5 to reference V unless $Need(R_i, G)$ includes all the base relations of V except R_i.* $\odot$

Similar to the rule for deletions, Rule 4.3 is used to simplify the maintenance expressions for ∇V_R and $\triangle V_R$ to use the contents of the view and fewer base relations so that the maintenance expressions can be rewritten in terms of the auxiliary views.

4.4.1 Rewriting the Maintenance Expressions to Use Auxiliary Relations

After simplifying the maintenance expressions according to Rule 4.3, the simplified expressions are rewritten to use the auxiliary views generated by Algorithm 3.1 by replacing each $\pi\sigma R_i$ subexpression in the maintenance expression with the corresponding auxiliary view A_{R_i} for R_i. The rewriting is similar to the rewriting for insertions and deletions.

Consider again our running example. The maintenance expressions of Table 8 are simplified using Rule 4.3 as follows.

$$\begin{aligned}
\nabla V_{St} &= \pi^{old}_{P(\mu St)}\ (V \bowtie_{sale_id} Sa \bowtie_{store_id} \mu St) \\
\nabla V_{Sa} &= \pi^{old}_{P(\mu Sa)}\ (V \bowtie_{oldsale_id} \mu Sa) \\
\nabla V_{L} &= \pi^{old}_{P(\mu L)}\ (V \bowtie_{oldline_id} \mu L) \\
\nabla V_{I} &= \pi^{old}_{P(\mu I)}\ (V \bowtie_{olditem_id} \mu I)
\end{aligned}$$

The maintenance expressions $\blacktriangle V_{St}$, $\blacktriangle V_{Sa}$, $\blacktriangle V_L$, and $\blacktriangle V_I$ are obtained by substituting π^{new} for π^{old} in the above expressions.[1] The maintenance expressions $\blacktriangledown V_{St}$ and $\blacktriangle V_{St}$ are rewritten to use the auxiliary view A_{Sa} in place of base relation Sa. No other maintenance expressions reference base relations. The rewritten expressions using the auxiliary views are equivalent to the expressions using the base relations for the same reason that rewriting maintenance expressions for deletions to use auxiliary views preserves equivalence. Again, a proof that the auxiliary views are sufficient in general to evaluate the maintenance expressions is given in Appendix A.

5 Maintaining Auxiliary Views

This section shows that the set of auxiliary views is itself self-maintainable. We begin with an intuitive argument for this claim based on join graphs. Recall that the auxiliary views derived by Algorithm 3.1 are of the form:

$$A_R = (\pi_{Schema(A_R)} \sigma_S R) \ltimes_{C_1} A_{R_1} \ltimes_{C_2} A_{R_2} \ltimes_{C_3} \ldots \ltimes_{C_m} A_{R_m}$$

where C_i equijoins a foreign key of R with the corresponding key for relation R_i. Because the joins are along foreign key referential integrity constraints, each semijoin could be replaced by a join. Thus, each auxiliary view is an SPJ view, and its join graph can be constructed as discussed in Section 3. Further, note that the join graph for each auxiliary view is a subgraph of the graph for the original view, because each join in an auxiliary view is also a join in the original view. Thus, the information needed to maintain the original view is also sufficient to maintain each of its auxiliary views.

Below we give the maintenance expressions for the auxiliary views. The auxiliary views are a special class of SPJ views where all joins are across foreign key constraints. Thus, their maintenance expressions are simpler than the expressions for general views described in Section 4. Because of the simple structure of auxiliary views, they are extremely efficient to maintain. In fact, maintenance expressions for propagating deletions and protected updates to auxiliary views involve only a single join, independent of the number of joins in the auxiliary view. For propagating insertions to auxiliary view A_R, only one maintenance expression is needed, because only the insertions to R affect the auxiliary view.

5.1 Insertions

For the auxiliary view A_R as given above, the maintenance expression for insertions is is:

$$\triangle A_R = \pi_{Schema(A_R)}((\sigma_S \triangle R) \bowtie_{C_1} A_{R_1} \bowtie_{C_2} A_{R_2} \bowtie_{C_3} \ldots \bowtie_{C_m} A_{R_m})$$

Insertions into any of the auxiliary views used to define A_R do not affect $\triangle A_R$ due to Rule 4.1: a tuple inserted into any auxiliary view A_{R_i} will not join with any tuple in relation R because of the referential integrity constraint from R to R_i. Thus, insertions to A_{R_i} need not be propagated to A_R.

