Incremental Maintenance of Aggregate and Outerjoin Expressions

Himanshu Gupta*
SUNY, Stony Brook
hgupta@cs.sunysb.edu

Inderpal Singh Mumick
Kirusa Inc.
mumick@mumick.com

Abstract

Views stored in a data warehouse need to be kept current. As recomputing the views is very expensive, incremental maintenance algorithms are required. Over recent years, several incremental maintenance algorithms have been proposed. None of the proposed algorithms handle the general case of relational expressions involving aggregate and outerjoin operators efficiently.

In this article, we develop the change-table technique for incrementally maintaining general view expressions involving relational and aggregate operators. We show that the change-table technique outperforms the previously proposed techniques by orders of magnitude. The developed framework easily extends to efficiently maintaining view expressions containing outerjoin operators. We prove that the developed change-table technique is an optimal incremental maintenance scheme for a given view expression tree under some reasonable assumptions.

1 Introduction

In a data warehouse, views are computed and stored in the database to allow efficient querying and analysis of the data. These views stored at the data warehouse are known as *materialized views*. In order to keep the views in the data warehouse up to date, it is necessary to maintain the materialized views in response to the changes at the sources. The view can be either recomputed from scratch, or *incrementally maintained* by propagating the base data changes onto the view so that the view reflects the changes. Incrementally maintaining a view can be significantly cheaper than recomputing the view from scratch, especially if the size of the view is large compared to the size of the changes [BM90, MQM97, CKL$^+$97].

The problem of finding such changes at the views based on changes to the base relations has come to be known as the *view maintenance problem* and has been studied extensively. Several algorithms have been proposed over the recent years [BLT86, BCL89, CW91, QW91, GMS93, GL95, Qua97, MQM97, GJM97] for incremental maintenance of view expressions. The previously proposed algorithms on incremental maintenance suffer from the following shortcomings:

- None of the earlier work handles the case of general view expressions involving aggregate and outerjoin operators. Quass in [Qua97] is the only work that attempts to maintain general view expressions involving aggregate operators, but the expressions obtained are very inefficient and complicated. Most of the other work [MQM97, PSCP02, GMS93] is limited to view expressions having only one aggregation operator

*Contact Author. Department of Computer Science, State University of New York, Stony Brook NY 11794.

as the last operator of the expression tree. Gupta et al. in [GJM97] show how to maintain a simple outerjoin view, but do not address general expressions involving outerjoin operators.[1]

- To date, most of the incremental maintenance approaches compute and propagate insertions and deletions at each node in a view expression tree, which could be very inefficient in view expressions that involve aggregation or outerjoin operators.

Our Contributions. In this article, we develop the change-table technique for incremental maintenance of general view expressions involving aggregate and outerjoin operators. Change table of a particular view is applied to the view using a special refresh operator. We develop techniques for computation and propagation of change tables through various operators in a given view expression, in response to changes at the base relations. In contrast to the previously developed techniques which propagate data in terms of insertions and deletions through a view expression, our developed change-table technique propagates data (in terms of change-tables) as well as action (in terms of parameters of the refresh operation) through the given view expression. We show that the developed change-table framework yields very efficient incremental maintenance expressions for general view expressions.

Paper Organization. In the rest of this section, we present some basic notation used throughout this article. Section 2 presents a motivating example that illustrates the idea behind this paper and contrasts previous techniques with the change-table technique developed in this article. The example shows that the change-table technique outperforms the previously proposed techniques by orders of magnitude. In Section 3, we briefly describe how our work fits in the previous frameworks of incremental view maintenance algorithms. In Section 4, we define the refresh operator used to apply the changes represented as a change table and briefly outline its implementation. In the following section, we discuss propagation of change tables that originate at an aggregate operator. Section 6 discusses propagation of change tables that originate at an outerjoin node. We discuss the optimality of our techniques under some reasonable cost model in Section 7. A brief survey of related work is presented in Section 8. Finally, we present our concluding remarks in Section 9.

Notations. We consider only *bag* semantics in this article, i.e., all the relational operators used are duplicate-preserving. We use $\uplus$ to denote bag union, $\dot{-}$ to denote monus (bag minus), $\bigtriangledown E$ to denote deletions from a bag-algebra expression E, $\bigtriangleup E$ to denote insertions into E, σ_p to denote selection on condition p, Π_A to denote duplicate-preserving projection on a set of attributes A, π to denote the generalized projection operator (note that we use slightly different symbols for duplicate-preserving projection (Π) and for generalized projection (π) operators), $\times$ to denote cross-product, $\bowtie$ to denote natural join, and $\bowtie_J$ and $\overset{fo}{\bowtie}_J$ to denote join and full outerjoin operations with the join condition J. The symbols $\overset{lo}{\bowtie}$ and $\overset{ro}{\bowtie}$ are used for left and right outerjoin respectively. Also, $Attrs(J)$ denotes the set of attributes used in a predicate J or a relation J.

The only operators that may require explanation are the outerjoin and generalized projection operators. The (full) *outerjoin* differs from an ordinary join by including in the result any "dangling"[2] tuple of either relation after "padding" it with `NULL`'s in those attributes that belong to the other relation. For example, $R(A,B) \overset{fo}{\bowtie}_{R.B=S.B} S(B,C)$ will include a tuple $(a, b, \texttt{NULL}, \texttt{NULL})$, if $(a,b) \in R$ and $(b,c) \notin S$ for any c. One variant of the outerjoin operator is a *left* (*right*) outerjoin, where the dangling tuples of only the left (right) operand relation are padded with `NULL`'s and included in the result. Hence, in the above example, $(a, b, \texttt{NULL}, \texttt{NULL})$ would be included in $R \overset{lo}{\bowtie}_J S$, but not in $R \overset{ro}{\bowtie}_J S$. The *generalized projection* operator

[1] In work done concurrently with ours, Griffin and Kumar [GK98] derived expressions for propagating insertions and deletions through outerjoin operators.

[2] Dangling tuples are the ones that fail to join with any tuple from the other relation.

introduced in [GHQ95] is used to algebraically represent the groupby operation of SQL. For example, we could use the following expression to define the `SISales` view of Example 1 on the next page.

$$\texttt{SISales} = \pi_{storeID,itemID,SumSISales=sum(price),NumSISales=count(*)}(\sigma_{date>1/1/95}(\texttt{sales}))$$

We will explain the notation $\equiv_G$ in Section 5.

2 Motivating Examples and Previous Approaches

EXAMPLE 1

Relations. Consider the classic example of a warehouse containing information about stores, items, and day-to-day sales. The warehouse stores the information in three base relations viz. `stores`, `items`, and `sales` having the following schemas.

`stores`(storeID, city, state)
`items`(itemID, category)
`sales`(storeID, itemID, date, price)

For each store location, the relation `stores` contains the storeID, the city, and the state in which the store is located. For each item, the relation `items` contains its itemID and its category. An item can belong to multiple categories. The relation `sales` contains detailed information about sales transactions. For each item sold, the relation `sales` contains a tuple storing the storeID of the selling store, itemID of the item sold, date of sale, and the sale price.

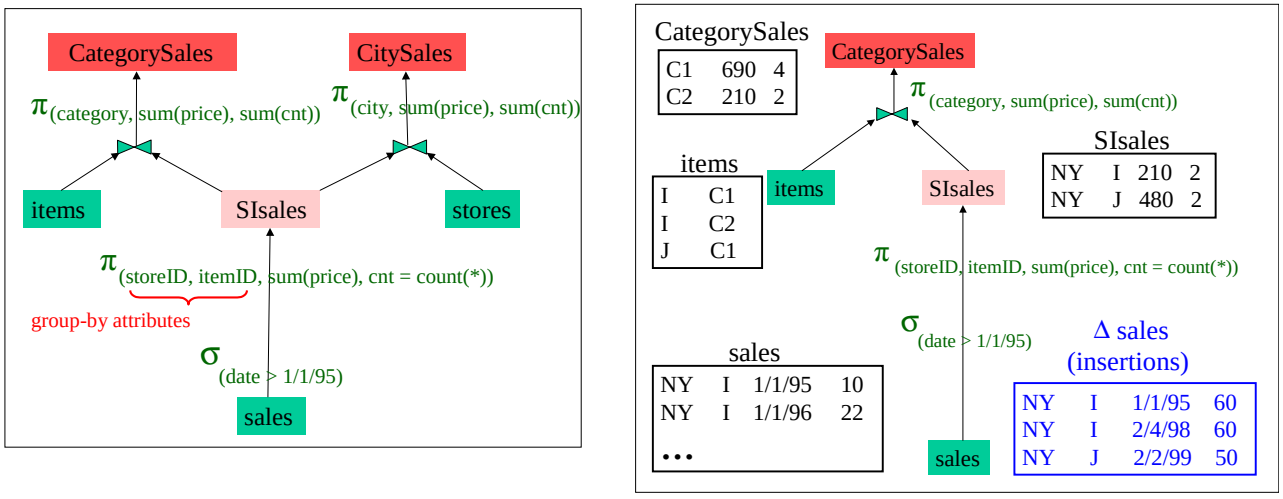


Figure 1: Example 1. a) Relations and Views. b) Instance of base relations and insertions into `sales`. [The aggregated column names have been omitted or shortened for clarity.]

Views. Consider the views `SISales`, `CitySales`, and `CategorySales` defined over the base relations as shown in Figure 1(a). The view `SISales` computes for each storeID and itemID the total price of items sold after 1/1/95. The view `SISales` is an intermediate view used to define the views `CitySales` and `CategorySales`. The view `CitySales` stores, for each city, the total number and dollar value of sales of all the stores in the city. The view `CategorySales` stores the total sale for each category of items. All the above described views consider only those sales that occur after 1/1/95. The views `CitySales` and `CategorySales` are stored

(materialized) at the data warehouse and this is represented below by the keyword "MATERIALIZED"[3] in the SQL definitions of the views. We wish to maintain these materialized views in response to insertions to the base relation `sales` for the instance shown in Figure 1(b).

```
CREATE VIEW SISales AS
SELECT storeID, itemID, sum(price) AS SumSISales, count(*) AS NumSISales
FROM   sales
WHERE date > 1/1/95
GROUP BY storeID, itemID;

CREATE MATERIALIZED VIEW CitySales AS
SELECT city, sum(SumSISales) AS SumCiSales, sum(NumSISales) AS NumCiSales
FROM   SISales, stores
WHERE SISales.storeID = stores.storeID
GROUP BY city;

CREATE MATERIALIZED VIEW CategorySales AS
SELECT category, sum(SumSISales) AS SumCaSales, sum(NumSISales) AS NumCaSales
FROM   SISales, items
WHERE SISales.itemID = items.itemID
GROUP BY category;
```

Previous Techniques. Of the previous approaches, only [Qua97] provides techniques to maintain general view expressions involving aggregate operators. Prior works in [GMS93], [GL95], and [MQM97] consider aggregates, but in a very limited fashion. The works of [GL95], [PSCP02], and [MQM97] are restricted to views that have at most one aggregate operator as the last operator in the view expression.[4] In contrast, the techniques developed in this article apply to general view expressions involving aggregate and outerjoin operators. Also, group-by attributes are not allowed in [GL95], and [MQM97] maintains views on star schemas only. Thus, for the case of general view expressions involving aggregate operators, we can compare our techniques with that of [Qua97] only.

Griffin and Libkin in [GL95] update view expressions by recursively computing insertions and deletions for each of the subexpressions in the view expression in response to changes at the base relations. Quass in [Qua97] extends the techniques in [GL95] by including aggregate operators. In our example, the insertions to `sales`, $\triangle$`sales`, result in insertions ($\triangle$`SISales`) and deletions ($\triangledown$`SISales`) to the view `SISales`, which is an aggregate view over the base relation `sales`. The expressions that compute $\triangle$`SISales` and $\triangledown$`SISales`, as derived in [Qua97], are quite complex (see Appendix A). As `SISales` is not materialized, the maintenance expressions for `SISales` essentially recompute the aggregate values of the affected tuples in `SISales` from the base relation `sales`. Using the propagation equations from [Qua97], one can propagate $\triangle$`SISales` and $\triangledown$`SISales` upwards to obtain expressions for $\triangledown$`CitySales`, $\triangle$`CitySales`, $\triangledown$`CategorySales`, and $\triangle$`CategorySales`. Figure 2(a) illustrates the [Qua97] technique for updating `CategorySales` in response to insertions into `sales`. As emphasized in the figure, the computation of $\triangle$`SISales` and $\triangledown$`SISales` require querying the base relation `sales`, because the intermediate view `SISales` is *not* materialized.

[3]The keyword "MATERIALIZED" is not supported by SQL, but has been introduced in this article.

[4]In some cases, view expressions can be rewritten so that aggregation is the last operator, but the rewritten query has worse query performance.

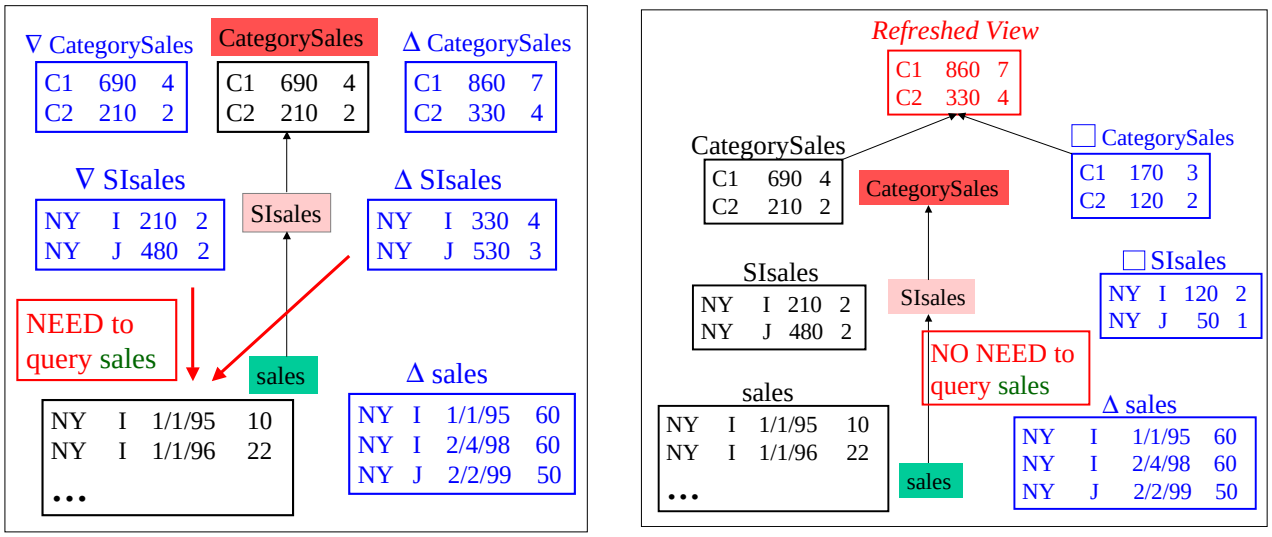


Figure 2: (a) [Qua97] Approach (b) Our Change-Table Approach

Our Techniques. The approach proposed in this article is the following. Instead of computing and propagating insertions and deletions beyond an aggregate node `SISales`, we compute and propagate a change table for `SISales`.[5] We show that propagation of change tables yields very efficient and simple maintenance expressions for general view expressions involving aggregate and/or outerjoin operators. The change table cannot be simply inserted into or deleted from the materialized view. Rather, the change table must be applied to the materialized view using a special "refresh" operator, which we will define formally in Section 4. We denote the change table of a view V by $\square V$, and a refresh operator by $\sqcup_{\theta}^{U}$, where θ and U are its parameters specifying join conditions and update functions respectively. However, in this section, we will simply use *REFRESH* to denote the refresh operator and ignore the refresh parameters.

