Efficient Maintenance of Materialized Outer-Join Views

Per- ˚Ake Larson
Microsoft Research
palarson@microsoft.com

Jingren Zhou
Microsoft Research
jrzhou@microsoft.com

Abstract

Queries containing outer joins are common in data warehousing applications. Materialized outer-join views could greatly speed up many such queries but most database systems do not allow outer joins in materialized views. In part, this is because outer-join views could not previously be maintained efficiently when base tables are updated. In this paper we show how to efficiently maintain general outer-join views, that is, views composed of selection, projection, inner and outer joins. Foreign-key constraints are exploited to reduce maintenance overhead. Experimental results show that maintaining an outer-join view need not be more expensive than maintaining an inner-join view.

1 Introduction

Queries containing outer joins are common in OLAP applications, typically joining a fact table with some number of dimension tables followed by aggregation. Outer-join queries are also used for constructing tree-structured objects (e.g. XML) from data stored in flat tables. Outer joins are needed so we can also retain objects that lack some subobjects.

Materialized views can speed up query processing greatly, but to realize the benefits two subproblems must be solved: view matching and incremental view maintenance. The goal of view matching is to determine, at optimization time, whether and how part or all of a query can be computed from a view. Incremental view maintenance is required to efficiently bring a view up to date when base tables are updated.

View matching and efficient incremental view maintenance algorithms for SPJG views, that is, views composed of selection, projection and *inner joins* with an optional group-by on top, are well understood. Our goal is to extend view support to SPOJG view, that is, allow views to also contain *outer joins*.

We introduced a view matching algorithm for outer-join views in a previous paper [6]. In this paper, we introduce an

Example 1 Suppose we create a view, *oj view*, as shown below. The view consists of outer joins of the tables *part*, *lineitem* and *orders* from the TPC-H database [10]. Recall that *p partkey* is the primary key of *part* and *o orderkey* is the primary key of *orders*. There is a foreign key constraint between *lineitem* and *part*, and also one between *lineitem* and *orders*.

```
create view oj view as
select p partkey, p name, p retailprice, o orderkey,
    o custkey, l linenumber, l quantity, l extendedprice
from part full outer join
    (orders left outer join lineitem on l orderkey=o orderkey)
    on p partkey=l partkey
```

We first analyze what types of tuples the view may contain. The join between *orders* and *lineitem* may output tuples of two types: *{orders, lineitem}* and *{orders}*. Each *lineitem* tuple joins with exactly one *orders* tuple (because of the foreign key from *l orderkey* to *o orderkey*) and produces an *{orders, lineitem}* tuple. *orders*-only tuples are *orphaned* orders (no matching *lineitem* tuples), which occur in the result null-extended on all *lineitem* columns.

Now consider what happens with the tuples in the second join. Because of the foreign key from *lineitem* to *part*, every *{orders, lineitem}* tuple will join with a *part* tuple, producing a *{part, orders, lineitem}* tuple. All orphaned *order* tuples are null extended on the join column *l partkey* so they will not join with any *part* tuples. However, the full outer join retains the orphaned *order* tuples. Similarly, a *part* may not join with any *lineitem* tuples but such orphaned *part* tuples are retained by the outer join. In summary, the view may contain tuples of three types:*{part, orders, lineitem}*, *{orders}*, and *{part}*.

Now suppose we insert new tuples into the *part* table. The view can then be brought up to date simply by inserting the new tuples, appropriately extended with nulls, into the view. Nothing more is required because the foreign key constraint between *lineitem* and *part* guarantees that a new part tuple cannot join with any *lineitem* tuples. If

efficient incremental maintenance procedure. We show that maintenance can be divided into two steps: computing and applying a *primary delta* and a *secondary delta*. The first step is very similar to maintaining an inner-join view. The second step is a "clean-up" step and we show how to perform this step efficiently. We also describe how foreign key constraints can be exploited to reduce maintenance overhead.


```
select p partkey, p name, p retailprice, o orderkey,
      o custkey, l linenumber, l quantity, l extendedprice
into #delta1
from new lineitems, orders, part
where l orderkey = o orderkey and l partkey = p partkey

insert into oj view -- apply primary delta --
select * from #delta1

delete from oj view -- apply secondary delta --
where l linenumber is null
and (p partkey in (select p partkey from #delta1) or
            o orderkey in (select o orderkey from #delta1))
```