[1]Note that in the $\blacktriangle$ maintenance expressions we still join on the old attribute values of the keys. For example, if the key of L were updated we would still join V with μL on the old value of the key of L in $\blacktriangle V_L$.

5.2 Deletions

For the auxiliary view A_R as given above, the maintenance expression for deletions is:

$$\begin{array}{rll} \triangledown A_R = & \pi_{Schema(A_R)}(& A_R \ltimes_{key(R)} (\sigma_S \triangledown R) \\ \cup & & A_R \ltimes_{C_1} \triangledown A_{R_1} \\ \cup & & \ldots \\ \cup & & A_R \ltimes_{C_m} \triangledown A_{R_m}) \end{array}$$

Deletions from auxiliary view A_{R_i} can be joined directly with A_R because A_R contains the keys of each A_{R_i}.

5.3 Protected Updates

We calculate the effect on A_R of protected updates to base relation R by giving two maintenance expressions: one returning the tuples to delete from A_R and another returning the tuples to insert into A_R. Protected updates to the auxiliary views $A_{R_1}, \ldots, A_{R_m}$ do not affect A_R because A_R includes only attributes from relation R. The maintenance expression for updates is:

$$\begin{array}{rcl} \triangledown\!\!\!\!\!\triangledown A_R & = & \pi^{old}_{Schema(A_R)}(A_R \ltimes_{old\ key(R)} \mu R) \\ \triangle\!\!\!\!\triangle A_R & = & \pi^{new}_{Schema(A_R)}(A_R \ltimes_{old\ key(R)} \mu R) \end{array}$$

In practice this pair of expressions can be combined into a single SQL update statement.

5.4 Evaluating the Maintenance Expressions

The maintenance expressions for an auxiliary view A_R rely upon auxiliary views $A_{R_1}, \ldots, A_{R_m}$. Thus, to compute changes to A_R, we first need to compute changes to $A_{R_1}, \ldots, A_{R_m}$. We can order the computation of auxiliary view maintenance expressions because the auxiliary view definitions are non-recursive. That is, an auxiliary view will not (directly or indirectly) depend on itself. More formally, consider a graph that has a node for each auxiliary view and an edge from the node for view A to the node for view B if A occurs in the definition of B; this graph is guaranteed to be acyclic: A_R uses A_{R_i} only if relation R_i is in the *Dep* set of R, and a relation cannot be in the closure of its own *Dep* set for tree-structured join graphs. Thus, auxiliary views can be maintained in a bottom-up fashion, starting with views that depend on no other auxiliary views, and working up to the final original view.

6 Related Work, Summary, and Future Directions

The problem of view self-maintainability was considered initially in [BCL89, GJM96]. For each modification type (insertions, deletions, and updates), they identify subclasses of SPJ views that can be maintained using only the view and the modification. [BCL89] states necessary and sufficient conditions on the view definition for the view to be self-maintainable for updates specified using a particular SQL modification statement (*e.g.*, delete all tuples where $R.A > 3$). [GJM96] uses information about key attributes to determine self-maintainability of a view with respect to all modifications of a certain type.

In this paper we consider the problem of making a view self-maintainable by materializing a set of auxiliary views such that the original view and the auxiliary views taken together are self-maintainable. Although the set of base relations over which a view is defined forms one such set of auxiliary views, our approach is to derive auxiliary views that are much smaller than storing the base relations in their entirety. Identifying a set of small auxiliary views to make another view self-maintainable is an important problem in data warehousing, where the base relations may not be readily available.

In [HZ96], views are made self-maintainable by pushing down selections and projections to the base relations and storing the results at the warehouse. Thus, using our terminology, they consider auxiliary views based only on select and project operators. We improve upon their approach by considering auxiliary views based on select, project, and semijoin operators, along with using knowledge about key and referential integrity constraints. We have shown in Section 1.1 that our approach can significantly reduce the sizes of the auxiliary views. We also have shown (Section 5) that auxiliary views of the form our algorithm produces can be (self-)maintained efficiently.

In [TSI94], inclusion dependencies (similar to referential integrity constraints) are used to determine when it is possible to answer from a view joining several relations, a query over a subset of the relations; *e.g.*, given V is a view joining R and other relations, when $\pi_{Schema(R)}V \equiv R$. We on the other hand, use similar referential integrity constraints to simplify view maintenance expressions.