For our example, we start with computing the change table $\square$`SISales` that summarizes the net changes to `SISales`. For this first level of aggregates, the expression that computes $\square$`SISales` is similar to that derived in [MQM97]. The change table $\square$`SISales` is computed from the insertions and deletions into `sales` by using the same generalized projection (aggregation) as that used for defining `SISales`. More precisely,

$$\square\texttt{SISales} = \pi_{storeID,itemID,SumSISales=sum(price),NumSISales=sum(\_count)}(\Pi_{storeID,itemID,price,\_count=1}(\sigma_p(\triangle\texttt{sales})) \uplus \Pi_{storeID,\ itemID,\ price=\ -price,\ \_count=\ -1}(\sigma_p(\triangledown\texttt{sales}))), \text{ where } p \text{ is } (date > 1/1/95)$$

Figure 2(b) presents an instance of the base relation `sales` and the table $\triangle$`sales`, which is the set of insertions into `sales`. For the given tables, Figure 2(b) also shows the computed table $\square$`SISales`. Next we propagate the change table $\square$`SISales` upwards to derive expressions for the change tables $\square$`CitySales` and $\square$`CategorySales`.

$$\square\texttt{CitySales} = \pi_{city,SumCiSales=sum(SumSISales),NumCiSales=sum(NumSISales)}(\square\texttt{SISales} \bowtie \texttt{stores})$$

$$\square\texttt{CategorySales} = \pi_{category,SumCaSales=sum(SumSISales),NumCaSales=sum(NumSISales)}(\square\texttt{SISales} \bowtie \texttt{items})$$

Figure 2(b) shows the change table $\square$`CategorySales` for the instance of the base table `items` in Figure 1(b). The change table $\square$`CitySales` can be similarly computed. The new propagated change tables are then used to refresh their respective materialized views `CitySales` and `CategorySales` using the refresh equations below.

[5] A change table is a general form of summary-delta tables introduced in [MQM97].

Table	Number of Tuples	Changes (No. of Tuples)	Tuple Reads and Writes	
			Previous Work	Our Approach
`sales`	1,000,000,000	10,000		
`stores`	1,000	-		
`InfoStates`	100	-		
`items`	10,000	-		
V_1 = `SISales`	1,000,000	600	610,000 ([Qua97])	10,000
V_2 = `CitySales`	100	10	1,020 ([Qua97])	1,020
V_3 = `CategorySales`	1,000	1,000	12,000 ([Qua97])	12,000
Total for V_1, V_2, V_3			**623,020** [Qua97]	**23,020**

Table 1: Benefits of propagating change tables (Materialized views are V_2 and V_3.)

The details of the refresh equations are given later in Example 4.

$$\texttt{CitySales} = \texttt{CitySales}\ \textit{REFRESH}\ \square\texttt{CitySales}, \text{ and}$$

$$\texttt{CategorySales} = \texttt{CategorySales}\ \textit{REFRESH}\ \square\texttt{CategorySales}$$

As `SISales` is not materialized, it does not need to be refreshed. Also, as emphasized in Figure 2(b), we don't need to query the base relation `sales` for updating `CitySales` or `CategorySales`, which results in huge savings. We illustrate the refresh operation by showing how the `CategorySales` view is refreshed. Figure 2(b) shows the materialized table `CategorySales` for the given instance of base tables. For each tuple $\square v$ in $\square$`CategorySales`, we look for a matching tuple in `CategorySales` using the join condition `CategorySales`.category = $\square$`CategorySales`.category (specified in one of the parameters of *REFRESH*). For example, the tuple <C1,170,3> in $\square$`CategorySales` matches with the tuple v =<C1,690,4> of `CategorySales`. The tuple <C1,170,3> in $\square$`CategorySales` means that three more sales totaling $170 have occurred for C1 category. The total number of sales for C1 is now 7 for a total amount of $860. To reflect the change, the tuple v is updated to <C1,860,7> by adding together the corresponding aggregated attributes (specified in another parameter of *REFRESH*).

Cost Comparison. Consider the database sizes shown in Table 1. We assume that the base relation `sales` has one billion sales transactions, and the base relations `stores`, `InfoStates`, and `items` have 1,000, 100, and 10,000 tuples respectively. We illustrate the various maintenance approaches for the case when 10,000 tuples are inserted into the base relation `sales`. Table 1 shows the number of tuples changed in the views, as a result of the insertion of 10,000 tuples into `sales`. The table also shows the number of tuple accesses (reads and writes) incurred by different maintenance techniques to update the materialized views. We explain the computation of these tuple access numbers below.

Cost Model. We have used the simple model of counting tuple accesses for the sake of convenience as orders of magnitude improvement in the number of tuples computed and accessed translates directly into significant improvement in number of disk accesses. In Section 7, we show that the change-table technique is superior to previous techniques under a data warehouse cost model.

Computation of Tuple Accesses. Appendix A shows that the total number of tuples accessed by the [Qua97] technique is 623,020. To compute the number of tuple accesses for our techniques, note that most of the computation is done in computing $\square$`SISales`, which requires 10,000 tuple accesses to read $\triangle$`sales`. Given the small sizes of $\square$`SISales`, `items`, and `stores`, the rest of the computation can be done in main memory and hence, the total number of tuples accesses is 10,000 (to read $\triangle$`sales`) + 11,000 (to read `items` and `stores`)

+ 2,020 (to refresh CitySales and CategorySales) = 23,020, showing that our technique is very efficient in comparison to previous approaches.[6] □

EXAMPLE 2

Outerjoin Views. The change-table technique can also be used for maintenance of view expressions involving outerjoin operators. Outerjoin views are supported by SQL and are commonly used in practice, such as for data integration [GJM96]. We extend the previous example to illustrate our techniques for the case of outerjoin views. For this example, we require another relation InfoStates that stores area and population for each state. We also define views SSInfoand SSFullInfo over the base relations. **Only the view SSFullInfo is materialized.** The schema of the relation InfoStates and the view definitions are as follows.

$$\begin{aligned} & \texttt{InfoStates}(state, area, population) \\ \texttt{SSInfo} &= \texttt{sales} \overset{fo}{\bowtie}_{\texttt{sales}.storeID=\texttt{stores}.storeID} \texttt{stores} \\ \texttt{SSFullInfo} &= \texttt{SSInfo} \overset{fo}{\bowtie}_{\texttt{stores}.state=\texttt{InfoStates}.state} \texttt{InfoStates} \end{aligned}$$

The view SSInfo stores the full outerjoin of the base relations sales and stores, retaining stores that have had no sales due to some reasons and also retaining those sales whose corresponding storeID is missing from the table stores, because, maybe, the table stores has not been updated yet. We define another view SSFullInfo which is the outerjoin view of SSInfo and InfoStates.

Maintenance Expressions. We illustrate our technique for maintaining outerjoin views by deriving maintenance expressions for SSFullInfo view. The net changes to SSInfo, in response to insertions $\triangle$sales into sales, can be succinctly summarized in a change table $\square$SSInfo. The change table $\square$SSInfo is computed, propagated up, and then used to refresh the SSFullInfo view as shown below.

$$\begin{aligned} \square\texttt{SSInfo} &= \triangle\texttt{sales} \overset{lo}{\bowtie}_{\texttt{sales}.storeID=\texttt{stores}.storeID} \texttt{stores} \\ \square\texttt{SSFullInfo} &= \square\texttt{SSInfo} \overset{lo}{\bowtie}_{\texttt{stores}.state=\texttt{InfoStates}.state} \texttt{InfoStates} \\ \texttt{SSFullInfo} &= \texttt{SSFullInfo}\ \mathit{REFRESH}\ \square\texttt{SSFullInfo} \end{aligned}$$

Recall that $\overset{lo}{\bowtie}$ denotes the left-outerjoin operator. The refresh of SSFullInfo proceeds as follows. Each tuple in $\square$SSFullInfo is matched with tuples in SSFullInfo that have the same stores and InfoStates attributes, but have all NULL's in the attributes of sales. This join condition used for matching is specified in one of the parameters of the refresh operator. Each matching pair $(\square v, v)$, where $\square v \in \square$SSFullInfo and $v \in$ SSFullInfo, results in an update of v to $\square v$, in accordance with the update parameter of *REFRESH*. The tuples in $\square$SSFullInfo that do not find a match in SSFullInfo are inserted into the view SSFullInfo.

Cost Comparison. Given the small sizes of stores, InfoStates, $\triangledown$sales, and $\square$SSInfo, the number of tuple accesses required to compute the change tables and refresh V_5 is 10,000 (to read $\triangle$sales) + 1,000 (to read stores) + 100 (to read InfoStates) + 20,000 (to refresh SSFullInfo).

This is the first paper to address maintenance of general view expressions involving outerjoin operators. In work done concurrently with ours, [GK98] also reports an algorithm to handle view expressions involving outerjoins, extending previous work on maintenance of outerjoin views in [GJM97]. [GK98] uses insertion and deletion sets to propagate changes through outerjoin operators. Thus, insertions in sales results in insertions and deletions at SSInfo, which in turn result in insertions and deletions at SSFullInfo. However, according to the change propagation equations in [GK98], in order to compute the insertions and deletions at SSFullInfo, we have to compute the intermediate view SSInfo, thereby incurring more than a billion tuple accesses. □

[6] Note that the above estimation of number of tuple accesses will also hold irrespective of the items and stores table sizes, if the change tables $\square$SISales, $\square$CitySales, and $\square$CategorySales are small enough to fit in the main memory.

Table	Number of Tuples	Changes (No. of Tuples)	Tuple Reads and Writes	
			Previous Work	Our Approach
`InfoStates`	100	-		
V_4 = `SSInfo`	1,000,000,010	10,000	2,000,000,020 [GJM97] 11,000 [GK98]	11,000
V_5 = `SSFullInfo`	1,000,000,020	10,000	10,100 [GJM97]/[GL95] 1,000,020,110 [GK98]	20,100
Total for V_4 and V_5			**1,000,031,110** [GK98]	**31,100**

Table 2: Benefits of propagating change tables (V_5 is the materialized view.)

3 The Change-Table Technique for View Maintenance

In this section, we explain the framework developed in [QW91, GL95] for deriving incremental view maintenance expressions and relate it to the change-table technique developed in this article.

Let a database contain a set of relations $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$. A *change transaction* t is defined to contain the expression $R_i \leftarrow (\ R_i \ \dot{-}\ \nabla R_i) \ \uplus \ \triangle R_i$, for each relation R_i, where ∇R_i are the deletions from R_i, and $\triangle R_i$ are the insertions into R_i. Let V be a bag-algebra expression defined on a subset of the relations in $\mathcal{R}$. The *refresh-expression* $New(V,t)$[7] is used to compute the new value of V. Griffin and Libkin in [GL95] define the expression New(V,t) to be:

$$\text{New}(V,t) = (V \ \dot{-}\ \nabla(V,t)) \uplus \triangle(V,t).$$

So, the goal in deriving view maintenance expressions for a view V is to derive two functions $\nabla(V,t)$ and $\triangle(V,t)$ such that for any transaction t, the view V can be maintained by evaluating $(V \ \dot{-}\ \nabla(V,t)) \uplus \triangle(V,t)$. In order to derive $\nabla(V,t)$ and $\triangle(V,t)$, [GL95] gives *change propagation equations* that show how deletions and insertions are propagated up through each of the relational operators. The work of [GL95] was extended to include aggregate operators by Quass in [Qua97].

The change-table technique presented in this article can be thought of as introducing a new definition for New(V,t). We define the expression New(V,t) for general view expressions as

$$\text{New}(V,t) = (V \sqcup_{\theta}^{U} \square(V,t)),$$

where $\square(V,t)$ is called the *change table*, $\sqcup_{\theta}^{U}$ is the `refresh` operator used to apply the net changes in a change table to its view, and (θ, U) are the parameters of the `refresh` operator. The parameter θ specifies the join conditions on the basis of which the tuples from the change table and the view are matched, and U specifies the functions that are used to update the matched tuples in V.

The new definition of New(V,t) is motivated from the following observation. In the case of general view expressions involving aggregate operators, it is usually more efficient to propagate the change tables beyond an aggregate operator, instead of propagating insertions and deletions. Propagation of a change table is particularly efficient when the change table depends only on the changes to the base relation (self-maintainability [GJM96]), while the insertions and deletions depend on the old value of the view. As we showed in the motivating example, if the aggregate node is not materialized, the computation of insertions and deletions could be very expensive.

The new definition of New(V,t) means that in order to obtain a complete technique, we need to define a general refresh operator, show how to generate a change table, and how to propagate a change table through various operators. In the following section, we present a formal definition of the `refresh` operator. In later

[7] [GL95] uses the notation pre(t,V) instead.

sections, we derive change propagation equations for general view expressions involving aggregate and outerjoin operators. Although in the following sections we derive incremental maintenance expressions for changes at one base table at a time, changes occurring simultaneously at multiple base tables can be incorporated by propagating one change at a time[8] using the techniques presented in this article.

