Maintenance Expressions for Views with Aggregation

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

Abstract

Materialized views, especially views involving aggregation, are often used in environments such as data warehouses to speed up the evaluation of complex queries. Because the views can be quite complex, as changes are made to the underlying base data it is usually better to *incrementally maintain* a view by propagating changes to base data onto the view than to recompute the view from scratch. Griffin and Libkin [GL95] provide a framework for deriving incremental view maintenance expressions for a view with duplicate semantics, where the view can be defined using any number of select, project, join, bag union, and monus bag-algebra operators. However, aggregation was considered only in a limited sense; for example, aggregation with group-by attributes was not allowed. We extend the framework of [GL95] to include maintaining views with aggregation in the general case, where aggregation can include group-by attributes and a view can include several aggregate operators.

1 Introduction

Materialized views, views that are computed and stored in a database, are necessary to the success of data warehouses where they are used to speed up query processing on large amounts of data. Materialized views become out of date when changes are made to the base data upon which the view is derived. To bring a view back up to date with the base data 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 [GL95].

Materialized views involving aggregation are especially important in data warehouses because clients of the warehouse often want to summarize data in order to analyze trends [GBLP95, HRU96]. Being able to incrementally maintain aggregate views is critical to large warehouses because many such views may be materialized, and recomputing them from scratch requires reading the often very large base relations.

Existing formulas for incremental view maintenance [GMS93, GL95] are not able to maintain views with aggregation in the general case. In particular, Griffin and Libkin [GL95] give an algebraic approach to incremental view maintenance that is based upon *bag algebra*, which allows maintaining views that have the SQL bag semantics. Changes to base data are propagated onto a view by propagating the changes up through each of the operators in the view definition. Using their approach it is possible to derive maintenance expressions for views that include any number of select, project, join, bag union, and monus (bag difference) operators. However, the view is restricted to include at most one aggregate operator as the final (top-most) operator in the view definition, and group-by attributes are not allowed. Reasoning about aggregation is difficult because aggregate operators behave very differently than other relational- and bag-algebra operators. For example, while rules for optimizing queries with other operators have long been known, only recently have rules for optimizing queries with aggregate operators been given [CS94, GHQ95, YL95].

In this paper we extend the work of [GL95] in an important way by giving incremental view maintenance expressions for the general case of views with aggregation. The view can include additional operators following the aggregate operator, including other aggregate operators, and the aggregate operator can include group-by attributes. All of the SQL aggregate functions (`count`, `sum`, `avg`, `min`, and `max`) are considered. For example, using our extensions it is possible to incrementally maintain a materialized view based on the following query, which asks for the maximum daily sales total for each store.

```
SELECT store_id, max(daily_total)
FROM   ( SELECT store_id, date,
                sum(sale_price) AS daily_total
         FROM   sales_log
         GROUP BY store_id, date )
GROUP BY store_id
```

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

This paper presents the following results:

- We give simple maintenance expressions that are sufficient for propagating insertions up through an aggregate operator in a view definition and for propagating deletions when the operator does not include `min` or `max` aggregate functions.

- Maintaining the values of `min` and `max` functions in the presence of deletions may require recomputing the function value from the state of the base data after the changes have been applied. We therefore give more complex maintenance expressions that handle propagating insertions and deletions up through an aggregate operator when the operator includes `min` and `max` functions.

- Propagating insertions and deletions up through an aggregate operator often results in updates to the aggregate function values. Updates are usually expressed in view maintenance expressions as delete-insert pairs, but it may be more efficient to apply them to a view if they are expressed directly as updates. We present maintenance expressions for aggregate operators that return results as updates when possible.

Paper Outline Related work is given in Section 2. Section 3 briefly explains the framework given in [GL95] for deriving view maintenance expressions and introduces the notation we will use for aggregation. Section 4 presents simple maintenance expressions for propagating insertions and for propagating deletions when the aggregate operator does not include `min` or `max` functions. Section 5 presents more complex maintenance expressions that handle propagating insertions and deletions when the aggregate operator does include `min` and `max` functions. Section 6 gives maintenance expressions for aggregate operators that return results as updates when possible. Conclusions and areas of future work are given in Section 7.

2 Related Work

As explained previously, Griffin and Libkin [GL95] present an algorithm for deriving incremental view maintenance expressions for views with duplicates. Their algorithm is based on functions that propagate deletions and insertions to base relations onto a materialized view by propagating them up through the operators in the view definition. They give functions for propagating deletions and insertions up through select, duplicate-eliminating project, duplicate-preserving project, cartesian-product, bag union, and monus operators.

Aggregation is considered in [GL95] but group-by attributes are not allowed. Further, the effect of deletions and insertions on the result of an aggregate function is handled outside the rest of their framework by updating an aggregate function result. Because aggregation is handled outside their framework, the aggregate operator is restricted to be the top-most operator in the view.