The first statement computes the set of tuples to be inserted into the view and saves them in a temporary table. The second statement adds the new tuples into the view. The new *lineitem* tuples may cause some orphaned *part* or *orders* tuples to be eliminated from the view. The third statement deletes all orphaned *part* and *orders* tuples, if any, that cease to be orphans because of the insertions

Two earlier papers [2, 5] describe algorithms for incremental maintenance of outer join views. However, the algorithm in [2] is significantly more expensive than ours and the algorithm in [5] is incorrect. Oracle [8] supports materialized outer-join views but with many restrictions and only a limited class of outer-join views are incrementally maintainable. We discuss related work in more detail in Section 8.

The rest of the paper is organized as follows. Section 2 contains preliminary material and introduces concepts used in later sections. The overall maintenance procedure is described in Section 3. We show how to efficiently compute the primary delta in Section 4 and the secondary delta in Section 5. Foreign-key constraints are considered in Section 6. We report experimental results in Section 7 and describe related work in Section 8. Due to space limitations, we have state our results without proofs. Derivations and proofs can be found in [7].

2 Preliminaries

We assume that base tables and views satisfy the following restrictions: every base table has a unique key that does not contain nulls; a view can reference the same table only once (no self-joins); every view outputs a unique key (no duplicates) ; and all predicates of a view are null-rejecting (defined below).

2.1 Definitions and Notation

The selection operator is denoted in the normal way as σ_p where p is a predicate. A predicate p referencing some set S of columns is said to be *strong* or *null-rejecting* if it evaluates to false on a tuple as soon as one of the columns in S is null. Projection (without duplicate elimination) is denoted by π_c where c is a list of columns. Borrowing from SQL, we use the shorthand $T.*$ to denote all columns of it did, the joining *lineitem* tuples would have violated the foreign key constraint. Insertions into the *orders* table can be handled in the same way.

Next consider insertions into the *lineitem* table. Suppose the new lineitems are contained in a table *new lineitems*. The view can then be updated using the following sequence of statements.

denoted by T_1 T_2, first null-extends (pads with nulls) the tuples of each operand to schema $S_1 \cup S_2$ and then takes the union of the results (without duplicate elimination).

A tuple t_1 is said to *subsume* a tuple t_2 if they are defined on the same schema, t_1 agrees with t_2 on all columns where they both are non-null and t_1 contains fewer null values than t_2. The operator *removal of subsumed tuples* of T, denoted by $T\downarrow$, returns the tuples of T that are not subsumed by any other tuple in T.

The *minimum union* of tables T_1 and T_2 is defined as $T_1 \oplus T_2 = (T_1 \quad T_1)\downarrow$. It can be shown that minimum union is both commutative and associative.

Let T_1 and T_2 be tables with disjoint schemas S_1 and S_2, respectively, and p a predicate referencing some subset of the columns in $(S_1 \cup S_2)$. The *left semijoin* is defined as $T_1 \ {}^{ls}_{p}\ T_2 = \{t_1 | t_1 \in T_1, (\exists\, t_2 \in T_2 | p(t_1,t_2))\}$, that is, a tuple in T_1 qualifies if it joins with some tuple in T_2. The *left anti(semi)join* is $T_1 \ {}^{la}_{p}\ T_2 = \{t_1 | t_1 \in T_1, (\ t_2 \in T_2 | p(t_1,t_2))\}$, that is, a tuple in T_1 qualifies if it does not join with any tuple in T_2. The *left outer join* is $T_1 \ {}^{lo}_{p}\ T_2 = T_1 \ {}_{p}\ T_2 \oplus T_1$. The *right outer join* is $T_1 \ {}^{ro}_{p}\ T_2 = T_1 \ {}_{p}\ T_2 \oplus T_2$. The *full outer join* is $T_1 \ {}^{fo}_{p}\ T_2 = T_1 \ {}_{p}\ T_2 \oplus T_1 \oplus T_2$.