4 The `refresh` Operator

In this section, we give a formal treatment of the refresh operator. Given a materialized table V and its change table $\square V$, the `refresh` operator is used to apply the changes represented in a change table. The binary `refresh` operator is a generalization of the refresh algorithm used in [MQM97] and can be implemented using the `modify` operation discussed in [QM97].

We denote the `refresh` operator by $\sqcup_{\theta}^{U}$, where θ is a pair of two mutually exclusive join conditions and U is a list of update function specifications. The `refresh` operator takes two operands, a view V to be updated and a corresponding change table (denoted by $\square V$).

Let V and $\square V$ be tables with the same attribute names $A_1, A_2, \ldots, A_n$. The subscript θ associated with the operator is a pair of join conditions $\mathcal{J}_1$ and $\mathcal{J}_2$. The update list U is a specification of how the attributes are updated. In an expression $V \sqcup_{\theta}^{U} \square V$, each tuple $\square v$ of $\square V$ is checked for possible matches (due to $\mathcal{J}_1$ or $\mathcal{J}_2$) with tuples in V. If a match is found due to the join condition $\mathcal{J}_1$, then the corresponding matching tuple v of V is changed using the specifications in the update list U (as described in the next paragraph). If the match is due to $\mathcal{J}_2$, the tuple v of V is deleted. The unmatched tuples in $\square V$ are inserted into V. The matching done is `one-to-one` in the sense that a tuple $\square v \in \square V$ is matched with at most one tuple in V and vice-versa. If $\square v$ finds more than one match[9] in V, then an arbitrary matching tuple from V is picked.

The tuple v of V matching with the tuple $\square v$ of $\square V$ due to join condition $\mathcal{J}_1$ is updated as follows. Let $U = <(A_{i_1}, f_1), (A_{i_2}, f_2), \ldots, (A_{i_k}, f_k)>$, where $A_{i_1}, \ldots, A_{i_k}$ are attributes of V and $f_1, \ldots, f_k$ are binary functions. For *each* pair (A_{i_j}, f_j) in U, the A_{i_j} attribute of v is changed to $f_j(v(A_{i_j}), \square v(A_{i_j}))$, where $v(X)$ and $\square v(X)$ denote the values of the X attribute of v and $\square v$ respectively.

Implementation. One simple way to implement the `refresh` operator is to use a nested loop algorithm, with the change table as the outer table, and the materialized view table as the inner table. The nested loop algorithm is just one possible way to implement the `refresh` operator. In fact, Quass and Mumick [QM97] show that the refresh operation can be implemented more efficiently by using existing outerjoin methods inside the DBMS.

EXAMPLE 3 Consider the view `CategorySales` defined in Example 1 earlier. The `CategorySales` table as defined in Example 1 computes the total sales for each category. In this example, we illustrate the refresh operation by applying the changes summarized in a change table $\square$`CategorySales` to its view `CategorySales` using the `refresh` operator.

For this example, we consider the instance of the base table shown in Figure 2(b). The figure also shows the materialized table `CategorySales` for the given instance. In response to the insertion of the table $\triangle$`sales` into the base table `sales`, the change table $\square$`CategorySales` can be computed and is shown in the figure. The view `CategorySales` is refreshed using the expression `CategorySales` $\sqcup_{\theta_3}^{U_3}$ $\square$`CategorySales`, where the

[8] Multiple changes to the same base table can be merged into one insertion and one deletion, and multiple changes occurring at different base tables can be handled in an arbitrary order. Also, multiple occurrences of a base table in a view expression can be treated as different base tables for the purposes of incremental view maintenance.

[9] Such a situation may arise when a change-table is propagated through a cross product operator (Row 3 of Table 1). See Page 15 for details.

parameters, $\theta_3 = (\mathcal{J}_1, \mathcal{J}_2)$ and U_3, of the `refresh` operator are defined as follows. Here, we use $\equiv_{category}$ to represent the predicate (`CategorySales`.category = (□ `CategorySales`).category).

- $\mathcal{J}_1$ is ($\equiv_{category}$ $\wedge$ ((`CategorySales`.NumCaSales + (□`CategorySales`).NumCaSales) $\neq$ 0))
- $\mathcal{J}_2$ is ($\equiv_{category}$ $\wedge$ ((`CategorySales`.NumCaSales + (□`CategorySales`).NumCaSales) = 0))
- U_3 = <(SumCaSales, f), (NumCaSales, f)>, where $f(x, y) = x + y$ for any x, y.

Now, we try to run the refresh operation on the view `CategorySales` and its change table □`CategorySales` of Figure 2(b). The first tuple $\square v_3$ = <C1,170,3> of □`CategorySales` matches with the tuple v_3 = <C1,690,4> in `CategorySales` due to the join condition $\mathcal{J}_1$. The match results in an update of the tuple <C1,690,4> in `CategorySales` according to the specifications in the update list U_3. Thus, the attribute *SumCaSales* of the tuple v_3 is changed to $\square v_3(SumCaSales) + v_3(SumCaSales) = 170 + 690 = 860$ and the attribute *NumCaSales* is changed to 4 + 3 = 7. Similarly, the tuple <C2,210,2> ∈ `CategorySales` matches with the tuple <C2,120,2> of □`CategorySales` and is updated to <C2,330,4>.

To illustrate deletion from the view `CategorySales`, let us assume that the change table □`CategorySales` contains a tuple b = <C2,-210,-2> as a result of a deletion of a couple of tuples from the `sales` table. The tuple b = <C2,-210,-2> will match with the tuple v = <C2,210,2> in `CategorySales` due to the join condition $\mathcal{J}_2$, and the match will result in deletion of v from `CategorySales`. □

5 Propagating Change Tables Generated at Aggregates

In this section, we show how to generate a change table at an aggregate node, and derive change-propagation equations used to propagate these change tables through various operators. We start with a few definitions.

Definition 1 (**Aggregate-change table**) A change table is defined as an *aggregate-change table* if it either originated at an aggregate operator or is a result of propagation of a change table that originated at an aggregate node.

For example, the change tables, □`SISales`, □`CitySales`, and □`CategorySales`, computed for the views `SISales`, `CitySales`, and `CategorySales` respectively in Example 1 are aggregate-change tables. □

The notions of aggregate-change table and outerjoin-change table (introduced later in Section 6) have been defined only for simplying the presentation of the material in this article. We will show that a restricted definition of the general `refresh` operator suffices to refresh a view using its aggregate-change table. The restricted `refresh` operator yields very simple change-propagation equations.

More Notations. We use the notation *Attrs*(φ) to represent the set of attributes referenced in a refresh parameter φ. Thus, Attrs(U) refers to the set of attributes specified in the update list U and Attrs(θ) represents Attrs($\mathcal{J}_1$) ∪ Attrs($\mathcal{J}_2$), where $\mathcal{J}_1$ and $\mathcal{J}_2$ are the join conditions in $\theta = (\mathcal{J}_1, \mathcal{J}_2)$. Also, for a set of attributes G, we use the notation $\equiv_G$ to represent the predicate

$$\bigwedge_{g \in G}(LHS.g = RHS.g)$$

in a join condition, where LHS and RHS are the left and right operand relations of the join operator. For example, when J is ($\equiv_G$ $\wedge$ p), the expression $R \bowtie_J S$ denotes a join operation with the join condition $(\bigwedge_{g \in G}(R.g = S.g) \wedge p)$, for a predicate p and a set of attributes G in R and S.

5.1 Generating the Aggregate-Change Table

Consider a view V defined as an aggregation over a select-project-join (SPJ) expression. In this section, we give a brief description of how an aggregate-change table is generated at V in response to the insertions

Aggregate	Change due to N
COUNT(*)	-1
SUM(expr)	-expr
MIN(expr)	expr
MAX(expr)	expr

Table 3: Table used to change the aggregate attributes in deletions

and deletions at the base tables. The method is similar to that of generating "summary-delta table" for a "summary table" [MQM97].

For the case when a view V is defined as an aggregation over an SPJ expression, the insertions and deletions to the base tables can be propagated to V as a single aggregate-change table, which we denote by $\square V$. Without loss of generality, assume V to be $\pi_{G,B=f(A)}(R)$, where R is the SPJ subview, G is the set of group-by attributes, f is an aggregate function, and A is an attribute of R. In the case of self-maintainable aggregate functions, the aggregate-change table $\square V$ can be computed from the insertions and deletions into R by using the same generalized projection as that used for defining the view V. More precisely, $\square V$ can be computed as

$$\square V = \pi_{G,f(A),Count=sum(\_count)}(\Pi_{G,A,\_count=1}(\triangle R) \uplus \ \Pi_{G,A=N(A),\_count=-1}(\triangledown R)),$$

where the function N is suitably defined depending on the aggregate function f as shown in Table 3. For example, in the case of a sum aggregate the function N negates the attribute value passed.

For the case of aggregate functions that are not self-maintainable, we need more complex functions in the update list U of the `refresh` operator. For example, to handle deletions from a subview of a simple MAX aggregate view, the change table is appropriately defined as before (aggregation of the deleted values), but the update function has an embedded SQL query that efficiently computes, when needed, the new MAX value by accessing the base relations. The advantage of our approach comes from *delaying* the update-query execution as late as possible in the view expression tree and executing the query only when the matching tuple exists in the materialized view (and has not been filtered through other operators after the aggregation). The change propagation equations and other formalisms presented in this article are independent of the complexity of update functions used in the update list U of `refresh`.

In a general view expression tree V, an aggregate-change table is generated at the first non-relational operator which is an aggregate node, and then propagated upwards through various operators to V, as shown in Figure 3 (a) on page 20. Figure 3 (a) refers to an `aggregate-refresh` operator, which is a restricted form of the `refresh` operator that is used to apply aggregate-change tables and is defined in the next subsection. We derive change propagation equations for aggregate-change tables in the following subsection.

5.2 `Refresh` Operator for Applying Aggregate-Change Tables

In this section, we define the characteristics of the "aggregate-refresh" operator that is used to apply an aggregate-change table to its view. The "aggregate-refresh" operator is a special case of the generic `refresh` operator defined in Section 4. In the next subsection, we derive simple change propagation equations using these special characteristics.

Recall that the expression used to refresh a view V using its change table $\square V$ is: $V = V \ \sqcup_{\theta}^{U} \square V$, where $\theta = (\mathcal{J}_1, \mathcal{J}_2)$ and U is the update list. In the case of the `aggregate-refresh` operator used to apply aggregate-change tables, $\mathcal{J}_1$ is $(\equiv_G \ \wedge \neg p_1)$ and $\mathcal{J}_2$ is $(\equiv_G \ \wedge p_1)$, for some predicate p_1 and a set of attributes G common to both V and $\square V$. As defined before, the notation $\equiv_G$ here represents the predicate $\bigwedge_{g \in G}(V.g = \square V.g)$, as V

No.	V	New V $\theta = (\mathcal{J}_1, \mathcal{J}_2)$ $\mathcal{J}_1$ is $\equiv_G \wedge \neg p_1$ $\mathcal{J}_2$ is $\equiv_G \wedge p_1$	Refresh Equation	$\square V$	Conditions
1	$\sigma_p(E_1)$	$\sigma_p(E_1 \sqcup_\theta^U \square E_1)$	$V \sqcup_\theta^U \sigma_p(\square E_1)$	$\sigma_p(\square E_1)$	$\text{Attrs}(p) \subseteq G$
2	$\Pi_A(E_1)$	$\Pi_A(E_1 \sqcup_\theta^U \square E_1)$	$V \sqcup_\theta^U \Pi_A(\square E_1)$	$\Pi_A(\square E_1)$	$\text{Attrs}(\theta) \subseteq A$
3	$E_1 \times E_2$	$(E_1 \sqcup_\theta^U \square E_1) \times E_2$	$V \sqcup_{\theta_1}^U (\square E_1 \times E_2)$ $\theta_1 = (\mathcal{J}_1 \wedge \equiv_\tau, \mathcal{J}_2 \wedge \equiv_\tau)$ $\tau = \text{Attrs}(E_2)$	$\square E_1 \times E_2$	
4	$E_1 \bowtie_J E_2$	$(E_1 \sqcup_\theta^U \square E_1) \bowtie_J E_2$	$V \sqcup_{\theta_1}^U (\square E_1 \bowtie_J E_2)$ $\theta_1 = (\mathcal{J}_1 \wedge \equiv_\tau, \mathcal{J}_2 \wedge \equiv_\tau)$ $\tau = \text{Attrs}(E_2)$	$\square E_1 \bowtie_J E_2$	$\text{Attrs}(J) \subseteq G$
5	$E_1 \uplus E_2$	$(E_1 \sqcup_\theta^U \square E_1) \uplus E_2$	$((V \dot{-} E_2) \sqcup_\theta^U \square E_1) \uplus E_2)$	$\square E_1$	
6	$\pi_{G',F}(E_1)$ $F = f_1(A_1),$ $\ldots, f_k(A_k).$	$\pi_{G',F}(E_1 \sqcup_\theta^U \square E_1)$	$V \sqcup_{\theta_3}^{U_3} \pi_{G',F}(\square E_1)$ $\theta_3 = (\equiv_{G'} \wedge \neg p_3, \equiv_{G'} \wedge p_3)$ p_3 is $(V.Cnt + \square V.Cnt) \neq 0$ $U_3 = <(A_1, f_1), \ldots, (A_k, f_k)>$	$\pi_{G',F}(\square E_1)$	$A_i \in \text{Attrs}(U), G' \subseteq G,$ f_i's are distributive, and the function in U for A_i is f_i.
7	$E_1 \overset{fo}{\bowtie}_J E_2$	$(E_1 \sqcup_\theta^U \square E_1) \overset{fo}{\bowtie}_J E_2$	$V \sqcup_{\theta_1}^U (\square E_1 \overset{lo}{\bowtie}_J E_2)$ $\theta_1 = (\mathcal{J}_1 \wedge \equiv_\tau, \mathcal{J}_2 \wedge \equiv_\tau)$ $\tau = \text{Attrs}(E_1)$	$(\square E_1 \overset{lo}{\bowtie}_J E_2)$	$\text{Attrs}(J) \subseteq G$

Table 4: Change propagation equations for propagating aggregate-change tables

and $\square V$ are the left and right operands of the join operator. Also, the set of attributes G is disjoint from the set of attributes, Attrs(U), that are being updated. The predicate p_1 specifies when the matching tuple in V is to be deleted, i.e., when the value of the attribute that stores the number of deriving base tuples becomes zero. The above characteristics are summarized in the definition of an `aggregate-refresh` operator below.