Gupta et al. [GMS93] also present algorithms for incrementally maintaining views with duplicates. Their algorithms are applied in the context of datalog programs that produce multisets (bags). The multisets are represented by annotating each tuple with a count indicating the number of distinct derivations (*i.e.*, the multiplicity) of the tuple in the multiset. Tuples to be inserted or deleted are also annotated with counts indicating the number of times the tuple is to be inserted or deleted, with deletions represented by negative counts. Changes are propagated by following an algorithm that correctly propagates tuple counts.

Aggregation is considered in [GMS93] and group-by attributes are allowed. But their formula does not include the cases where propagating insertions and deletions causes new tuples to be inserted into the result of the aggregate operator or tuples to be deleted from the result of the aggregate operator. Also, they do not consider the problems associated with maintaining `min` and `max` aggregate functions in the presence of deletions. In a sense this paper expresses the formula for aggregation of [GMS93] within the algebraic framework of [GL95]. We extend their formula to handle the case where tuples are inserted into or deleted from the result of the aggregate operator, the case of maintaining `min` and `max` aggregate functions in the presence of deletions, and treating updates to the result of aggregation as updates rather than delete-insert pairs.

Other work [GLT, QW91] has dealt with deriving maintenance expressions for views without duplicates (set semantics). Maintaining views that involve aggregation is not considered.

3 Preliminaries

In Section 3.1 we briefly explain the framework given in [GL95] that we will extend for aggregation. We then present in Section 3.2 the notation we will use to represent aggregate operators.

First we need some general notation. We use $\uplus$ to denote bag union, $\dot{-}$ to denote monus, $\triangledown R$ to denote deletions from a bag-algebra expression R, $\triangle R$ to denote insertions into R, σ_p, to denote selection on condition p, Π_A to denote duplicate-preserving projection on a set of attributes A, $\times$ to denote cross-product, $\bowtie_A$ to denote equijoin on a set of attributes A, $\ltimes$ to denote semijoin (*i.e.*, $R \ltimes S$ returns the tuples in R that are joinable with tuples in S), and $\overline{\ltimes}$ to denote antisemijoin (*i.e.*, $R \overline{\ltimes} S$ returns the tuples in R that are not joinable with any tuple in S).

3.1 Framework for deriving maintenance expressions

Let T be a bag-algebra expression and let s be a database state where s is a function that maps all relations mentioned in T to multisets. Then $s(T)$ represents the result of evaluating T in database state s. Given two bag-algebra expressions S and T, we say $S =_b T$ iff for every database state s such that s is defined on all of the relations mentioned in S and T, $s(S) = s(T)$.

Let a database contain a set of relations $\mathcal{R}$. A transaction t is defined to contain for each relation R_i in $\mathcal{R}$ the expression

$$R_i \leftarrow (\ R_i \dot{-} \bigtriangledown R_i) \uplus \triangle R_i$$

defining the (possibly empty) set of tuples to be deleted from R_i (denoted by $\bigtriangledown R_i$) and the (possibly empty) set of tuples to be inserted into R_i (denoted by $\triangle R_i$).

Let V be a bag-algebra expression defined on a subset of the relations in $\mathcal{R}$. An expression $pre(t, V)$ is defined as a *pre-expression* of V with respect to a transaction t if for every database state s, $s(pre(t, V)) = t(s)(V)$. In other words, $pre(t, V)$ can be evaluated before transaction t has been applied to s in order to determine the result of V after t has been applied.

The goal in deriving view maintenance expressions for a view V is to derive two functions $\bigtriangledown(t, V)$ and $\triangle(t, V)$ such that for any transaction t, $pre(t, V) =_b (V \dot{-} \bigtriangledown(t, V)) \uplus \triangle(t, V)$. $\bigtriangledown(t, V)$ is an expression returning the tuples that must be deleted from V due to t, and $\triangle(t, V)$ is an expression returning the tuples that must be inserted into V due to t. Of course, many such functions $\bigtriangledown(t, V)$ and $\triangle(t, V)$ are possible, but not all are equally desirable. For example, we could let $\bigtriangledown(t, V)$ equal V and $\triangle(t, V)$ equal $pre(t, V)$, but this is equivalent to recomputing the view from scratch. To guard against such wasteful definitions, [GL95] introduces the concept of "minimality" to ensure that no unnecessary tuples are produced. Functions are defined to be *weakly minimal* if $\bigtriangledown(t, V) \dot{-} V =_b \phi$, meaning that only tuples in V are deleted. Furthermore, they are defined to be *strongly minimal* if in addition the following condition also holds: $\bigtriangledown(t, V) \min \triangle(t, V) =_b \phi$, meaning that tuples are not deleted and then reinserted. (The operator "min" takes the minimum intersection of two multisets by taking the minimum count of each distinct tuple in common between the two multisets.)