We will make use of a special predicate $null(T)$ that evaluates to true if a tuple is null-extended on table T. $null(T)$ can be implemented in SQL as "$T.c$ is null" where c is any column of T that does not contain nulls, for example, a column of a key. When applying $null$ and $\neg null$ to a set of tables $T = \{T_1, T_2, \cdots, T_n\}$, we use the shorthand notation $n(T) = \bigwedge_{T_i \in T} null(T_i)$ and $nn(T) = \bigwedge_{T_i \in T} \neg null(T_i)$.

2.2 Join-Disjunctive Normal Form

Our derivation of view maintenance expressions builds on the join-disjunctive normal form for SPOJ expressions introduced by Galindo-Legaria [1]. We briefly describe the normal form by an example, but refer the reader to [1, 6] for more details and algorithms.

Example 2 We will use the following view as a running example throughout the paper

$$V_1 = (R \ {}^{fo}_{p(r,s)}\ S) \ {}^{lo}_{p(r,t)}\ (T \ {}^{fo}_{p(t,u)}\ U). \qquad (1)$$

All four tables have a unique key and all predicates are null-rejecting. The notation $p(r, s)$ means a predicate over columns from tables R and S, and similarly for the other predicates.

We convert the view expression to normal form bottom up. We first rewrite the join between R and S and the join between T and U in terms of inner joins and minimum union, which results in

$$V_1 = (\sigma_{p(r,s)}(R \times S) \oplus R \oplus S) \ {}^{lo}_{p(r,t)}\ (\sigma_{p(t,u)}(T \times U) \oplus T \oplus U).$$

table T. We also need an operator that removes duplicates, which we denote by δ.

A schema S is a set of columns. Let T_1 and T_2 be tables with schemas S_1 and S_2, respectively. The *outer union*,

the right operand. Tuples from $\sigma_{p(t,u)}(T \times U)$ may join with tuples from $\sigma_{p(r,s)}(R \times S)$ producing tuples that satisfy $\sigma_{p(r,s)\wedge p(r,t)\wedge p(t,u)}(T \times U \times R \times S)$. Similarly, tuples from $\sigma_{p(t,u)}(T \times U)$ may join with tuples from term R, producing a term $\sigma_{p(r,t)\wedge p(t,u)}(T \times U \times R)$. However, tuples from $\sigma_{p(t,u)}(T \times U)$ do not join with tuples from term S because tuples from the S term are null extended on R and the join predicate $p(r, t)$ is null rejecting.

We continue the "multiplication" process with other terms and, because the join is a left outer join, also add the terms from the left input. This produces the following normal form of the view

$$V_1 = \sigma_{p(r,s)\wedge p(r,t)\wedge p(t,u)}(T \times U \times R \times S) \oplus \sigma_{p(t,u)\wedge p(t,u)}(T \times U \times R) \oplus \sigma_{p(r,t)\wedge p(r,s)}(T \times R \times S) \oplus \sigma_{p(r,t)}(T \times R) \oplus \sigma_{p(r,s)}(R \times S) \oplus R \oplus S$$

As illustrated by this example, an SPOJ expression E over a set of tables U can be converted to a normal form consisting of the minimum union of terms composed from selections and inner joins (but no outer joins). More formally, the join-disjunctive normal form of E equals

$$E = E_1 \oplus E_2 \oplus \cdots \oplus E_n$$

where each term E_i is of the form

$$E_i = \sigma_{p_i}(T_{i1} \times T_{i2} \times \cdots \times T_{im})$$

$T_{i1}, T_{i2} \cdots T_{im}$ is a subset of the tables in U. Predicate p_i is the conjunction of a subset of the selection and join predicates found in the original form of the query.

The derivation of the normal form and a conversion algorithm, can be found in [6]. The algorithm is straightforward and traverses the operator tree once. It exploits null-rejecting predicates and foreign keys to reduce the number of terms. For a tree consisting of N full outer joins, the normal form may contain $2^N + N$ terms in the worst case. In practice, it normally contains far fewer terms.

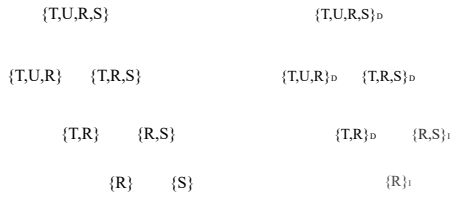
2.3 The Subsumption Graph

Suppose the complete set of operand tables for an SPOJ expression is U. Each term in the normal form is defined over a *unique* subset S of U and hence produces tuples that are null extended on $U-S$. We call the tables in subset S the term's *source tables*.