As part of our work, we have shown that by using key and referential integrity constraints, maintenance expressions calculating the effect on a view of insertions to certain base relations are guaranteed to be empty. Thus, these maintenance expressions do not have to be evaluated, reducing the overall cost of maintaining the view. To the best of our knowledge, no previous work considers using key and referential integrity constraints to reduce the sizes of auxiliary relations required to maintain a view, or even to reduce the cost of view maintenance in a traditional setting.

In the future we plan to extend our approach along three directions:

- Extend the class of views considered to include aggregation.
- Extend the class of auxiliary views considered to use join and anti-semijoin operators.
- Rather than restricting ourselves to a single view, consider the problem of making a set of views self-maintainable. Considering a set of views together can produce opportunities for "sharing" among views and auxiliary views.

References

[BCL89] J. Blakeley, N. Coburn, and P. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, September 1989.

[CGL+96] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In SIGMOD 1996.

[DEB95] *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2), June 1995.

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

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In SIGMOD 1995.

[GM95] A. Gupta and I. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing* [DEB95], pages 3–19.

[GM96] A. Gupta and I. Mumick, editors. *Materialized Views*. MIT Press, Cambridge, MA, 1996. To be published.

[GMS93] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *SIGMOD* 1993.

[HGW+95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. In *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing* [DEB95], pages 41–48.

[HZ96] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In SIGMOD 1996.

[LS93] A. Levy and Y. Sagiv. Queries independent of updates. In VLDB 1993.

[Mum95] I. Mumick. The Rejuvenation of Materialized Views. In *Proceedings of the Sixth International Conference on Information Systems and Management of Data (CISMOD)*, Bombay, India, November 1995.

[SP89] A. Segev and J. Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.

[TSI94] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A versatile tool for physical data independence. In Jorge Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the 20^{th} International Conference on Very Large Databases*, pages 367–378, Santiago, Chile, September 12-15 1994.

[ZGHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In SIGMOD 1995.

A Informal Proof of Theorem 3.1

Proof: We show that for view V with join graph $G(V)$ and the set of corresponding auxiliary views $\mathcal{A}$ derived by Algorithm 3.1,

1. $\mathcal{A}$ is sufficient to maintain V,
2. $\mathcal{A}$ is self-maintainable, and
3. $\mathcal{A}$ is the unique minimal set of views such that the above two conditions hold.

By minimal we mean that no element $A_R \in \mathcal{A}$ can be removed, and no additional selection conditions or semijoins can be applied to A_R to further reduce the number of tuples in A_R. The above claims hold when the auxiliary views considered are of the form:

$$A_{R_i} = (\pi\sigma R_i) \ltimes A_{R_{j_1}} \ltimes A_{R_{j_2}} \ltimes \ldots \ltimes A_{R_{j_n}}$$

First we show that $\mathcal{A}$ is sufficient to maintain V.

1. We show that the maintenance expressions propagating insertions to the base relations onto V rewritten by substituting for each base relation R its corresponding auxiliary view $A_R \in \mathcal{A}$ are equivalent to the maintenance expressions using the base relations.
   - If A_R is not created then R is the root of $G(V)$, and $Dep^+(R, G)$ contains every relation referenced in V other than R. Therefore, every maintenance expression that calculates the effect on V of insertions to a base relation other than R will be eliminated by Rule 4.1. The reason is that for every other relation S, there is a relation S' such that $S \in Dep(S', G)$. Thus, the maintenance expression that calculates the effect on V of insertions to S will be eliminated by Rule 4.1. Hence the only maintenance expression remaining will be the one that calculates the effect on V of insertions to R, and R is not referenced in that maintenance expression ($\triangle R$ is referenced).
   - If A_R is of the form $A_R = (\pi\sigma R) \ltimes A_{R_1} \ltimes A_{R_2} \ltimes \ldots \ltimes A_{R_k}$, then we need to show that the only tuples of R that could contribute to the maintenance expressions are those that are joinable with $A_{R_1}, A_{R_2}, \ldots, A_{R_k}$. We show this inductively. Let the *level* of a relation S be defined as the maximum path length from S to a leaf node in $G(V)$. For level 0, A_R is of the form $\pi\sigma R$, then A_R contains all tuples of R that could contribute to any maintenance expression. For level m+1, We illustrate the claim using only A_{R_1} out of $A_{R_1}, A_{R_2}, \ldots, A_{R_k}$, where each R_i is in level at most m. $R_1 \in Dep(R, G)$, so the maintenance expression that calculates the effect on V of insertions to R_1 will be eliminated by Rule 4.1. Thus, whenever R appears in a maintenance expression M it is joined with R_1. Since by induction A_{R_1} contains all tuples of R_1 that could contribute to any maintenance expression, only tuples in R that are joinable with tuples in A_{R_1} could contribute to M.
2. We now show that the maintenance expressions propagating deletions to the base relations onto V rewritten by substituting for each base relation R its corresponding auxiliary view $A_R \in \mathcal{A}$ are equivalent to the maintenance expressions using the base relations.
   - If A_R is not created then $\nexists S$ such that $R \in Need(S, G)$. Therefore, after rewriting the maintenance expressions according Rule 4.2, R will not appear in any maintenance expression.
   - Consider A_R of the form $A_R = (\pi\sigma R) \ltimes A_{R_1} \ltimes A_{R_2} \ltimes \ldots \ltimes A_{R_k}$, and let A_R appear in the maintenance expression M calculating the effect onto V of deletions to some base relation S. The only tuples in R that join with some tuple in S and could contribute to the view are those in A_R. Since $\bigtriangledown S \subseteq S$, all the tuples in R that join with some tuple in $\bigtriangledown S$ and could contribute to M are in A_R.