In order to derive $\bigtriangledown(t, V)$ and $\triangle(t, V)$, [GL95] begins by giving *change propagation equations* that show how deletions and insertions are propagated up through each of the following operators: σ, $\times$, $\uplus$, $\dot{-}$, Π, and duplicate-eliminating projection. For example, the following change propagation equations show how deletions and insertions are propagated up through σ and Π. We assume in the equations that $\bigtriangledown S$ and $\triangle S$ are at least weakly minimal.

$$\begin{aligned}
\sigma_p(S \dot{-} \bigtriangledown S) &=_b \sigma_p(S) \dot{-} \sigma_p(\bigtriangledown S) \\
\sigma_p(S \uplus \triangle S) &=_b \sigma_p(S) \uplus \sigma_p(\triangle S) \\
\Pi_A(S \dot{-} \bigtriangledown S) &=_b \Pi_A(S) \dot{-} \Pi_A(\bigtriangledown S) \\
\Pi_A(S \uplus \triangle S) &=_b \Pi_A(S) \uplus \Pi_A(\triangle S)
\end{aligned}$$

The change propagation equations, along with refinements to guarantee strong minimality, are used to derive functions $\bigtriangledown(t, O)$ and $\triangle(t, O)$ for each operator O that they consider. For example, the following functions from [GL95] propagate deletions and insertions to a bag-algebra expression S up through operators $\sigma_p(S)$ and $\Pi_A(S)$, and guarantee strong minimality. The first function, $\bigtriangledown(t, \sigma_p(S))$, returns the tuples to be deleted from $\sigma_p(S)$ due to a transaction t as the result of applying σ_p to the tuples to be deleted from expression S.

$$\begin{aligned}
\bigtriangledown(t, \sigma_p(S)) &\equiv \sigma_p(\bigtriangledown(t, S)) \\
\triangle(t, \sigma_p(S)) &\equiv \sigma_p(\triangle(t, S)) \\
\bigtriangledown(t, \Pi_A(S)) &\equiv \Pi_A(\bigtriangledown(t, S)) \dot{-} \Pi_A(\triangle(t, S)) \\
\triangle(t, \Pi_A(S)) &\equiv \Pi_A(\triangle(t, S)) \dot{-} \Pi_A(\bigtriangledown(t, S))
\end{aligned}$$

The functions $\bigtriangledown(t, V)$ and $\triangle(t, V)$ that derive the tuples to be deleted from and inserted into the result of a view V are mutually-recursively defined in terms of the functions that propagate deletions and insertions up through each of the operators in the definition of V. In this paper we extend the approach of [GL95] by defining change propagation equations and corresponding change propagation functions for aggregate operators.

3.2 Generalized projection

We will use the *generalized projection* operator, denoted as π_A, from [GHQ95] to represent aggregation. The generalized projection operator is an extension of duplicate-eliminating projection, where the projected attributes A can include aggregate functions as well as regular attributes. The regular attributes become group-by attributes for the aggregate functions. We use $GB(A)$ to denote the set of group-by attributes in A. Note that a generalized projection operator without any aggregate functions is identical to duplicate-eliminating projection ("select distinct").

For example, the expression

$$\pi_{store\_id, date, daily\_total = sum(sale\_price)} sales\_log$$

is equivalent to the SQL query

```
SELECT store_id, date, sum(sale_price) AS daily_total
FROM   sales_log
GROUP BY store_id, date
```

4 Simple Maintenance Expressions For Aggregate Operators

In this section we give simple change propagation equations that are sufficient for propagating insertions up through an aggregate operator and for propagating deletions when the operator does not include `min` or `max` aggregate functions. We use these change propagation equations to derive functions that return the tuples to be deleted from and inserted into the result of an aggregate operator due to a transaction t. Finally, we explore how these functions might be efficiently implemented.

4.1 Simple change propagation equations

Change propagation equations for aggregate operators are given in Figure 1, which is explained below. The equations show how deletions and insertions to an expression R are propagated up through an aggregate operator π_A. over R. Before explaining the equations we need to explain some notation.

In the (duplicate-preserving) projections in the equations we must distinguish between the attributes coming from the first aggregate operator (left-hand side of the join) and the attributes coming from the second aggregate operator (right-hand side of the join). We use $1.a$ to indicate an attribute coming from the first aggregate operator and $2.a$ to indicate an attribute coming from the second aggregate operator. Furthermore, we assume in the equation propagating deletions that a `count` aggregate function is also computed by the $\pi_A(\bigtriangledown R)$ and $\pi_A(R)$ operators if `count` does not already appear in A.

The functions f and g used in the equations are defined in Table 1. Function f computes the new value of an aggregate function based upon its old value in $\pi_A(R)$ and the result of applying the aggregate function to the inserted tuples ($\pi_A(\triangle R)$). Function g similarly computes the new value of an aggregate function based upon its old value and the result of applying the aggregate function to the deleted tuples. Functions f and g can be defined for any aggregate function that is *incrementally computable* with respect to the type of change being propagated. Because `min` and `max` are not always incrementally computable with respect to deletions, g is not defined for `min` and `max`, and the equations in this section are insufficient to propagate deletions up through aggregate operators having `min` or `max` functions. Handling deletions for `min` and `max` functions is covered in Section 5. Note that since `avg` is computed in terms of `sum` and `count`, if A includes an `avg` function then we assume it also includes the corresponding functions for `sum` and `count`. We do not further consider `avg` separately in this paper.

We are now ready to explain the equations of Figure 1. The equations assume that the changes to R are expressed in a form that is at least weakly minimal.