Definition 2 (`Aggregate-refresh` **Operator**) A `refresh` operator $\sqcup_\theta^U$, where $\theta = (\mathcal{J}_1, \mathcal{J}_2)$, is said to be an `aggregate-refresh` operator if, for some predicate p_1 and a set of attributes G common to both the view and its aggregate-change table,

(1) The join conditions $\mathcal{J}_1$ and $\mathcal{J}_2$ can be represented as:
- $\mathcal{J}_1$: $(\equiv_G \wedge \neg p_1)$
- $\mathcal{J}_2$: $(\equiv_G \wedge p_1)$, and

(2) $G \cap \text{Attrs}(U) = \phi$. □

The above characteristics of the `aggregate-refresh` operator are used to derive simple change propagation equations for propagating aggregate-change tables.

5.3 Change Propagation Equations

For the purposes of change propagation equations shown in Table 4, we assume that an aggregate-change table has been *generated* (as shown in Section 5.1) at the first aggregate operator in a view expression, in response to insertions and/or deletions at a base relation.[10] Table 4 gives change propagation equations for

[10] As mentioned before, when two or more base relations are updated simultaneously (or a relation appears more than once in a view expression), we handle the updates in an arbitrary order.

propagating (already generated) aggregate-change tables through relational, aggregate and outerjoin operators. In Theorem 1, we will prove the correctness of these change propagation equations.

Each row in the table considers propagation of an aggregate-change table through a relational, aggregate, or outerjoin operator. The first column gives the equation number used for later reference in examples. The second column in the table gives the outermost operator used in the definition of V. Consider a change transaction t consisting of the following change: $E_1 \leftarrow (E_1 \sqcup_\theta^U \square E_1)$, where E_1 is a subexpression, $\square E_1$ is an aggregate-change table and $\sqcup_\theta^U$ is an `aggregate-refresh` operator. We assume θ to be $(\mathcal{J}_1, \mathcal{J}_2)$, where $\mathcal{J}_1$ is $\equiv_G \ \wedge \neg p_1$ and $\mathcal{J}_2$ is $\equiv_G \ \wedge p_1$, for some predicate p_1 and a set of attributes G in E_1. The third column expresses the new expression for V due to change t, by replacing E_1 in V by $(E_1 \sqcup_\theta^U \square E_1)$.[11] The fourth column gives the refresh equation $New(V,t)$ that is used to refresh the view V. Theorem 1 proves that the refresh equations ($New(V,t)$ of the fourth column) are correct, by showing that they are equivalent to the expressions in the third column. Theorem 1 also proves that the `refresh` operator used in the refresh equation of the fourth column is an `aggregate-refresh` operator. The fifth column gives the expression for the propagated aggregate-change table $\square V$, which can be derived from the refresh equation of the fourth column. Finally, the last column of the table states the conditions under which the equivalence of the fourth column and third column expressions holds, i.e., the conditions under which the change propagation can be done. If the condition is not satisfied, then the refresh equation cannot be used to propagate the aggregate-change table. We will show later how to handle changes at an operator node when the conditions in the sixth column are not satisfied.

The *first row* of the table depicts the case of a selection view $V = \sigma_p(E_1)$, where E_1 is a subexpression. For the case of selection view, the condition required for the change propagation is $\text{Attrs}(p) \subseteq G$. In other words, an aggregate-change table can be propagated through a selection operator only if the selection condition is defined over the G attributes, which are the attributes that are not being updated by the update functions of the `refresh` operator.

The *second row* depicts the case of projection on a set of attributes A. In the case of propagation through the cross product (*third row*), the parameter θ of the refresh operator is changed to also include the condition $\equiv_{Attrs(E_2)}$ in the join conditions $\mathcal{J}_1$ and $\mathcal{J}_2$. If E_2 has duplicates then the resulting refresh equation may involve one tuple of $\square V$ finding a match with multiple tuples in V. Based on the definition of the refresh operator, one of the matches is picked randomly. No conditions are specified in the fifth column, hence, an aggregate-change table can always be propagated through a cross product operator.

The *fifth row* considers propagation of an aggregate-change table through the bag union operator. In this case, we need to first apply a set of deletions ($\bigtriangledown V = E_2$), followed by refreshing the result with the change table ($\square V = \square E_1$), followed by inserting the set ($\bigtriangleup V = E_2$) into the result. For propagation beyond a union operator using the given refresh equation, we need to algebraically apply the rest of the view expression to the refresh equation, using the refresh equations when needed. One can derive a very efficient 'normal' refresh equation for the case of bag union operator, if the tuples in V are "tagged" L/R depending on whether they come from the left operand E_1 or the right operand E_2. We omit the details here.

The *sixth row* depicts the case of propagation through a generalized projection (aggregate) operator. For the purposes of aggregate-change tables, we assume that any subexpression involving an aggregate operator stores with each tuple a count of the number of deriving base tuples. This count is stored in a general attribute which we will call the *count* attribute. For example, NumCiSales and NumCaSales are count attributes in the views `CitySales` and `CategorySales` of Example 1. After propagation through the aggregate operator, the join conditions and the update specifications of the `aggregate-refresh` operator change as shown in the

[11] The case of changes occurring at *both* the subexpressions E_1 and E_2 can be handled by first propagating changes due to E_1, followed by propagating changes due to E_2.

fourth column. The attribute named *Cnt*, used in defining p_3 in the join conditions of θ_3 represents the count attribute of V. The condition in the fifth column says that each aggregate function f_i is distributive, the set of group-by attributes G' is a subset of G, and the update list U used to change E_1 contains (A_i, f_i) for all $1 \leq i \leq k$. Note that the duplicate-elimination operation is a special case of generalized projection, and is covered by the sixth row.

The *seventh row* gives the refresh equation for the case of propagating an aggregate-change table through a full outerjoin operation. In this case, the `refresh` operator specifications change as in the case of cross product. For simplicity, we have assumed that the aggregate-change table $\square E_1$ does not result in any deletions from E_1. Otherwise, a more extended refresh operator is required as discussed in Section 6.3.

Note that we do not give any change propagation equation for the case of monus, which makes monus operator a singularity point (see paragraph below).

Singularity Points. We call the operator nodes in a view expression tree, where none of the refresh equations in Table 4 apply, as *singularity* points. Aggregate-change tables cannot be propagated through singularity points. For example, a selection on the result of an aggregate function is a singularity point, because it will not satisfy the condition $Attr(p) \subseteq G$ given in the first row of Table 4. Consider a view V and a singularity point V_1, which is a subexpression of V, in the expression tree of V. As the changes to V_1 cannot be summarized into a change table, we instead compute insertions ($\triangle V_1$) and deletions ($\triangledown V_1$) into V_1 and propagate the insertions and deletions beyond V_1. The tables $\triangle V_1$ and $\triangledown V_1$ can be easily computed from the change table of its descendant in the expression tree. The computed insertions and deletions at a singularity point can then be propagated further upwards in the expression tree using techniques presented in [GL95] and this article (as they may result in change tables further on). Hence, the presence of singularity points in an expression tree does not preclude application of our change-table techniques for incremental maintenance.

Theorem 1 *Assume that the* `refresh` *operator used in the expression of the third column in Table 4 is an* `aggregate-refresh` *operator. Then,*

(1) the change propagation equations given in Table 4 for propagation of aggregate-change tables are correct, i.e., for each row, the expression in the third column is equivalent to the refresh equation in the fourth column, and

(2) the `refresh` *operator derived in the refresh equation (column 4) is an* `aggregate-refresh` *operator as well.*

Proof: We refer to the expression $E_1 \sqcup_\theta^U \square E_1$ as the *change equation* throughout this proof. As shown in the Table 4, the expression in the fourth column is referred to as the refresh equation.

Selection: $V = \sigma_p(E_1)$. It is easy to see that if a tuple $v \in V$ is deleted or updated during the refresh equation of the fourth column, then $v \in E_1$ is also deleted or updated in the same manner by the change equation. And as updates do not affect any attributes in Attrs(p), the updated v is retained in $\sigma_p(E_1 \sqcup_\theta^U \square E_1)$.

In this paragraph, we show that the refresh equation indeed captures *all* the updates or deletions required. First, note that in the change equation $E_1 \sqcup_\theta^U \square E_1$ if a tuple $\square e_1 \in \square E_1$ deletes/updates a tuple $v \in \sigma_p(E_1) \subseteq E_1$, then $\square e_1 \in \sigma_p(\square E_1)$. The above is true because the matched pair $(v, \square e_1)$ should have the same G attributes, and as Attrs(p) $\subseteq G$, if v satisfies p then $\square e_1$ must also satisfies p. Thus, such a tuple $\square e_1$ in $\sigma_p(\square E_1) = \square V$ would update/delete the corresponding tuple $v \in V = \sigma_p(E_1)$ in the refresh equation. Therefore, all the updates or deletions that happen to tuples in $\sigma_p(E_1)$ due to the change equation are also captured by the refresh equation. The update to a tuple $\bar{v} \notin \sigma_p(E_1)$ due to the change equation is irrelevant, as neither $\bar{v}$ nor its updated form will satisfy the predicate p.

Let I be the set of tuples that is inserted into E_1 due to the change equation. Now, we show that $\sigma_p(I)$ is inserted into V by the refresh equation, implying that the refresh equation doesn't miss any legitimate

insertions into V. Note that I is the set of tuples in $\square E_1$ that do not find a match in E_1. As $\sigma_p(I) \subseteq \sigma_p(\square E_1) = \square V$, no tuple in $\sigma_p(I)$ will find a match in $V \subseteq E_1$. Hence, $\sigma_p(I)$ will be inserted into V due to the refresh equation.

Finally, we show that all the insertions into V due to the refresh equation are legitimate. A tuple $\square v$ matches with a tuple v only if they have the same G attributes. Hence, if a tuple $\square v \in \sigma_p(\square E_1)$ is inserted into $V = \sigma_p(E_1)$ in the refresh equation (due to a lack of match in V), then $\square v$ will not find a match in E_1 also. Hence, $\square v \in \square E_1$ will be inserted into E_1 by the change equation, and as $\square v$ satisfies p, it will also be retained in $\sigma_p(E_1 \sqcup_\theta^U \square E_1)$.

Projection: $V = \Pi_A(E_1)$. If $\text{Attrs}(\theta) \subseteq A$, then the information needed to decide the effect of a tuple $\square e_1 \in \square E_1$ on a tuple $e_1 \in E_1$, if any, is available in $\Pi_A(e_1)$. Hence, $\Pi_A(\square E_1)$ can be applied directly to $V = \Pi_A(E_1)$. Also, note that the resulting `refresh` operator is also an `aggregate-refresh` operator.

Cross Product: $V = E_1 \times E_2$. Consider $(E_1 \times E_2) \sqcup_{\theta_1}^U (\square E_1 \times E_2)$, the refresh equation. Let us partition the tables $(E_1 \times E_2)$ and $(\square E_1 \times E_2)$ by the tuple values of E_2. As $\mathcal{J}_1$ and $\mathcal{J}_2$ in θ_1 include $\equiv_{\text{Attrs}(E_2)}$, each of the partitions is refreshed independently by the refresh equation. It is easy to see that a tuple $<\square e_1, e_2> \in \square V$ will match with a tuple $<e_1, e_2> \in V$ due to θ_1 in the refresh equation, if and only if the tuple $\square e_1 \in \square E_1$ matches with a tuple $e_1 \in E_1$ due to the parameter θ in the change equation. The matches will result in same update to the E_1 attributes or deletion in both the expressions. Also, a tuple $\square e_1 \in \square E_1$ doesn't find a match in E_1 if and only if the tuples in $(\square e_1 \times E_2) \subseteq \square V$ do not find a match in V.

It is easy to see that the refresh operator in the fourth column with the new specifications is also an `aggregate-refresh` operator.

Join: $V = E_1 \bowtie_J E_2$. Follows from the previous cases, but stated in the table for convenience.

Union: $V = E_1 \uplus E_2$. As $((E_1 \uplus E_2) \dot{-} E_2) = E_1$, the equivalence of the expressions is obvious.

Aggregation: $V = \pi_{G',F}(E_1)$. Here, $F = f_1(A_1), \ldots, f_k(A_k)$. Without loss of generality, we prove this case when $k = 1$, i.e., we assume that $F = f(A)$. In addition, we assume that $(G' \subseteq G), A \in \text{Attrs}(U), U = <(A, f)>$, and that f is a distributive function. Consider tuples $e_1, e_2, \ldots, e_n$ in E_1 such that they have the same G' values and their attribute A values are $a_1, a_2, \ldots, a_n$. Also, assume that the tuple e_i matches with some tuple $\square e_i \in \square E_1$ due to $\mathcal{J}_1$ and that the aggregated attribute A value of $\square e_i$ is $\square a_i$.[12] If e_i doesn't find a match in $\square E_1$, then assume that $\square a_i$ is such that $f(a, \square a_i) = a$ for simplicity of the proof. Note that $\square e_i$'s have the same G' values too. The attribute value a_i of e_i is updated to $f(a_i, \square a_i)$ due to U in the change equation. Thus, due to the change equation the tuple $e_1, e_2, \ldots, e_n$ will result in an aggregated value of $f(f(a_1, \square a_1), f(a_2, \square a_2), \ldots, f(a_n, \square a_n))$ in the equation of the third column. Thus, we need to show that the aggregated A value of the tuple $v \in V = \pi_{G',f(A)}(E_1)$ that is derived from $e_1, \ldots, e_k$ changes from $f(a_1, a_2, \ldots, a_n)$ to $f(f(a_1, \square a_1), f(a_2, \square a_2), \ldots, f(a_n, \square a_n))$ in the refresh equation. By the definition of $\square V$, the tuples $\square e_1, \ldots, \square e_n \in \square E_1$ will be grouped to yield the aggregated attribute value $f(\square a_1, \ldots, \square a_n))$, and the refresh equation of V will change the aggregated value of the grouped value of e_is from $f(a_1, a_2, \ldots, a_n)$ to $f(f(a_1, a_2, \ldots, a_n),\ f(\square a_1, \ldots, \square a_n))$. As f is a distributive function, we have $f(f(a_1, \square a_1), f(a_2, \square a_2), \ldots, f(a_n, \square a_n)) = f(f(a_1, a_2, \ldots, a_n),\ f(\square a_1, \ldots, \square a_n))$, hence the refresh equation of V correctly updates the aggregated attribute value of the tuple v in V.