3. The rules for simplifying maintenance expressions for updates are similar to the rules for simplifying maintenance expressions for deletes. Therefore, the argument showing that the maintenance expressions propagating updates to the base relations onto V rewritten by substituting for each base relation R its corresponding auxiliary view $A_R \in \mathcal{A}$ is similar to the argument for deletions.

Next we show that $\mathcal{A}$ is self-maintainable.

A_R of the form $A_R = (\pi\sigma R) \ltimes A_{R_1} \ltimes A_{R_2} \ltimes \ldots \ltimes A_{R_k}$,

Consider again $A_R = (\pi_{Schema(A_R)} \sigma_S R) \ltimes_{C_1} A_{R_1} \ltimes_{C_2} A_{R_2} \ltimes_{C_3} \ldots \ltimes_{C_m} A_{R_m}$. Because the joins are along foreign key referential integrity constraints, each semijoin could be replaced by a join. Thus, each auxiliary view is an SPJ view, and its join graph can be constructed as discussed in Section 3. Further, note that the join graph for each auxiliary view is a subtree of the graph for the original view, because each join in an auxiliary view is also a join in the original view. Thus, every auxiliary view generated by Algorithm 3 for maintaining A_R is also in the set $\mathcal{A}$ for maintaining the original view.

Finally we show that $\mathcal{A}$ is a unique minimal set.

Every auxiliary view $A_R \in \mathcal{A}$ is necessary. Suppose $Dep^+(R, G) \neq \mathcal{R} - \{R\}$ (R is not the root of $G(V)$). Then A_R appears in the maintenance expression calculating the effect onto V of insertions to some other relation S. If $Dep^+(R, G) = \mathcal{R} - \{R\}$ (R is the root of $G(V)$), then A_R is only created if $\exists S$ such that $R \in Need(S)$, in which case A_R appears in the maintenance expression calculating the effect onto V of deletions to S.

For each auxiliary view $A_R \in \mathcal{A}$, we cannot further reduce the number of tuples in A_R by adding additional selections or semijoins. We cannot add additional selections because by definition all possible selection conditions are applied to R in A_R. There are two cases to consider for why we cannot add a semijoin with an auxiliary expression A_S.

1. If $S \in Dep(R, G)$, A_R already includes a semijoin with A_S.
2. If $S \notin Dep(R, G)$, then tuples in $\triangle S$ could join with tuples in R that are not in $R \ltimes S$, because Property 4.1 does not hold. Therefore, the maintenance expression calculating the effect onto V of insertions to S could potentially miss insertions into V.

∎

B General Join Graphs

In Section 3 we restricted our views to have join graphs that are forests of trees, as a simplification for deriving the auxiliary views. Here we discuss the impact of allowing general join graphs. We also consider self-joins.

First consider an arbitrary join graph for a view without self-joins. We define the *RI projection* of a join graph $G(V)$ to be the graph obtained by removing from $G(V)$ all edges that are not labeled RI. In other words, we remove edges due to joins that do not imply a referential integrity constraint.