The first equation propagates insertions by giving the result of $\pi_A(R \uplus \triangle R)$ in terms of $\pi_A(R)$ and $\pi_A(\triangle R)$. Intuitively, on the right-hand side of the equation the result of $\pi_A(\triangle R)$ is joined with the result of $\pi_A(R)$, and joining tuples in $\pi_A(R)$ having old aggregate function values are replaced by tuples with new aggregate function values. Note that since $GB(A)$ forms a key of both $\pi_A(\triangle R)$ and $\pi_A(R)$, each tuple in $\pi_A(\triangle R)$ joins with at most one tuple in $\pi_A(R)$ and vice-versa. The last component of the equation (involving the $\overline{\triangleright\!\!\!<}$) adds the tuples in $\pi_A(\triangle R)$ that do not join with a tuple in $\pi_A(R)$.

The second equation propagates deletions by giving the result of $\pi_A(R \dot{-} \bigtriangledown R)$ in terms of $\pi_A(R)$ and $\pi_A(\bigtriangledown R)$. The right-hand side of the equation is similar to the equation for insertions, but before adding a tuple with the new values a check is made to ensure that not all tuples of the group were deleted. Additionally, the last component of the equation for insertions can be omitted from the equation for deletions.

4.2 Simple change propagation functions

Functions that return the tuples to be deleted from and inserted into the result of an aggregate operator due to a transaction t are given in Figure 2, which we explain below. The figure shows two functions, $\bigtriangledown(t, \pi_A(R))$ and $\triangle(t, \pi_A(R))$, which determine the tuples to be deleted from and inserted into the result of an aggregate operator π_A over an arbitrary bag-algebra expression R. The functions can be used to propagate deletions and insertions up through an aggregate operator that appears anywhere in a view definition. They are based upon the simple change propagation equations given in Section 4.1. The first function removes tuples in $\pi_A(R)$ that are affected by the changes to R. The second function inserts corresponding tuples with the updated aggregate function values, as well as tuples created by the changes to R that have group-by attribute values not appearing in $\pi_A(R)$. We assume that if deletions are being propagated then a `count` aggregate function is also computed by $\pi_A(R)$ and $\pi_{A_\delta}(\delta R)$ if `count` does not already appear in A. If deletions are not being propagated then the selection on count can be removed from the second equation.

We now explain the notation used in Figure 2. In the figure δR is used as a shorthand for the following formula:

$$\Pi_{A_{ins}}(\triangle(t, R)) \uplus \Pi_{A_{del}}(\bigtriangledown(t, R))$$

$$\begin{aligned}
\pi_A(R \uplus \triangle R) \;&=_b\; (\ \pi_A(R) \dot{-} \Pi_{\{2.a|a\in A\}}(\ \pi_A(\triangle R) \bowtie_{GB(A)} \pi_A(R)))\\
&\quad \uplus \Pi_{\{f(2.a,1.a)|a\in A\}}(\ \pi_A(\triangle R) \bowtie_{GB(A)} \pi_A(R))\\
&\quad \uplus \pi_A(\triangle R)\ \overline{\ltimes}_{GB(A)}\ \pi_A(R))\\
\pi_A(R \dot{-} \bigtriangledown R) \;&=_b\; (\ \pi_A(R) \dot{-} \Pi_{\{2.a|a\in A\}}(\ \pi_A(\bigtriangledown R) \bowtie_{GB(A)} \pi_A(R)))\\
&\quad \uplus \Pi_{\{g(2.a,1.a)|a\in A\}}(\ \sigma_{2.\text{count}(*)-1.\text{count}(*)>0}\\
&\qquad (\ \pi_A(\bigtriangledown R) \bowtie_{GB(A)} \pi_A(R)))
\end{aligned}$$

Figure 1: Simple change propagation equations for aggregate operators

Definition of Attribute a	Function $f(2.a, 1.a)$	Function $g(2.a, 1.a)$
group-by attribute	$2.a$	$2.a$
`count(*)`	$2.a + 1.a$	$2.a - 1.a$
`sum`(x)	$2.a + 1.a$	$2.a - 1.a$
`avg`(x)	$\frac{2.\texttt{sum}(x)+1.\texttt{sum}(x)}{2.\texttt{count(*)}+1.\texttt{count(*)}}$	$\frac{2.\texttt{sum}(x)-1.\texttt{sum}(x)}{2.\texttt{count(*)}-1.\texttt{count(*)}}$
`min`(x)	$\texttt{min}(2.a, 1.a)$	undefined
`max`(x)	$\texttt{max}(2.a, 1.a)$	undefined

Table 1: Definition of functions f and g in Figure 1

Intuitively, δR represents the net effect of both the insertions and deletions to expression R. The set of attributes A_{ins} includes the group-by attributes of A from π_A, as well as an attribute f_i for each aggregate function $a_i \in A$, where f_i evaluates to the target of the aggregation. For example, if A includes an aggregate function $a_i = \texttt{sum}(b+c)$ then an attribute $f_i = b + c$ is added to A_{ins}. Furthermore, if A includes an aggregate function $a_i = \texttt{count(*)}$ then an attribute $cnt = 1$, evaluating to 1 for every tuple, is added to A_{ins}. The set of attributes A_{del} is similar to A_{ins}, except that columns for `sum` evaluate to the negative of their values in A_{ins}. For example, if A includes an aggregate function $a_i = \texttt{sum}(b+c)$ then an attribute $f_i = -(b+c)$ is added to A_{del}. Furthermore, if A includes an aggregate function $a_i = \texttt{count(*)}$ then an attribute $cnt = -1$ is added to A_{del}.