All insertions into E_1 due to $\square E_1$ in the change equation will be converted to appropriate aggregated insertions or updates into V by the refresh equation. The deletions from E_1 need not necessarily result in any deletions from V. A tuple is deleted from V only if its aggregated attributes become zero, which is independent of the deletions of the deriving tuples from E_1.

[12] If the match is due to $\mathcal{J}_2$, then we need to show that the pair of values a_i and $\square a_i$ is such that $f(a_i, \square a_i, b) = f(b)$ for any b. Once shown, the observation can be used to easily make the rest of the argument go through.

It is easy to see that the refresh operator in the fourth column with the new specifications is also an `aggregate-refresh` operator.

Outerjoin: $V = E_1 \overset{fo}{\bowtie}_J E_2$. Suppose $\Box E_1$ induces a set of insertions I into the relation E_1. Each tuple $i \in I$ results in a set of tuples $E_2^i = i \overset{lo}{\bowtie}_J E_2$ in $\Box V$. No tuple $e_2^i \in E_2^i$ finds a match in V due to the predicate $(\equiv_G \ \wedge \ \equiv_\tau)$, because if it did, i would have found a match in E_1 due to $\equiv_G$. Therefore, the refresh equation results in E_2^i being inserted into V for each $i \in I$.

Let us assume that $M (\subseteq \Box E_1)$ is a set of tuples that find a match in E_1 due to $\equiv_G$, and thus result in update of a tuple in E_1. Each tuple $m \in M$ results in a set of tuples $E_2^m = m \overset{lo}{\bowtie}_J E_2$ in $\Box V$. Note that, E_2^m may consist of just one tuple $<m, \texttt{NULL}>$. If $m \in M$ matches with a tuple $e_1 \in E_1$ due to both having the same G attributes, then each tuple $<m, e_2> \in E_2^m$ would match with the corresponding tuple $<e_1, e_2> \in E_2^{e_1} = e_1 \overset{lo}{\bowtie}_J E_2$. The tuple $<e_1, e_2>$ exists in $E_1 \overset{ro}{\bowtie}_J e_2$, because the pair (m, e_2) satisfies J and $\text{Attrs}(J) \subseteq G$. Note that if E_2^m consists of only $(m, \texttt{NULL})$, then $E_2^{e_1}$ consists of $<e_1, \texttt{NULL}>$ only. Also, as $E_1^{e_2} \subseteq V$, the refresh equation of V affects the updates correctly. As noted before, if $\Box E_1$ results in deletions from E_1, then the refresh equation derived here would need to be modified using an extended refresh operator as illustrated in Section 6.3. ∎

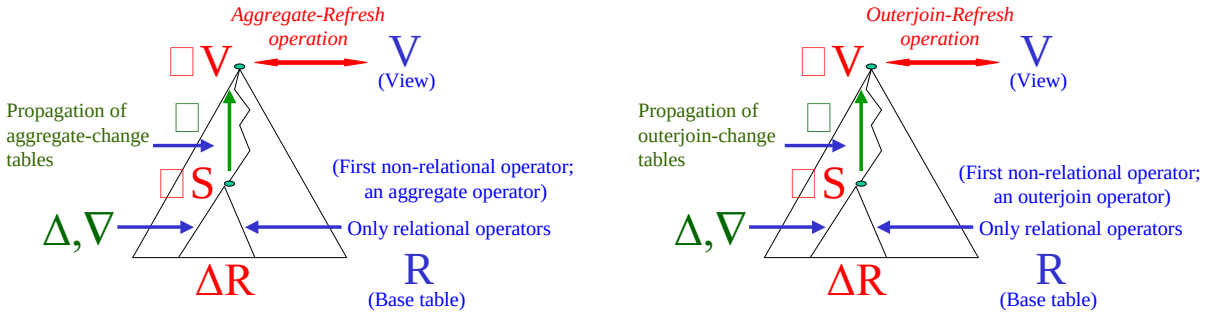
EXAMPLE 4 In this example, we illustrate the techniques developed in this section on the views of Example 1. Recall from Example 1 the definitions of `SISales`, `CitySales`, and `CategorySales`. For clarity of presentation, we use V_1, V_2, and V_3 to denote `SISales`, `CitySales`, and `CategorySales` respectively. Thus, we have

$$\begin{aligned}
V_1 &= \pi_{storeID, itemID, SumSISales=sum(price), NumSISales=count(*)}(\sigma_{date>1/1/95}\texttt{sales}) \\
V_2' &= V_1 \bowtie \texttt{stores} \\
V_2 &= \pi_{city, SumCiSales=sum(SumSISales), NumCiSales=sum(NumSISales)}(V_2') \\
V_3' &= V_1 \bowtie \texttt{items} \\
V_3 &= \pi_{category, SumCaSales=sum(SumSISales), NumCaSales=sum(NumSISales)}(V_3')
\end{aligned}$$

where the (virtual) views V_2' and V_3' have been added for better illustration of how the aggregate-change tables propagate. We use the change propagation equations of Table 4 to derive the maintenance expressions for V_2 and V_3, in response to changes in `sales`, as follows. In all the equations below, U is of the form $<(\texttt{SUM}, f), (\texttt{COUNT}, f)>$ and p is of the form $((LHS.\texttt{COUNT} + RHS.\texttt{COUNT}) = 0)$,[13] where `SUM` is the aggregated attribute (*SumSISales, SumCiSales, or SumCaSales*) in the corresponding view, `COUNT` is the count attribute (*NumSISales, NumCiSales, or NumCaSales*) depending on the view, and $f(x, y) = x + y$ for all x, y.

$$\begin{aligned}
\Box V_1 &= \pi_{storeID, itemID, SumSISales=sum(price), NumSISales=sum(_count)}(\Pi_{storeID, price, _count=1}(\sigma_{date>1/1/95}\triangle\texttt{sales}) \\
&\quad \uplus \ \Pi_{storeID, \ price=-price, \ _count=-1}(\sigma_{date>1/1/95}\triangledown\texttt{sales})) \qquad \text{[From Section 5.1]} \\
V_1 &= V_1 \sqcup_{\theta_1}^U (\Box V_1), \text{ where } \theta_1 \text{ is } (\equiv_{\{storeID, itemID\}} \ \wedge \neg p, \equiv_{\{storeID, itemID\}} \ \wedge p) \\
\Box V_2' &= \Box V_1 \bowtie \texttt{stores} \\
V_2' &= V_2' \sqcup_{\theta_{12}}^U \Box V_2', \qquad \text{[From (4) in Table 4]} \\
&\quad \text{where } \theta_{12} \text{ is } (\equiv_{\{storeID, itemID\} \cup Attrs(\texttt{stores})} \ \wedge \neg p, \equiv_{\{storeID, itemID\} \cup Attrs(\texttt{stores})} \ \wedge p) \\
\Box V_2 &= \pi_{city, SumCiSales=sum(SumSISales), NumCiSales=sum(NumSISales)}(\Box V_2') \\
V_2 &= V_2 \sqcup_{\theta_2}^U \Box V_2, \text{ where } \theta_2 \text{ is } (\equiv_{city} \ \wedge \neg p, \equiv_{city} \ \wedge p) \qquad \text{[From (6) in Table 4]}
\end{aligned}$$

[13] Recall that LHS and RHS refer to the left and right operands of the join operation where p occurs.



(a) Aggregate-change Tables

(b) Outerjoin-change Tables

Figure 3: Generation and Propagation of Change-Tables.

$$
\begin{aligned}
\square V_3' &= \square V_1 \bowtie \texttt{items} \\
V_3' &= V_3' \sqcup^U_{\theta_{13}} \square V_3', && \text{[From (4) in Table 4]} \\
&\quad \text{where } \theta_{13} \text{ is } (\equiv_{\{storeID, itemID\} \cup Attrs(\texttt{items})} \wedge \neg p,\ \equiv_{\{storeID, itemID\} \cup Attrs(\texttt{items})} \wedge p) \\
\square V_3 &= \pi_{category, SumCaSales = sum(SumSISales), NumCaSales = sum(NumSISales)}(\square V_3') \\
V_3 &= V_3 \sqcup^U_{\theta_3} \square V_3, \text{ where } \theta_3 \text{ is } (\equiv_{category} \wedge \neg p,\ \equiv_{category} \wedge p) && \text{[From (6) in Table 4]}
\end{aligned}
$$

As shown in Example 1, the above derived maintenance expressions for V_2 and V_3 are very efficient compared to the expressions derived by previous approaches. □

Overall Change-Table Technique. The overall technique of change-table incremental maintenance works as follows. In a view expression tree, the insertions and deletions at the base tables are propagated through the initial relational operators using techniques in [GL95]. If the first non-relational (aggregate or outerjoin) operator encountered during propagation of insertions/deletions is an aggregate operator as shown in Figure 3 (a), then the insertions and deletions are used to generate an aggregate-change table at that aggregate operator, and the aggregate-change table is propagated through various operators using the techniques developed in this section. On the other hand, if the first non-relational operator encountered is an outerjoin operator as shown in Figure 3 (b), then an outerjoin-change table is generated and propagated through various operators using the techniques developed in Section 6. Thus, the choice of whether an aggregate-change table or outerjoin-change table is used for the incremental maintenance process depends on the operator that is first encountered in the view expression tree during propagation of insertions/deletions from the base tables to the root of the expression tree.

6 Maintaining Outerjoin Views Efficiently

In this section, we show how our change-table technique can be used to derive efficient and simple algebraic expressions for maintenance of view expressions involving outerjoin operators. Outerjoin is supported in SQL. Further, outerjoins have recently gained importance because data from multiple distributed databases can be integrated by means of outerjoin views [GJM96, GM95, RU96]. Outerjoins are also extensively used in object-relational systems [BW89, BW90, BPP+93].

Definition 3 (**Outerjoin-change table**) A change table for a view involving outerjoin operators is defined as an *outerjoin-change table* if the change table was either generated at an outerjoin operator or is a result of propagation of an outerjoin-change table, using the propagation equations we will derive for propagating outerjoin-change tables.

For example, the change tables, $\Box$`SSInfo` and $\Box$`SSFullInfo`, computed for the views `SSInfo` and `SSFullInfo` respectively in Example 2 are outerjoin-change tables. □

We start by showing how the changes to an outerjoin view ($R \overset{fo}{\bowtie}_J S$), in response to insertions into the base table R, can be summarized into an outerjoin-change table. Computation of an outerjoin-change table at an outerjoin view in response to deletions from a base table requires a more general `refresh` operator and is briefly discussed in Section 6.3.

6.1 Generating Outerjoin-Change Table At An Outerjoin Node

Given tables $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_m)$, consider an outerjoin view $V(A_1, \ldots, A_n, B_1, \ldots, B_m) = R \overset{fo}{\bowtie}_J S$, where J is an equi-join condition. Insertions into R, $\triangle R$, can result in some insertions and deletions into view V. We summarize the set of changes to V in an outerjoin-change table $\Box V$ defined as $\Box V = \triangle R \overset{lo}{\bowtie}_J S$. Note that the tables $\Box V$ and V have the same schema and attribute names. We show that with the following specification of the `refresh` operator, the net changes in the outerjoin-change table $\Box V$ can be applied to the view V to obtain the correctly refreshed V. The specifications, $\theta = (\mathcal{J}_1, \mathcal{J}_2)$ and U, of the `refresh` operator used to apply $\Box V$ to V are:

- $\mathcal{J}_1$ is $(\equiv_G \ \wedge p)$, where $G =$ Attrs(S) and $p = (\bigwedge_{1 \le j \le n}(V.A_j = \texttt{NULL}))$.
- $\mathcal{J}_2$ is `FALSE`.
- The update list U is $<(A_1, f), (A_2, f), \ldots, (A_n, f)>$, where $f(x, y) = y$ for all x, y.

Theorem 2 *Consider the view $V = R \overset{fo}{\bowtie}_J S$ and the outerjoin-change table $\Box V = \triangle R \overset{lo}{\bowtie}_J S$. For the above definition of the* `refresh` *operator specifications of $\theta = (\mathcal{J}_1, \mathcal{J}_2)$ and U, the following holds:*

$$(R \uplus \triangle R) \overset{fo}{\bowtie}_J S = (R \overset{fo}{\bowtie}_J S) \sqcup_\theta^U (\Box V)$$

Proof: Due to insertion of $\triangle R$ into R, the view V should change as follows. First, the set of tuples $\triangle R \overset{lo}{\bowtie}_J S$ should be inserted to V. Then, if there is a tuple $\triangleright v = <r_1, r_2, \ldots, r_n, s_1, s_2, \ldots, s_m>$ in $(\triangle R \overset{lo}{\bowtie}_J S)$, i.e., if $\triangleright v$ is being inserted into V, then the tuple $<\texttt{NULL}, \ldots, \texttt{NULL}, s_1, s_2, \ldots, s_m> \in V$ should be deleted from V.

The above effect can be achieved by updating a tuple $v = <\texttt{NULL}, \ldots, \texttt{NULL}, s_1, s_2, \ldots, s_m>$ in V to $\triangleright v = <r_1, r_2, \ldots, r_n, s_1, s_2, \ldots, s_m>$ if such a tuple $\triangleright v$ exists in $\Box V = \triangle R \overset{lo}{\bowtie}_J S$. The refresh of V would be complete if the tuples $\triangleright v$ in $\Box V$ for which no such match occurs are inserted into V. By the definition of the `refresh` operator and its specification, one can see that this is exactly what is achieved by the refresh expression $V \sqcup_\theta^U \Box V$. ∎

6.2 Propagating Outerjoin-Change Tables

Only a special form of the generic `refresh` operator, which we call an `outerjoin-refresh` operator, is required to refresh a view using its outerjoin-change table.