B.1 Acyclic RI Projections

We first consider the case where the RI projection is acyclic. We compute the $Dep^+(R_i, G)$ sets as discussed in Section 3. $Need(R_i, G)$ sets are computed in three phases: First, we initialize the need sets using the following definition:

$$Need(R_i, G) = \begin{cases} \phi & \text{if the key of } R_i \text{ is preserved in } V, \\ \{\{R_j\} \cup Need(R_j, G)\} & \text{if the key of } R_i \text{ is not preserved in } V \text{ and there is an } R_j \\ & \text{such that } e(R_j, R_i) \text{ in } G(V)\text{, and } e(R_j, R_i) \text{ is not part} \\ & \text{of a cycle} \\ undefined & \text{otherwise} \end{cases}$$

We note that multiple need sets are possible since a node R_i can now have multiple parents. Second, we consider the nodes on each cycle. If all the nodes on the cycle have undefined need sets, we set $Need(R_i) = \{\mathcal{R}\}$, that is the set of all relations except relation R_i. However, if some node R_i on the cycle has a defined need set, we add to the need sets of other nodes R_k on the cycle the sets:

$$\{\{R_j\} \cup Need(R_j, G)\}$$

provided the key of R_k is not preserved in V and there is an R_j such that $e(R_j, R_i)$ is part of the cycle under consideration. If a cycle is contained wholly within another cycle, the containing cycle should be considered first.

Thirdly, if a relation R_i has multiple need sets, we will remove any need sets containing the unique root of the graph (if there is a unique root).

After these steps, we will use Algorithm 3 to compute the set of auxiliary views. We now reason why there exists a minimal set of auxiliary views, and why Algorithm 3 adapted as above computes this set.

Consider a connected component of the full join graph. The graph can be seen as a generalization of a tree where (1) there can be more than one root, and (2) an internal node can have more than one parent. Having more than one root does not impact either the Algorithm 3 to compute the auxiliary views (except that the condition $(Dep^+(R_i, G) = \mathcal{R} - \{R_i\})$ cannot be true for any relation R_i), or the expressions that maintain the view using the auxiliary views. When a node R has multiple parents, it can have multiple need sets, one through each parent. The need sets determine which relations can be joined with the deleted or updated tuples of R to determine deleted or updated tuples in the view. When there are multiple need sets, and if more than one is available in an auxiliary expression, there will be multiple maintenance expressions for deletions and updates to relation R, and one then has to choose between them. However, it is interesting that even in presence of multiple need sets, there is still a unique minimal set of auxiliary views. To see why this holds, note that the only application of the need sets on choosing auxiliary views is that if the graph has a single root then an auxiliary view may not be needed for the root unless it is in some relation's Need set. Therefore, for graphs with a single root, if a relation has multiple need sets one of which contains the root, we should remove the need set containing the root. Algorithm 3 will then choose the minimal set of auxiliary views.

Need sets are used to simplify the maintenance expressions for deletions and updates. Since we have multiple need sets, we can chose any one of these to simplify the expressions by Rules 4.2 and 4.3.

B.2 Cyclic RI Projections

We now consider nonrecursive views whose RI projection is cyclic. We construct a reduced join graph by coalescing all strongly connected components of the RI projection into single nodes, and define an intermediate view representing the join of all relations in the cycle, with a projection that preserves the keys of the base relations.

The new view definition, in terms of the intermediate views, has an acyclic join graph, and is solved as discussed in Sections B.1. The solution tell us whether an auxiliary view corresponding to the intermediate view needs to be materialized. If one needs to be materialized, then we have two choices:

- Materialize a select-project expression of each relation in the cycle, restricted by semi-joins defined on the intermediate view.
- Materialize the intermediate view. (This is one case where we deviate from our standard form of auxiliary expressions.)

Note that for the second option, we do not need to materialize any other auxiliary view. The cycle of RI joins implies that new insertions into any relation in the cyclic join will not cause any change in the result of the join.

B.3 Self-joins

Let a view be defined by joining multiple (for simplicity, say two) occurrences of relation T. Label each occurrence with a unique name T_a and T_b, and run Algorithm 3 on the resulting join graph G to determine the auxiliary views A_{T_a} and A_{T_b}, if any.

We have the following scenarios:

- One or both of the auxiliary views A_{T_a} and A_{T_b} are not needed, or the two are identical: In this case materialize the auxiliary view that is needed.
- The auxiliary views A_{T_a} and A_{T_b} are both needed and they are not equal: In this case we have two options:
  - Materialize a select-project expression of relation T. The selection condition is the disjunction of the selection conditions in T_a and T_b, and the projection list is the union of the projection lists of T_a and T_b.
  - Materialize both the auxiliary views A_{T_a} and A_{T_b}.

  In other words we have two minimal sets of auxiliary views.