We use A_δ to denote a set of attributes that includes the same attributes as A except that the aggregate function definitions are modified to refer to the attributes f_i in A_{ins} and A_{del}. For example, in the above case where A includes an aggregate function $a_i = \texttt{sum}(b+c)$ and A_{ins} and A_{del} include an attribute $f_i = b + c$ (or $-(b+c)$), then A_δ includes $a_i = \texttt{sum}(f_i)$. Furthermore, if A includes an aggregate function $a_i = \texttt{count(*)}$ then A_δ includes $a_i = \texttt{sum}(cnt)$.

The functions given in Figure 2 assume that the functions $\bigtriangledown(t, R)$ and $\triangle(t, R)$ returning the deletions and insertions to R guarantee at least weak minimality. The functions $\bigtriangledown(t, \pi_A(R))$ and $\triangle(t, \pi_A(R)$ guarantee weak minimality but not strong minimality, even if $\bigtriangledown(t, R)$ and $\triangle(t, R)$ guarantee strong minimality. In order to guarantee strong minimality we need to ensure that the tuples that are removed by $\bigtriangledown(t, \pi_A(R))$ are not replaced by identical tuples in $\triangle(t, \pi_A(R))$. Tuples in $\bigtriangledown(t, \pi_A(R))$ and $\triangle(t, \pi_A(R))$ that share the same values for group-by attributes are not identical if at least one of the aggregate function values is different. Therefore, to guarantee strong minimality we must perform the additional selections shown in Figure 3. In the figure we assume that the set of aggregate functions in A is $\{a_1, a_2, \ldots, a_k\}$.

EXAMPLE 4.1 The following example illustrates how the functions in Figure 2 work.

Suppose a `sales_log` table contains the following data. We purposely limit the number of tuples in the table for ease of exposition of the example.

sale_id	store_id	date	sale_price
0001	555	1 May 1996	10
0002	555	1 May 1996	20
0003	555	2 May 1996	40
0004	555	3 July 1996	100

Suppose that we want to materialize a `daily_sales` view computing the total daily sales for each store. We would materialize the following view. Since we are propagating deletions, we need to add a `count(*)` aggregate function to the view in order to incrementally maintain it. Computing the value of `count(*)` should not require much overhead when materializing the view.

$$\begin{aligned}\triangledown(t, \pi_A(R)) &\equiv \Pi_{\{2.a|a\in A\}}(\ \pi_{A_\delta}(\delta R) \bowtie_{GB(A)} \pi_A(R))\\ \triangle(t, \pi_A(R)) &\equiv \Pi_{\{f(2.a,1.a)|a\in A\}}(\ \sigma_{2.\mathrm{count}(*)+1.cnt>0}(\ \pi_{A_\delta}(\delta R) \bowtie_{GB(A)} \pi_A(R)))\\ &\quad \uplus\ \sigma_{1.cnt>0}\ (\ \pi_{A_\delta}(\delta R) \ltimes_{GB(A)} \pi_A(R))\end{aligned}$$

Figure 2: Functions guaranteeing weak minimality

```
CREATE VIEW daily_sales AS
SELECT store_id, date, sum(sale_price) AS daily_total
       count(*) AS total_count
FROM   sales_log
GROUP BY store_id, date
```

Given the above `sales_log` table, the materialized `daily_sales` view would contain the following tuples.

store_id	date	daily_total	total_count
555	1 May 1996	30	2
555	2 May 1996	40	1
555	3 July 1996	100	1

Now let the following tuples be deleted from `sales_log` due to a transaction t ($\triangledown(t, sales\_log)$):

sale_id	store_id	date	sale_price
0001	555	1 May 1996	10
0004	555	3 July 1996	100

Also, let the following tuples be inserted into `sales_log` due to transaction t ($\triangle(t, sales\_log)$):

sale_id	store_id	date	sale_price
0004	555	3 May 1996	100
0005	555	1 May 1996	30
0006	555	3 May 1996	50

We want to determine the effect of the deletions and insertion to `sales_log` on the materialized `daily_sales` view. We do this using the functions of Figure 2. In the following the functions are computed in several steps in order to show the intermediate results. Efficient computation of the functions, including taking advantage of common subexpressions, is discussed in Section 4.3.