Definition 4 (`Outerjoin-refresh` **operator**) Let $\{A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m\}$ be the set of attributes in V and its outerjoin-change table $\Box V$. A `refresh` operator $\sqcup_\theta^U$ used to apply the outerjoin-change table $\Box V$ to its view V is said to be an `outerjoin-refresh` operator if:

- the join condition $\mathcal{J}_1$ is $(\equiv_G \ \wedge p)$, where $G = \{B_1, B_2 \ldots, B_m\}$ and p is a predicate on the attributes $(LHS.A_1, LHS.A_2, \ldots, LHS.A_n)$.
- the join condition $\mathcal{J}_2$ is `FALSE`, and
- the update list U is $<(A_1, f), (A_2, f), \ldots, (A_n, f)>$, where $f(x, y) = y$ for all x, y. Note that $(\text{Attrs}(U) \cap G) = \phi$ and $(\text{Attrs}(U) \cup G) = \text{Attrs}(V)$. □

The refresh equations given in Table 4 correctly propagate an outerjoin-change table as well, except for the case of propagation through the outerjoin operator, for which we derive a different equation below.

Consider a view $V = E_1 \overset{fo}{\bowtie}_J E_2$, where E_1 and E_2 are general view expressions. Suppose that the expression E_1 changes to to $E_1 \sqcup_\theta^U \square E_1$ using its outerjoin-change table $\square E_1$, where the refresh operator $\sqcup_\theta^U$ is an `outerjoin-refresh` operator. Let θ be $(\equiv_G \ \wedge p, \texttt{FALSE})$, where p is a predicate, and G is a set of attributes common to E_1 and $\square E_1$. The following row, which replaces the row (7) in Table 4, shows how to propagate an outerjoin-change table $\square E_1$ through the outerjoin operator.

7b	$E_1 \overset{fo}{\bowtie}_J E_2$	$(E_1 \sqcup_\theta^U \square E_1) \overset{fo}{\bowtie}_J E_2$	$V \sqcup_{\theta_1}^{U_1} (\square E_1 \overset{lo}{\bowtie}_J E_2)$	$(\square E_1 \overset{lo}{\bowtie}_J E_2)$	$\text{Attrs}(J) \subseteq G$

As already mentioned, θ is $(\equiv_G \ \wedge p, \texttt{FALSE})$ in the row above. Also,

- $\theta_1 = (\mathcal{J}_1, \texttt{FALSE})$, where $\mathcal{J}_1$ is $\equiv_{Attrs(E_2)} \ \wedge ((\equiv_G \ \wedge p) \vee (\bigwedge_{e_1 \in Attrs(E_1)} LHS.e_1 = \texttt{NULL}))$, and
- $U_1 = <(\mathcal{A}_1, f), (\mathcal{A}_2, f), \ldots, (\mathcal{A}_k, f)>$, where $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k\} = \text{Attrs}(E_1)$ and $f(x, y) = y$ for all x, y.

Theorem 3 *Assume that the* `refresh` *operator used in the expression of the third column in Table 4 is an* `outerjoin-refresh` *operator.*

(1) The change propagation equations given in Table 4, with the following two changes, correctly propagate outerjoin-change tables.

- *Disregard the condition in column 6 of the first row (selection view)*
- *Replace the seventh row by row (7b) given above*

(2) The `refresh` *operator derived in each of the refresh equations (column 4) is also an* `outerjoin-refresh` *operator, except for the case of propagation through an aggregate operator (sixth row) where the derived operator is an* `aggregate-refresh` *operator.*

Proof: It is easy to see that the refresh operator in the fourth column with its new specifications is also an `outerjoin-refresh` operator, except for in the *sixth* row where the derived operator is an `aggregate-refresh` operator. As before, we refer to the expression $E_1 \sqcup_\theta^U \square E_1$ as the *change equation* throughout this proof.

Selection: $V = \sigma_q(E_1)$. We use the characteristics of the `outerjoin-refresh` operator to show that $\sigma_q(E_1 \sqcup_\theta^U \square E_1) = \sigma_q(E_1) \sqcup_\theta^U \sigma_q(\square E_1)$.

First, we show that the refresh equation of V doesn't miss any legitimate insertions into V affected by the change equation $E_1 \sqcup_\theta^U \square E_1$. Let I be the set of tuples that is inserted into E_1 due to the change equation. We will show that $\sigma_q(I)$ is inserted into V by the refresh equation, implying the result. Note that I is the set of tuples in $\square E_1$ that do not find a match in E_1. As $\sigma_q(I) \subseteq \ \sigma_q(\square E_1) = \square V$, no tuple in $\sigma_q(I)$ will find a match in $V \subseteq E_1$. Hence, $\sigma_q(I)$ will be inserted into V using the refresh equation.

Now, we show that all insertions into V due to the refresh equation are legitimate. The refresh equation may induce an insertion of the tuple $\square v \in \square V$ into V if $\square v$ doesn't find a match in V. If $\square v \in I$, where I is the set of tuples in $\square E_1$ that don't find a match in E_1, then the insertion is obviously legitimate. Suppose, $\square v \notin I$. This implies that $\square v \in \square E_1$ found a match with a tuple $e_1 \in E_1$. Because of the specifications of the `outerjoin-refresh` operator, the match results in the tuple e_1 in E_1 being updated to $\square v$ by the change equation $E_1 \sqcup_\theta^U \square E_1$. Now note that as $\square v \in \square V$, it satisfies the selection condition q, and hence the tuple

$\square v$ (updated from e_1 in E_1) will be included in the expression $\sigma_q(E_1 \sqcup_\theta^U \square E_1)$ of the third column. Thus, the insertion of $\square v$ into V by the refresh equation is correct. As the match between tuples is one-to-one, the tuple $\square v$ results in only one update in the table E_1 due to the change equation, which as shown above corresponds to the insertion of $\square v$ into V due to the refresh equation.

In the refresh equation, if a tuple $v \in V$ is updated by a tuple $\square v \in \square V = \sigma_q(\square E_1)$, then $v \in E_1$ would have been updated by $\square v \in \square E_1$ by the change equation too. According to the update characteristics of the `outerjoin-refresh` operator, the tuple v is updated to $\square v$ by $\square v$. Now, $\square v$ satisfies the predicate p, hence the updated tuple $\square v$ is correctly retained in V by the refresh equation. This shows that the updates to V in the refresh equation are legitimate.

The only updates the refresh equation might miss are of the kind where a tuple $\square e_1 \in \square E_1$ matches with a tuple $e' \in \sigma_{\overline{q}}(E)$ in the change equation. The match results in the tuple e' being updated to $\square e_1$. If $\square e_1$ satisfies the condition p, then it is included in V by the change equation. In the refresh equation, as $\square e_1 \in \square V$ doesn't find a match in V, it is inserted into V. Hence, the desired effect is achieved.

Projection: $V = \Pi_A(E_1)$. If $\text{Attrs}(\theta) \subseteq A$, then the information needed to decide the effect of a tuple $\square e_1 \in \square E_1$ on a tuple $e_1 \in E_1$, if any, is available in $\Pi_A(e_1)$. Hence, $\Pi_A(\square E_1)$ can be applied directly to $V = \Pi_A(E_1)$.

Cross Product: $V = E_1 \times E_2$. Consider $(E_1 \times E_2) \sqcup_{\theta_1}^U (\square E_1 \times E_2)$, the refresh equation. Let us partition the tables $(E_1 \times E_2)$ and $(\square E_1 \times E_2)$ by the tuple values of E_2. As $\mathcal{J}_1$ and $\mathcal{J}_2$ in θ_1 include $\text{Attrs}(E_2)$, each of the partitions is refreshed independently by the refresh equation. It is easy to see that a tuple $<\square e_1, e_2> \in \square V$ will result in a match with a tuple $<e_1, e_2> \in V$ due to θ_1 in the refresh equation, if and only if the tuple $\square e_1 \in \square E_1$ matches with a tuple $e_1 \in E_1$ due to the parameter θ in the change equation. The match will result in the same updates to the E_1 attributes in both the expressions. Also, a tuple $\square e_1 \in \square E_1$ doesn't find a match in E_1 if and only if the tuples in $(\square e_1 \times E_2) \subseteq \square V$ do not find a match in V.

Join: $V = E_1 \bowtie_J E_2$. Follows from the previous cases.

Union: $V = E_1 \uplus E_2$. As $((E_1 \uplus E_2) \dot{-} E_2) = E_1$, the equivalence of the expressions is obvious.

Aggregation: $V = \pi_{G', f(A)}(E_1)$. Without loss of generality, we prove this case when $k = 1$. Let $f_1 = f$ and $A_1 = A$, the aggregated attribute. We assume that $(G' \subseteq G)$, $A \in \text{Attrs}(U)$, $U = <(A, f)>$, and that f is a distributive function. Consider tuples $e_1, e_2, \ldots, e_n$ in E_1 such that they have the same G' values, and let their attribute A values be $a_1, a_2, \ldots, a_n$. Assume that the tuples $e_1, \ldots, e_l$ find matches with tuples $\square e_1, \ldots, \square e_l$ in $\square E_1$ due to the join condition $\mathcal{J}_1$. Thus, $a_i =$ `NULL` and $\square e_i$'s have the same G' values as e_i's for $1 \leq i \leq l$. Also, the attribute A value of e_i is updated to $\square a_i$, for $1 \leq i \leq l$, due to the update parameter U in the change equation. Hence, due to the change equation the tuples $e_1, e_2, \ldots, e_n$ will result in an aggregated value of $f(\square a_1, \square a_2, \ldots, \square a_l, a_{l+1}, a_{l+2}, \ldots, a_n)$ in the expression of the third column. Thus, we need to show that the aggregated A value of the tuple v that is derived from $e_1, \ldots, e_n$ in the view $V = \pi_{G', f(A)}(E_1)$ changes from $f(a_1, a_2, \ldots, a_n)$ to $f(\square a_1, \square a_2, \ldots, \square a_l, a_{l+1}, a_{l+2}, \ldots, a_n)$ in the refresh equation. Now, by the definition of $\square V$, the tuples $\square e_1, \ldots, \square e_l \in \square E_1$ will be grouped to yield the aggregated attribute value $f(\square a_1, \ldots, \square a_l)$ in a tuple in $\square V$. The refresh equation of V then updates the aggregated value of the grouped value of e_is from $f(a_1, a_2, \ldots, a_k)$ to $f(f(a_1, a_2, \ldots, a_n), f(\square a_1, \ldots, \square a_l)) = f(a_{l+1}, \ldots, a_n, \square a_1, \ldots, \square a_l)$, as a_1 to a_l are `NULL`. Hence the refresh equation of V correctly updates the aggregated attribute value of the tuple v in V.

All insertions into E_1 due to $\square E_1$ will be converted to aggregated insertions into V by the refresh equation.

Outerjoin: $V = E_1 \overset{fo}{\bowtie}_J E_2$. Let $M(\subseteq \square E_1)$ be the set of tuples in $\square E_1$ that find a match in E_1 in the change equation due to $\mathcal{J}_1 = (\equiv_G \wedge p)$. Note that G is a set of attributes in E_1 and p is a predicate over the rest of the attributes in E_1. Consider $m \in M$ and that m finds a match with a tuple e_1 in E_1 due to $\mathcal{J}_1$. Each m results in a set of tuples $E_2^m = m \overset{lo}{\bowtie}_J E_2$ in $\square V = \square E_1 \overset{fo}{\bowtie}_J E_2$, where E_2^m may consist of just one tuple $<m,$ `NULL`$>$. Each tuple $e_2^m = <m, e_2> \in E_2^m$ will find a match with the corresponding tuple $v = <e_1, e_2> \in V$

(note that e_2 may be NULL) due to the join condition $\mathcal{J}_1 \wedge \equiv_{Attrs(E_2)}$. As Attrs($J$) $\subseteq G$ and $<m, e_2> \in E_2^m$, the tuple v exists in V (even if e_2 = NULL). The tuple $<e_1, e_2>$ gets updated to $<m, e_2>$ due to the update list U_1 in the refresh equation of V. This update affected in V by the refresh equation is correct because $m \in \square E_1$ also updates $e_1 \in E_1$ to m in the change equation, as Attrs(G) $\cup$ Attrs(U) = Attrs(E_1), and m and e_1 have the same G attributes. Thus, all the updates of tuples in E_1 due to the change equation are correctly applied to V by $\square V$ in the refresh equation.

Lets consider the other set of tuples I in $\square E_1$, that are inserted into E_1 (because they didn't find a match in E_1). Let $i \in I$. The tuple i results in the set of tuples $E_2^i = i \stackrel{lo}{\bowtie}_J E_2$ in $\square V$. Tuple $<i, e_2> \in E_2^i$ may find a match v in V due to the condition $\equiv_{Attrs(E_2)} \wedge (\bigwedge_{e \in Attrs(E_1)} LHS.e = \texttt{NULL})$ in $\mathcal{J}_1$. In that case the tuple v is correctly updated to $<i, e_2>$. All the other unmatched tuples in $\square V$ are correctly inserted into V. ∎

As the propagation of an outerjoin-change table through an aggregate operator results in an `aggregate-refresh` operator, the resulting change table is an *aggregate*-change table. This is an exception to Definitions 1 and 3, not mentioned earlier to avoid confusion.

EXAMPLE 5 Consider a view $V = \pi_{A,B,F=sum(D),H=sum(E),Num=Count(*)}(\sigma_{A>5}((R \stackrel{fo}{\bowtie}_{C=D} S) \bowtie T))$, where $R(A, B, C), S(D, E)$, and $T(A, B, L)$ are base relations, and $\bowtie$ is the natural join operation, i.e., a join with the join condition ($\equiv_{\{A,B\}}$). Recall that for computing the SUM aggregates, the attribute value of NULL is taken as 0, provided at least one tuple has a non-NULL value.