First we evaluate the result of $\delta sales\_log$ using the definition of $\delta sales\_log$ given previously. The schema of A_{ins} and A_{del} is (`store_id`, `date`, f_1, cnt). For each insertion, attribute f_1 evaluates to the target of the `sum(sale_price)` aggregation, which is `sale_price`. For each deletion, f_1 evaluates to the negative of the target, which is $-$`sale_price`. Attribute cnt evaluates to 1 for insertions and to -1 for deletions.

store_id	date	f_1	cnt
555	1 May 1996	-10	-1
555	3 July 1996	-100	-1
555	3 May 1996	100	1
555	1 May 1996	30	1
555	3 May 1996	50	1

Second, we evaluate the result of $\pi_{store\_id, date, daily\_total=sum(f_1), total\_count=sum(cnt)}(\delta sales\_log)$ by performing an aggregation on the value of $\delta sales\_log$: The result appears below.

store_id	date	daily_total	total_count
555	1 May 1996	20	0
555	3 July 1996	-100	-1
555	3 May 1996	150	2

We are now ready to compute the tuples to be deleted from and inserted into the result of the `daily_sales` view due to transaction t. The tuples to be deleted from the `daily_sales` view due to transaction t, $\triangledown(t, \pi_{store\_id, date, daily\_total, total\_count}(daily\_sales))$, are:

store_id	date	daily_total	total_count
555	1 May 1996	30	2
555	3 July 1996	100	1

The tuples to be inserted into the `daily_sales` view, $\triangle(t, \pi_{store\_id, date, daily\_total, total\_count}(daily\_sales))$, are:

store_id	date	daily_total	total_count
555	1 May 1996	50	2
555	3 May 1996	150	2

If the `daily_sales` view were actually a subquery of a more complex view, rather than being materialized itself, the deletions and insertions computed to `daily_sales` could be propagated further up through the other operators in the view definition. □

4.3 Efficiency considerations

The functions of Figure 3 include the expression

$$\pi_{A_\delta}(\delta R) \bowtie_{GB(A)} \pi_A(R)$$

in two places as a common subexpression. Therefore, the expression need be evaluated only once and saved

$$
\begin{aligned}
\nabla(t, \pi_A(R)) \;\equiv\; & \Pi_{\{2.a \mid a \in A\}}\big(\, \sigma_{2.\mathrm{count}(*) + 1.cnt = 0 \text{ or } f(2.a_1, 1.a_1) <> 2.a_1 \text{ or } f(2.a_2, 1.a_2) <> 2.a_2 \text{ or } \ldots \text{ or } f(2.a_k, 1.a_k) <> 2.a_k} \\
& \quad (\pi_{A_\delta}(\delta R) \bowtie_{GB(A)} \pi_A(R))\big) \\
\triangle(t, \pi_A(R)) \;\equiv\; & \Pi_{\{f(2.a, 1.a) \mid a \in A\}}\big(\, \sigma_{2.\mathrm{count}(*)+1.cnt>0 \text{ and } (f(2.a_1, 1.a_1) <> 2.a_1 \text{ or } f(2.a_2, 1.a_2) <> 2.a_2 \text{ or } \ldots \text{ or } f(2.a_k, 1.a_k) <> R.a_k)} \\
& \quad (\pi_{A_\delta}(\delta R) \bowtie_{GB(A)} \pi_A(R))\big) \\
& \uplus\, \sigma_{1.cnt>0}\, (\, \pi_{A_\delta}(\delta R)\, \overline{\ltimes}_{GB(A)}\, \pi_A(R))
\end{aligned}
$$

Figure 3: Functions guaranteeing strong minimality

either on disk or in memory if the result is sufficiently small.

In evaluating $\pi_{A_\delta}(\delta R) \bowtie_{GB(A)} \pi_A(R)$ and also $\pi_{A_\delta}(\delta R)\, \overline{\ltimes}_{GB(A)}\, \pi_A(R)$ it is important to note that both expressions are "controlled" by $\pi_{A_\delta}(\delta R)$. That is, the only groups in $\pi_A(R)$ for which aggregate values need to be computed (or looked up if $\pi_A(R)$ is stored) are those groups that join with a tuple in $\pi_{A_\delta}(\delta R)$. In a sense, we can "push" the join with $\pi_{A_\delta}(\delta R)$ down past the aggregate operator $\pi_A(R)$ to reduce the number of tuples involved in the aggregation, which can reduce the overall cost of computation [GHQ95].

Furthermore, both $\pi_{A_\delta}(\delta R) \bowtie_{GB(A)} \pi_A(R)$ and $\pi_{A_\delta}(\delta R)\, \overline{\ltimes}_{GB(A)}\, \pi_A(R)$ can be evaluated simultaneously. For each tuple in the result of $\pi_A(\delta R)$, if it is joinable with a tuple in R (and hence a tuple in $\pi_A(R)$) then it is part of the result of the first expression. Otherwise, it is belongs to the result of the second expression.

5 Handling Deletions for Min and Max

In this section we extend the functions $\nabla(t, \pi_A(R))$ and $\triangle(t, \pi_A(R))$ of Figure 2 to handle `min` and `max` aggregate functions in the presence of deletions. Although the functions we present guarantee only weak minimality for simplicity, they could be made to guarantee strong minimality easily by adding extra selection conditions as in Figure 3.

Functions `min` and `max` are not always incrementally computable with respect to deletions. The problem arises in the case of `min` when

1. a tuple t having the minimum value m is deleted,
2. not all tuples in the same group as t (*i.e.*, having the same values for group-by attributes) are deleted, and
3. a tuple t' in the same group as t with a value $\leq m$ is not inserted.

If the above conditions hold then the new minimum value for the group cannot be determined from the old minimum value and the changes. The new minimum value for that group must be computed by evaluating the expression R in the new state (after the changes have been applied). A similar situation holds for `max` when a tuple having the maximum value is deleted.

The functions in Figure 4 handle maintaining `min` and `max` aggregate functions in the presence of deletions. They are explained below. It should be noted that the only groups of R that need to be evaluated in the new state are those groups for which the above conditions hold.

The functions use an extended definition of the δR shorthand, called $\delta' R$, that contains additional attributes to separate the values being deleted from those being inserted for `min` and `max` aggregate functions. $\delta' R$ is defined similarly to δR:

$$\Pi_{A'_{ins}}(\triangle(t, R)) \uplus \Pi_{A'_{del}}(\nabla(t, R))$$

Attributes in A'_{ins} and A'_{del} are defined similarly to attributes in A_{ins} and A_{del} for δR, except for attributes that correspond to `min` and `max` aggregate functions in A. For each attribute $a_i \in A$ that is a `min` or `max` aggregate function, A'_{ins} and A'_{del} each contain an attribute f_i that evaluates to the target of the aggregation for A'_{ins}, and to *null* for A'_{del}. In addition, A'_{ins} and A'_{del} each contain another attribute g_i that evaluates to *null* for A'_{ins} and to the target of the aggregation for A'_{del}.

The definition of the set of attributes A_δ must also be extended. The extended definition, A'_δ, contains the same set of attributes as A, except that for each $a_i \in A$ that is an aggregate function, the definition of a_i in A_δ is modified to refer to f_i (as in Section 4.2). Also, for each $a_i \in A$ that is a `min` or `max` aggregate function, a new attribute b_i is added to A'_δ that computes the minimum (if a_i is a `min` aggregate function) or maximum (if a_i is a `max` aggregate function) of the values in g_i. Thus, a_i computes the value of the aggregate function for tuples in $\triangle(t, R)$, and b_i computes the value of the aggregate function for tuples in $\nabla(t, R)$. With regard to nulls in aggregation we assume the SQL-92 [MS93] semantics—that nulls are ignored when computing aggregate values.

We are now ready to define the C_i conditions used in the functions. Each condition $\{C_1, C_2, \ldots, C_k\}$ tests whether a `min` or `max` aggregate function $\{a_1, a_2, \ldots, a_k\}$ needs to be computed from the new state of R.

$$
\begin{aligned}
\nabla(t, \pi_A(R)) &\equiv \Pi_{\{2.a|a \in A\}}(\ \pi_{A'_\delta}(\delta' R) \bowtie_{GB(A)} \pi_A(R)) \\
\triangle(t, \pi_A(R)) &\equiv \Pi_{\{f(2.a,1.a)|a \in A\}}(\ \sigma_{2.\text{count}(*)+1.cnt>0 \text{ and not } C_1 \text{ and not } C_2 \ldots \text{ and not } C_k} \\
&\qquad (\pi_{A'_\delta}(\delta' R) \bowtie_{GB(A)} \pi_A(R))) \\
&\quad \uplus \Pi_{\{1.a|a \in A\}}(\ \sigma_{1.cnt>0}\ (\ \pi_{A'_\delta}(\delta' R) \overline{\ltimes}_{GB(A)} \pi_A(R))) \\
&\quad \uplus \Pi_{\{3.a|a \in A\}}(\ \sigma_{2.\text{count}(*)+1.cnt>0 \text{ and } (C_1 \text{ or } C_2 \ldots \text{ or } C_k)} \\
&\qquad (\pi_{A'_\delta}(\delta' R) \bowtie_{GB(A)} \pi_A(R)) \bowtie_{GB(A)} \pi_A((R \dot{-} \nabla(t, R)) \uplus \triangle(t, R)))
\end{aligned}
$$

Figure 4: Functions handling deletions to `min` and `max`

$$
\begin{aligned}
\nabla(t, \pi_A(R)) &\equiv \Pi_{\{2.a|a \in A\}}(\ \sigma_{2.\text{count}(*)+1.cnt=0}(\ \pi_{A_\delta}(\delta R) \bowtie_{GB(A)} \pi_A(R)) \\
\mu(t, \pi_A(R)) &\equiv \Pi_{\{a_{old}=2.a, a_{new}=f(2.a,1.a)|a \in A\}}(\ \sigma_{2.\text{count}(*)+1.cnt>0}(\ \pi_{A_\delta}(\delta R) \bowtie_{GB(A)} \pi_A(R))) \\
\triangle(t, \pi_A(R)) &\equiv \Pi_{\{1.a|a \in A\}}(\ \sigma_{1.cnt>0}(\ \pi_{A_\delta}(\delta R) \overline{\ltimes}_{GB(A)} \pi_A(R)))
\end{aligned}
$$

Figure 5: Functions expressing changes as deletions, updates, and insertions

Condition C_i for a `min` aggregate function a_i is defined as follows:

$$1.b_i = 2.a_i \text{ and } 1.a_i > 2.a_i$$

In other words, C_i is true when a tuple having the minimum value for a_i is deleted and a tuple having a value $\leq$ the minimum value is not inserted. Condition C_i for a `max` aggregate function is defined similarly, using $<$ for the second comparison. If C_i is true for some `min` or `max` aggregate function a_i and not all tuples in the group have been deleted ($2.$`count(*)` $+ 1.cnt > 0$), then a_i must be evaluated for the tuples in that group in the new state of R (which is $(R \dot{-} \nabla(t, R)) \uplus \triangle(t, R)$).