For clarity of presentation, assume that $V_1 = R \stackrel{fo}{\bowtie}_{C=D} S$, $V_2 = V_1 \bowtie T$, $V_3 = \sigma_{A>5}(V_2)$. Let us define a predicate p as $((LHS.D = \texttt{NULL}) \wedge (LHS.E = \texttt{NULL}))$, a predicate q as $((LHS.Num + RHS.Num) = 0)$, an update list U_1 as $<(F, f), (H, f)>$, and U as $<(D, g), (E, g), (Num, g)>$. Here, $f(x, y) = y$ for all x, y, $g(x, y) = x + y$ for all $x, y \neq$ NULL, and $g(\texttt{NULL}, y) = y$. We illustrate our techniques of maintaining views involving outerjoin operators by deriving maintenance expressions for V in response to insertions, $\triangle S$, into S. The first equation used for computing $\square V_1$ is similar to that derived in Theorem 2. In this case, we have insertions into S and hence, we use a right outerjoin operation instead.

$$
\begin{aligned}
\square V_1 &= (R \stackrel{ro}{\bowtie}_{C=D} \triangle S) \\
V_1 &= V_1 \sqcup_{\theta_1}^{U_1} \square V_1, \text{ where } \theta_1 \text{ is } (\equiv_{Attrs(R)} \wedge p, \texttt{FALSE}) && \text{[From Theorem 2]} \\
\square V_2 &= \square V_1 \bowtie T \\
V_2 &= V_2 \sqcup_{\theta_2}^{U_1} \square V_2, \text{where } \theta_2 \text{ is } (\equiv_{Attrs(R) \cup Attrs(T)} \wedge p, \texttt{FALSE}) && \text{[From (4) in Table 4]} \\
\square V_3 &= \sigma_{A>5}(\square V_2) \\
V_3 &= V_3 \sqcup_{\theta_2}^{U_1} \square V_3 && \text{[From (1) in Table 4]}
\end{aligned}
$$

Now, in order to propagate the outerjoin-change table $\square V_3$ through an aggregate operator, we need to replace function f by g in the update parameter $U_1 = <(F, f), (H, f)>$ of the above refresh equation. Actually, it is valid to replace f by g in U_1, because an outerjoin-change table results in an update of *only* NULL values. Hence, f in U_1 could be restricted to having the first parameter as NULL, in which case f behaves exactly as g. Note that g is essentially the SUM aggregate function extended for NULL input values. Thus, if $U_2 = <(F, g), (H, g)>$:

$$
\begin{aligned}
V_3 &= V_3 \sqcup_{\theta_2}^{U_2} \square V_3 \\
\square V &= \pi_{A,B,F=Sum(D),H=Sum(E),Num=Count(*)}(\square V_3) \\
V &= V \sqcup_{\theta}^{U} \square V, \text{where } \theta \text{ is } (\equiv_{\{A,B\}} \wedge \neg q, \equiv_{\{A,B\}} \wedge q) && \text{[From (6) in Table 4]}
\end{aligned}
$$

A similar analysis allows propagation of outerjoin-change tables through other aggregate functions. Note that $\square V$ is an *aggregate*-change table, as $\sqcup_{\theta}^{U}$ is an `aggregate-refresh` operator. Hence, propagation of $\square V$ (for

views that use V as a subview) would be in accordance with change propagation equations for aggregate-change tables (Theorem 1). □

6.3 Propagation of Deletions through Outerjoin Operators

The changes in an outerjoin $V = R \overset{fo}{\bowtie}_J S$ due to deletions from a base relation R cannot be summarized in an outerjoin-change table within our restricted definition of the `refresh` operator. In this section, we show that by using an extended definition of the `refresh` operator, we can apply changes summarized in an appropriately defined change table to V in response to deletions from a base table.

Consider a simple outerjoin view $V = R \overset{fo}{\bowtie}_J S$. Let $\bigtriangledown R$ be the set of deletions from R, and let $S^{set} = \pi_{Attrs(S),Num=Count(*)}(S)$. We define the *OJDeletion-change table* $\square V$ that succinctly represents changes to V in response to $\bigtriangledown R$ as

$$\square V = \bigtriangledown R \overset{lo}{\bowtie}_J S^{set}.$$

The view V is refreshed using the refresh equation $V \sqcup_\theta^U \square V$, where $\sqcup_\theta^U$ is the `OJDeletion-refresh` operator as defined in Algorithm 1. Here, $\theta = (\equiv_{Attrs(V)}, (\equiv_{Attrs(S)} \wedge \text{Attrs}(S) \neq \texttt{NULL}))$ and $U = <(A_1, f), \ldots, (A_n, f)>$, where $f(x, y) = \texttt{NULL}$ for all x, y and $\{A_1, \ldots, A_k\} = \text{Attrs}(R)$.

The OJDeletion-refresh algorithm (Algorithm 1) used to refresh the view V works as follows. A tuple (r, s, l) in $\square V$ comes from l copies of s in S and a tuple r in $\bigtriangledown R$. Thus, the tuple r belonged to R before deletions and V has at least l copies of (r, s). Now, if there is another tuple $v' = (r_1, s) \in V$, then the l copies of (r, s) can be deleted from V as a result of deletion of r from R. But, if no such v' exists in V, then each of the copies of (r, s) in V should be changed to $(\texttt{NULL}, s)$, as the tuple $s \in S$ now becomes a dangling tuple after deletions from R. Thus, the l copies of (r, s) are updated accordingly in Algorithm 1. One can see that it is essential to store in $\square V$ the number of copies l of s from S. Also, note that the OJDeletion-refresh algorithm doesn't need to query any sources, and hence, can be executed very efficiently.

Algorithm 1 **OJDeletion-Refresh Algorithm**

Used to apply an OJDeletion-change table to its view

Input

View $V(A_1, \ldots, A_n, B_1, \ldots, B_m)$

OJDeletion-change Table $\square V(A_1, \ldots, A_n, B_1, \ldots, B_m, Num)$

Characteristics of the OJDeletion-refresh Parameters

$\theta = (\mathcal{J}_1, \mathcal{J}_2).\mathcal{J}_1$ is $\equiv_{Attrs(V)}$ and $\mathcal{J}_2$ is $\equiv_G$, where $G = \{B_1, B_2, \ldots, B_m\}$

$U = <(A_1, f_1), (A_2, f_2), \ldots, (A_n, f_n)>$.

Output

Refreshed table V, i.e., $V \sqcup_\theta^U \square V$.

BEGIN

for each tuple $\square v = (r, s, l)$ in $\square V$ /* r is the value of U attributes, */
/* s is the value of the G attributes, and l is an integer. */

Let $\{v_1, \ldots, v_l\}$ be the tuples in V that match $\square v$ due to the join condition $\mathcal{J}_1$.
/* Note that there are at least l tuples in V that will match $\square v$ due to $\mathcal{J}_1$. */

if there is a tuple $v' \in V$ such that $v' \notin \{v_1, \ldots, v_l\}$ and
v' matches with $\square v$ due to the join condition $\mathcal{J}_2$

then Delete tuples $v_1, v_2, \ldots, v_l$ from V;

else Update each tuple $v_1, v_2, \ldots, v_l$ in V using the specifications in U;

end if;

end for;

No.	V	New V $\theta = (\mathcal{J}_1, \mathcal{J}_2)$ $\mathcal{J}_1$ is $\equiv_V$ $\mathcal{J}_2$ is $\equiv_G$	Refresh Equation	$\square V$	Conditions
1	$\sigma_q(E_1)$	$\sigma_q(E_1 \sqcup_\theta^U \square E_1)$	$V \sqcup_\theta^U \sigma_q(\square E_1)$	$\sigma_q(\square E_1)$	$\text{Attrs}(q) \subseteq G$
2	$\Pi_A(E_1)$	$\Pi_A(E_1 \sqcup_\theta^U \square E_1)$	$V \sqcup_\theta^U \Pi_A(\square E_1)$	$\Pi_A(\square E_1)$	$G \subseteq A$
3	$E_1 \times E_2$	$(E_1 \sqcup_\theta^U \square E_1) \times E_2$	$V \sqcup_{\theta_1}^U (\square E_1 \times E_2)$ $\theta_1 = (\mathcal{J}_1 \wedge \equiv_\tau, \mathcal{J}_2 \wedge \equiv_\tau)$ $\tau = \text{Attrs}(E_2)$	$\square E_1 \times E_2$	
4	$E_1 \bowtie_J E_2$	$(E_1 \sqcup_\theta^U \square E_1) \bowtie_J E_2$	$V \sqcup_{\theta_1}^U (\square E_1 \bowtie_J E_2)$ $\theta_1 = (\mathcal{J}_1 \wedge \equiv_\tau, \mathcal{J}_2 \wedge \equiv_\tau)$ $\tau = \text{Attrs}(E_2)$	$\square E_1 \bowtie_J E_2$	$\text{Attrs}(J) \subseteq G$
5	$E_1 \uplus E_2$	$(E_1 \sqcup_\theta^U \square E_1) \uplus E_2$	$((V \dot{-} E_2) \sqcup_\theta^U \square E_1) \uplus E_2)$	$\square E_1$	
6	$E_1 \overset{fo}{\bowtie}_J E_2$	$(E_1 \sqcup_\theta^U \square E_1) \overset{fo}{\bowtie}_J E_2$	$V \sqcup_{\theta_1}^U (\square E_1 \overset{lo}{\bowtie}_J E_2)$ $\theta_1 = (\mathcal{J}_1 \wedge \equiv_\tau, \mathcal{J}_2 \wedge \equiv_\tau)$ $\tau = \text{Attrs}(E_1)$	$(\square E_1 \overset{lo}{\bowtie}_J E_2)$	$\text{Attrs}(J) \subseteq G$

Table 5: Change propagation equations for propagating OJDeletion-change tables

return V;
END. ◇

Table 5 presents change propagation equations that are used to propagate an OJDeletion-change table through various operators. The proof of Theorem 4 is omitted and can be found in [Gup00].

Theorem 4 *Assume that the* `refresh` *operator used in the expression of the third column in Table 5 is an* `OJDeletion-refresh` *operator. Then,*

(1) the change propagation equations given in Table 5 for propagation of OJDeletion-change tables are correct, i.e., for each row, the expression in the third column is equivalent to the refresh equation in the fourth column, and

(2) the `refresh` *operator derived in the refresh equation (column 4) is an* `OJDeletion-refresh` *operator as well.* □

7 Optimality Issues

In this section, we discuss the optimality of our change-table incremental maintenance algorithm. In particular, we show that our developed techniques result in the minimum number of sources (base relations) being queried. Note that in a data warehouse, the dominant cost is the cost incurred in querying the sources.

Let us consider change-table techniques presented in this article for incremental maintenance of general view expressions along with the following two minor optimizations:

- In computing a view, we tag the tuples in the result of a union operator with L/R depending on whether the tuple comes from the left operand or the right operand. The above improvement makes propagation of an aggregate and outerjoin-change table through a union operator very efficient. In essence, the refresh equation of the fifth row in Table 4 will not involve E_1 or E_2.
- When computing the refresh equation, the view contents are used, whenever possible, to compute a required subexpression using minimum number of source queries. For e.g., consider $V = R \times (S \times T)$. In response to insertions into R, to compute changes at V, we need to query $(S \times T)$ according to the propagation equations in Table 4. Instead, we query R and compute $(S \times T)$ using the value of V, minimizing the number of source queries.

We incorporate the above two improvements into our change-table techniques and prove the following result (see [Gup00] for the proof).

Theorem 5 *Given an expression tree of a view V, the change-table technique queries the minimum number of sources required in order to compute changes at* each *node in the expression tree.* □

We define an algorithm to be a change-propagating algorithm if it computes changes (in some form) at each node in the given expression tree of the view. The above theorem shows that our change-table technique is better than any change-propagating maintenance algorithm for a given expression tree (even in the presence of singularity points). Although it is not necessary for an incremental maintenance algorithm to be change-propagating, all the previously proposed maintenance algorithms fall in this category.

An interesting open problem is to design a provably optimal incremental maintenance algorithm under the above cost model for maintenance of general view expressions (without restricting ourselves to the class of change-propagating algorithms). We have focussed our recent work on adapting the maintenance algorithm based on our change-table techniques to obtain an optimal approach. We conjecture that the change-table technique with some minor improvements/optimizations can be translated into an optimal incremental maintenance algorithm under the above cost model.

8 Related Work

A large body of work exists describing different algorithms for incrementally maintaining materialized views [BLT86, RK86, Han87, BCL89, CW91, QW91, GMS93, GLT94, GL95, CGL$^+$96, GJM97, Qua97, GK98, LVM99, LV01, KR02, ESWDR02, PSCP02]. Each work applies to different classes of views and has various advantages and disadvantages. Below, we discuss some of the above algorithms.

Qian and Wiederhold in [QW91] present a technique (later corrected in [GLT94]) to propagate sets of insertions and deletions through relational algebra operators without duplicates. The techniques in [QW91] were extended to bag algebra by Griffin and Libkin in [GL95]. While [QW91] did not deal with aggregates at all, [GL95] did consider aggregates when they are applied as the last operator in an expression, and when there are no groupby columns. The work of [GL95] was further extended by Quass [Qua97] to include general expressions involving aggregation. However, as illustrated in Section 1, the technique of [Qua97] works with insertions and deletions, and is thereby less efficient, and more complex, than the change table technique presented in this paper. None of [QW91, GL95, Qua97] can deal with outerjoins.

Gupta et al. [GMS93] also present algorithms for incrementally maintaining views with duplicates. Aggregation is considered only for the case where a view is materialized at each aggregation node. Mumick et al. in [MQM97] give an algorithm for efficiently maintaining a set of summary tables, where a summary table is the result of applying a single aggregation over an SPJ expression over star schema tables in a data warehouse.

Thus, [MQM97] does not consider general view expressions involving aggregate operators. In this article, we have generalized the concept of summary tables to our concept of aggregate-change tables to facilitate propagation through various relational, aggregation, and outerjoin operators. Palpanas et al. in [PSCP02] extend the above works ([GMS93, MQM97, Qua97]) by including non-distributive aggregate functions and optimizing maintenance by efficiently recomputing only the set of affected groups. However, the views considered in [PSCP02] are restricted to view expressions that have at most one aggregate operator as the last operator. Also, views are not permitted to have duplicate rows. Based on some of the above techniques, Oracle 8.1 [BDD+98] and DB2 [LSPC00] support incremental maintenance for views that are formed by a single aggregate over an equi-join view, but support only full refresh mode (i.e., complete recomputation) for general view expressions involving aggregate operators. None of the above described works ([GMS93, MQM97, PSCP02]) considers outerjoin operators in view expressions.

Gupta et al. in [GJM97] present maintenance and self-maintenance algorithms to compute the incremental changes to a materialized outerjoin view $R \overset{fo}{\bowtie} S$, where R and S are base tables. The algorithms presented in [GJM97] are procedural rather than algebraic, and do not apply to general view expressions containing outerjoin operators. Similarly, Oracle 8.1 [BDD+98] supports incremental maintenance of equi-outerjoin views, but supports only complete recomputation for general view expressions involving outerjoin operators. In work done concurrently with ours, Griffin and Kumar in [GK98] extend the techniques of [GJM97] by deriving propagation equations through outerjoin operators. As [GK98] propagates changes in the form of insertions and deletions, their incremental algorithm is less efficient than our change-table techniques for general view expressions, as illustrated in Example 1.