6 Propagating Changes as Updates

Propagating changes up through an aggregate operator usually results in updates to the tuples in the result of the aggregation. Updates in maintenance expressions are usually expressed as delete-insert pairs, but may be applied to a view more efficiently directly as updates. If a view includes a key, then updates can be applied to the view using an SQL `update` statement. In this section we give functions for propagating changes up through an aggregate operator that return updates, rather than delete-insert pairs, when possible.

The functions given in Figure 5 express the changes to the result of an aggregate operator $\pi_A(R)$ due to a transaction t as a set of tuples to be deleted, a set of tuples to be updated, and a set of tuples to be inserted. For the sake of simplicity, the functions are based on the simple functions of Figure 2 rather than those of Figures 3 or 4. They are defined as follows.

The expression $\nabla(t, \pi_A(R))$ returns the tuples to deleted from the result of $\pi_A(R)$ due to the changes in transaction t. Intuitively, a tuple t is deleted from $\pi_A(R)$ when the last tuple contributing to the group to which t belongs is deleted from R; *i.e.*, when the combined count of $\pi_{A_\delta}(\delta R)$ and $\pi_A(R)$ is zero.

The expression $\mu(t, \pi_A(R))$ returns the tuples to be updated in the result of $\pi_A(R)$ due to the changes in transaction t. Intuitively, a tuple t is updated in $\pi_A(R)$ when a tuple t' corresponding to the same group as t is found in $\pi_{A_\delta}(\delta R)$ and the combined count is not zero. For each attribute $a \in A$, the schema of $\mu(t, \pi_A(R))$ includes an attribute a_{old} containing the old value of a, and an attribute a_{new} containing the new value of a. In the functions a_{old} is set to $2.a$ and a_{new} is set to $f(2.a, 1.a)$.

The expression $\triangle(t, \pi_A(R))$ returns the tuples to be inserted into the result of $\pi_A(R)$ due to the changes in transaction t. Intuitively, tuples of $\pi_{A_\delta}(\delta R)$ are inserted when there is no tuple for the same group in $\pi_A(R)$.

In the equations of Figure 5 updates to expression R must be represented as delete-insert pairs, as in all previous equations. Therefore, the use of updates directly in view maintenance expressions is limited to generating updates for result of an aggregate operator. An interesting area of future study is to expand the use of updates in view maintenance expressions by deriving maintenance expressions that propagate updates directly through each of the bag-algebra operators.

7 Conclusions and Future Work

We first gave simple maintenance expressions for maintaining views that include aggregation by propagating insertions and deletions up through aggregate operators. We then enhanced the maintenance expressions to guarantee strong minimality, to handle the case of `min` and `max` aggregate functions in the presence of deletions, and to express changes in terms of updates rather than delete-insert pairs when possible.

In the future we plan to explore efficient algorithms for maintaining aggregate views. In addition, propagating updates directly rather than as delete-insert pairs may yield more efficient view maintenance algorithms, and we plan to explore this approach for other bag-algebra operators.

Acknowledgments The author would like to thank Inderpal Mumick for stimulating initial discussions and also Jennifer Widom and the reviewers for helpful comments on the paper.

References

[CS94] Surajit Chaudhuri and Kyuseok Shim. Including groupby in query optimization. In Jorge Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the 20^th International Conference on Very Large Databases*, pages 354–366, Santiago, Chile, September 12-15 1994.

[GBLP95] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and subtotals. Technical report no. msr-tr-95-22, Microsoft, 1995.

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

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In M. Carey and D. Schneider, editors, *Proceedings of ACM SIGMOD 1995 International Conference on Management of Data*, pages 328–339, San Jose, CA, May 23-25 1995.

[GLT] Timothy Griffin, Leonid Libkin, and Howard Trickey. An improved algorithm for incremental recomputation of active relational expressions. to appear in IEEE Transactions on Knowledge and Data Engineering.

[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.

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. In *Proceedings of ACM SIGMOD 1996 International Conference on Management of Data*, 1996.

[MS93] J. Melton and A. Simon. *Understanding the New SQL: A Complete Guide.* Morgan Kaufmann, 1993.

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

[YL95] W. Yan and P. Larson. Eager aggregation and lazy aggregation. In Umeshwar Dayal, Peter M.D. Gray, and Shojiro Nishio, editors, *Proceedings of the 21^st International Conference on Very Large Databases*, pages 345–357, Zurich, Switzerland, September 11-15 1995.