The problem of incremental view maintenance is closely related to the problem of self-maintainability of views [GJM94, QGMW96, GJM97]. A view is defined as self-maintainable with respect to certain kinds of changes if the view can be updated using the old view value and the changes, without accessing any base relations. The change-table technique presented in this articles, in most cases, derives maintenance expressions that do not refer to the base tables, even when such self-maintenance expressions were not possible using only insertions and deletions as types of updates. Hence, the techniques presented in this article help in deriving efficient self-maintenance expressions.

In other related works, Ali et al. [AFP00] solve the incremental maintenance problem for a large class of views expressed in Object Query Language (OQL). Liu and Vincent in [LVM99, LV01] derive incremental expressions for the general nested relational model which is used in warehouses and in XML data. El-Sayed et al. in [KR02, ESWDR02] present an algebraic approach for incremental maintenance of materialized views expressed in XQuery, an XML query language.

9 Conclusions

In this article, we have developed a change-table technique for incremental maintenance of general view expressions involving aggregate and outerjoin operators. To the best of our knowledge, this is the first paper to present algebraic expressions for maintaining general views involving aggregate and outerjoin operators.

Traditional maintenance techniques [QW91, GL95, Qua97] propagate insertions and deletions from the base relations to the view through each of its operators. In contrast, we compute change tables at an aggregate or outerjoin operator, and use change propagation equations to propagate the change tables through the relational, aggregate and outerjoin operators. We show that the changes represented in change tables can be applied to its corresponding materialized view using an appropriately defined refresh operator. The resulting maintenance expressions for general view expressions are simple and very efficient compared to previous techniques.

The change-table technique presents a new paradigm for view maintenance using change tables. The maintenance expressions derived by the change-table techniques are usually self-maintenance expressions, as they usually refer only to the view and the changes to the base tables, minimizing the number of queries to the base tables. Such a paradigm is likely to encourage research into developing more efficient maintenance and self-maintenance expressions than are possible using the insertion/deletion paradigm. For example, the change-table technique can be used to (1) efficiently propagate certain kinds of deletions, and (2) for propagating certain kinds of updates directly, without querying the sources. These extensions of our techniques are discussed in [Gup00].

References

[AFP00] M. Akhtar Ali, Alvaro A. A. Fernandes, and Norman W. Paton. Incremental maintenance of materialized oql views. In *Proceedings of the third ACM international workshop on Data warehousing and OLAP*, 2000.

[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.

[BDD+98] R. Bello, K. Dias, A. Downing, J. Feenan Jr., J. Finnerty, W. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in oracle. In *Proceedings of International Conference on Very Large Data Bases*, pages 659–664, 1998.

[BLT86] J. Blakeley, P. Larson, and F. Tompa. Efficiently Updating Materialized Views. In Carlo Zaniolo, editor, *Proceedings of ACM SIGMOD 1986 International Conference on Management of Data*, pages 61–71, Washington, D.C., May 28-30 1986.

[BM90] Jose A. Blakeley and Nancy L. Martin. Join index, materialized view, and hybrid hash join: A performance analysis. In *Proceedings of the Sixth IEEE International Conference on Data Engineering*, pages 256–263, Los Angeles, CA, February 5-9 1990.

[BPP+93] B.Mitschang, H. Pirahesh, P. Pistor, B. Lindsay, and N. Sudkamp. Sql/xnf - processing composite objects as abstractions over relational data. In *Proceedings of the Ninth IEEE International Conference on Data Engineering*, Vienna, Austria, April 1993.

[BW89] T. Barsalou and G. Wiederhold. Knowledge based mapping of relations into objects. In *Proceeding of Computer Aided Design*, 1989.

[BW90] B.S.Lee and G. Wiederhold. Outer joins and filters for instantiating objects from relationals databases through views. Technical report, CIFE - Stanford University, 1990.

[CGL+96] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of ACM SIGMOD 1996 International Conference on Management of Data*, pages 469–480, 1996.

[CKL+97] L. Colby, A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. Supporting multiple view maintenance policies. In *Proceedings of ACM SIGMOD 1997 International Conference on Management of Data*, pages 405–416, 1997.

[CW91] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In Guy M. Lohman, Amilcar Sernadas, and Rafael Camps, editors, *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 108–119, Barcelona, Spain, September 3-6 1991.

[ESWDR02] Maged EL-Sayed, Ling Wang, Luping Ding, and Elke A. Rundensteiner. An algebraic approach for incremental maintenance of materialized xquery views. In *Proceedings of the fourth international workshop on Web information and data management*, 2002.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Generalized projections: A powerful approach to aggregation. In *Proceedings of the 21st International Conference on Very Large Databases*, Zurich, Switzerland, September 11-15 1995.

[GJM94] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. Technical Memorandum 113880-941101-32, AT&T Bell Laboratories, November 1994.

[GJM96] A. Gupta, H. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Proceedings of the Fifth International Conference on Extending Database Technology*, pages 140–144, Avignon, France, March 1996. Industrial Session.

[GJM97] A. Gupta, H.V. Jagadish, and I.S. Mumick. Maintenance and self-maintenance of outerjoin views. In *Proceedings of the NGITS*, Tel Aviv, Israel, June 1997.

[GK98] T. Griffin and B. Kumar. Algebraic change propagation for semijoin and outerjoin queries. *SIGMOD Record*, 27(3), September 1998.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, San Jose, California, May 1995.

[GLT94] Timothy Griffin, Leonid Libkin, and Howard Trickey. A correction to "incremental recomputation of active relational expressions" by Qian and Wiederhold. Technical report, AT&T Bell Laboratories, Murray Hill NJ, 1994.

[GM95] A. Gupta and I. Mumick. Maintenance of materialized views: Problems, techniques, and applica tions. *Special Issue on Materialized Views and Data Warehousing,* IEEE Data Engineering Bulletin, 18(2):3–19, June 1995.

[GMS93] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data*, Washington, DC, May 26-28 1993.

[Gup00] H. Gupta. *Selection and Maintenance of Views in a Data Warehouses*. PhD thesis, Stanford University, Department of Computer Science, 2000. Chapter 4.

[Han87] E. Hanson. A performance analysis of view materialization strategies. In Umeshwar Dayal and Irv Traiger, editors, *Proceedings of ACM SIGMOD 1987 International Conference on Management of Data*, pages 440–453, San Francisco, CA, May 27-29 1987.

[KR02] Andreas Koeller and Elke A. Rundensteiner. Incremental maintenance of schema-restructuring views. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2002.

[LSPC00] W. Lehner, R. Sidle, H. Pirahesh, and R. Cochrane. Maintenance of automatic summary tables. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pages 512–513, 2000.

[LV01] Jixue Liu and Millist Vincent. Derivation of incremental equations for nested relations. In *Proceedings of the 12th Australasian conference on Database technologies*, 2001.

[LVM99] Jixue Liu, Millist Vincent, and Mukesh Mohania. Incremental maintenance of nested relational views. In *Proceedings of the International Symposium on Database Engineering and Applications (IDEAS)*, 1999.

[MQM97] I. Mumick, D. Quass, and B. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proceedings of the ACM SIGMOD International Conference of Mangement of Data*, Tucson, Arizona, June 1997.

[PSCP02] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *Proceedings of the International Conference on Very Large Data Bases*, 2002.

[QGMW96] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proceedings of the Fifth International Conference on Parallel and Distributed Information Systems (PDIS)*, 1996.

[QM97] D. Quass and I. Mumick. Optimizing the refresh of materialized views. Unpublished Manuscript, 1997.

[Qua97] D. Quass. *Materialized Views in Data Warehouses.* PhD thesis, Stanford University, Department of Computer Science, 1997. Chapter 4. Preliminary version appears as *Maintenance Expressions for Views with Aggregation* in the *ACM Workshop on Materialized Views*, 1996.

[QW91] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, pages 337–341, 1991.

[RK86] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS±. *IEEE Computer*, pages 19–25, December 1986.

[RU96] A. Rajaraman and J. Ullman. Integrating information by outerjoins and full disjunctions. In *Proceedings of the Fifteenth Symposium on Principles of Database Systems (PODS)*, Montreal, Canada, June 1996.

Recommended by Yannis Ioannidis, Area Editor.

A Maintenance expressions derived using techniques in [Qua97]

For our purposes, its not important to understand the maintenance expressions given below and they are given here primarily to show their complexity. We have used $\bowtie$ to denote a natural join operation, $\bowtie_G$ to denote an equi-join operation on a set of attributes G, $\overline{\ltimes}$ to denote an anti-semijoin operation, and π_A to denote the generalized projection symbol, which represent the groupby operation of SQL as described in [GHQ95] and Section 1. Also, $l.a$ and $r.a$ refer to the attribute a of the left and right operands respectively.

$$
\begin{aligned}
\text{Let } A_1 &= \{\text{storeID}, \text{itemID}, \text{SumSISales}, \text{NumSISales}\} \\
\text{Let } A_{1,ins} &= \{\text{storeID}, \text{itemID}, \text{price}, \_\text{count} = 1\} \\
\text{Let } A_{1,del} &= \{\text{storeID}, \text{itemID}, \text{price} = -\text{price}, \_\text{count} = -1\} \\
\text{Let } A_{1,\delta} &= \{\text{storeID}, \text{itemID}, \text{SumSISales} = \text{sum}(\text{price}), \text{NumSISales} = \text{sum}(\_\text{count})\} \\
\delta\texttt{sales} &= \Pi_{A_{1,ins}}(\sigma_{(date>1/1/95)}(\triangle\texttt{sales})) \uplus \Pi_{A_{1,del}}(\sigma_{(date>1/1/95)}(\triangledown\texttt{sales})) \\
\triangledown(\texttt{SISales}) &= \Pi_{r.a|a\in A_1}(\pi_{A_{1,\delta}}(\delta\texttt{sales}) \bowtie_{storeID,itemID} \texttt{SISales}) \\
\triangle(\texttt{SISales}) &= \Pi_{\{(r.a+l.a)|a\in A_1\}}(\sigma_{r.NumSISales+l.NumSISales>0}(\pi_{A_{1,\delta}}(\delta\texttt{sales}) \bowtie_{storeID,itemID} \texttt{SISales})) \\
&\quad \uplus \sigma_{l.NumSISales>0}(\pi_{A_{1,\delta}}(\delta\texttt{sales}) \overline{\ltimes}_{storeID,itemID} \texttt{SISales}) \\
\text{Let } A_2 &= \{\text{city}, \text{SumCiSales}, \text{NumCiSales}\} \\
\text{Let } A_{2,ins} &= \{\text{city}, \text{SumSISales}, \text{NumSISales}\} \\
\text{Let } A_{2,del} &= \{\text{city}, \text{SumSISales} = -\text{SumSISales}, \text{NumSISales} = -\text{NumSISales}\} \\
\text{Let } A_{2,\delta} &= \{\text{city}, \text{SumCiSales} = \text{sum}(\text{SumSISales}), \text{NumCiSales} = \text{sum}(\text{NumSISales})\} \\
\delta_2 &= \Pi_{A_{2,ins}}(\triangle\texttt{SISales} \bowtie \texttt{stores}) \uplus \Pi_{A_{2,del}}(\triangledown\texttt{SISales} \bowtie \texttt{stores}) \\
\triangledown(\texttt{CitySales}) &= \Pi_{r.a|a\in A_2}(\pi_{A_{2,\delta}}(\delta_2) \bowtie_{storeID,itemID} \texttt{CitySales}) \\
\triangle(\texttt{CitySales}) &= \Pi_{\{(r.a+l.a)|a\in A_2\}}(\sigma_{r.NumCiSales+l.NumCiSales>0}(\pi_{A_{2,\delta}}(\delta_2) \bowtie_{city} \texttt{CitySales})) \\
&\quad \uplus \sigma_{l.NumCiSales>0}(\pi_{A_{2,\delta}}(\delta_2) \overline{\ltimes}_{city} \texttt{CitySales})
\end{aligned}
$$

$$
\begin{aligned}
\text{Let } A_3 &= \{\text{category}, \text{SumCaSales}, \text{NumCaSales}\} \\
\text{Let } A_{3,ins} &= \{\text{category}, \text{SumSISales}, \text{NumSISales}\} \\
\text{Let } A_{3,del} &= \{\text{category}, \text{SumSISales} = -\text{SumSISales}, \text{NumSISales} = -\text{NumSISales}\} \\
\text{Let } A_{3,\delta} &= \{\text{category}, \text{SumCaSales} = \text{sum}(\text{SumSISales}), \text{NumCaSales} = \text{sum}(\text{NumSISales})\} \\
\delta_3 &= \Pi_{A_{3,ins}}(\triangle\texttt{SISales} \bowtie \texttt{items}) \uplus \Pi_{A_{3,del}}(\triangledown\texttt{SISales} \bowtie \texttt{items}) \\
\triangledown(\texttt{CategorySales}) &= \Pi_{3.a|a\in A_3}(\pi_{A_{3,\delta}}(\delta_3) \bowtie_{storeID,itemID} \texttt{CategorySales}) \\
\triangle(\texttt{CategorySales}) &= \Pi_{\{(r.a+l.a)|a\in A_3\}}(\sigma_{r.NumCaSales+l.NumCaSales>0}(\pi_{A_{3,\delta}}(\delta_3) \bowtie_{category} \texttt{CategorySales})) \\
&\quad \uplus \sigma_{l.NumCaSales>0}(\pi_{A_{3,\delta}}(\delta_3) \overline{\ltimes}_{category} \texttt{CategorySales})
\end{aligned}
$$

Computing Tuple Accesses. As `SISales` is not materialized, computing tuples of $\triangle$`SISales` requires that for each tuple in $\pi_{A_{1,\delta}}(\delta$`sales`$)$, we must look up all tuples of `sales` that have the same *storeID* and *itemID* values. Given the database sizes of Table 1, assume that each tuple of `SISales` is derived from 1,000 tuples of `sales` on an average. Thus, computing 600 tuples of $\triangle$`SISales` requires 600,000 tuple accesses. We need 11,000 accesses to read the base relations `stores` and `items` into main-memory. Even assuming that rest of the computation can be done in main memory, the total number of tuples accesses to refresh the views `CitySales` and `CategorySales` is $10,000 + 600,000 + 11,000 + 2,020$, where 2,020 tuple accesses are due to the final tuple updates in `CitySales` and `CategorySales`. Note that each update requires a read and a write access.