A tuple produced by a term with source table set S can only be subsumed by tuples produced by terms whose source set is a superset of S, see Lemma 2 in [6]. The subsumption relationships among terms can be modeled by a DAG, which we call the *subsumption graph*.

Definition 2.1 *Let $E = E_1 \oplus \cdots \oplus E_n$ be the join-disjunctive form of an SPOJ expression. The* subsumption graph *of E contains a node n_i for each term E_i in the normal form and the node is labeled with the source table set S_i of E_i. There is an edge from node n_i to node n_j, if S_i is a minimal superset of S_j. S_i is a minimal superset of S_j if there does not exist a node n_k in the graph such that $S_j \subset S_k \subset S_i$.*

To convert the remaining outer join, we "multiply" the two input expressions, that is, we consider every combination of a term from the left operand and a term from



(a) Subsumption (b) Maintenance (update T)

Figure 1. Subsumption and maintenance graphs for view V_1

2.4 Net Contribution of a Term

The minimum-union operators in the join-disjunctive normal form have two functions: to eliminate subsumed tuples and to union the remaining tuples. If we first eliminate subsumed tuples from every term, we can replace the minimum unions by outer unions. The resulting form clearly shows what terms are affected by an update and how.

Subsumption among terms are not arbitrary; when checking whether a tuple of a term is subsumed, it is sufficient to check against tuples in (immediate) parent terms. The following lemma shows how to eliminate subsumed tuples from a term.

Lemma 1 *Let E_i be a term with source set T_i in the normal form E of an SPOJ expression. Then the set of tuples generated by E_i that are not subsumed by any other tuples in E can be computed as*

$$D_i = E_i \; {}^{la}_{eq(T_i)} (E_{i1} \; E_{i2} \; \cdots E_{im})$$

where $E_{i1}, E_{i2}, \cdots E_{im}$ are the parent terms of E_i and $eq(T_i)$ is an equijoin predicate over columns forming a key of E_i.

We call D_i the *net contribution* of term E_i because the tuples of D_i are not subsumed by any other tuples and thus appear explicitly in the view result.

Theorem 1 *Let E be an SPOJ expression with normal form $E_1 \oplus E_2 \oplus \cdots \oplus E_n$. Then*

$$E = E_1 \oplus E_2 \oplus \cdots \oplus E_n = D_1 \; D_2 \; \cdots D_n$$

where each D_i is computed from E_i as defined in Lemma 1.

We call the form $D_1 \; \cdots D_n$ the *net-contribution form* of the expression. Because the terms are connected by (outer) unions, there is no interaction among net contributions from different terms so each term can be maintained independently from other terms.

3 View Maintenance Procedure

This section describes our overall maintenance procedure. We consider only with insertions or deletions; an up-

The subsumption graph for V_1 is shown in Figure 1(a).

3.1 Terms Affected by an Update

Consider a view V and suppose one of the its base tables T is modified. This may change the net contribution of a term D_i only if T occurs in the expression defining D_i. By inspection of the expression for D_i with source table set T_i, it is immediately apparent that the change may affect the result in one of three ways:

1. *Directly*, which occurs if T is among the tables in T_i;
2. *Indirectly*, which occurs if T is not among the tables in T_i but it is among the source tables of at least one of its parent nodes;
3. No effect, otherwise.

Based on this classification of how terms are affected, we create a *view maintenance graph* as follows.

1. Eliminate from the subsumption graph all nodes that are unaffected by the update of T.
2. Mark the remaining nodes by D or I depending on whether the node is affected directly or indirectly.

The maintenance graph for view V_1 when updating T is shown in Figure 1(b).

A node n in the maintenance graph may have multiple parents. We denote the set of parents by $par(n)$. Some of the parents may be directly affected, denoted by $pard(n)$, and some of them may be indirectly affected, denoted by $pari(n)$. If node n is directly affected, $pari(n) = \emptyset$. If node n is indirectly affected, $pard(n) \geq 1$. The equality $par(n) = pard(n) \cup pari(n)$ holds by definition.

3.2 Maintenance Procedure

Suppose table T has been updated and we need to maintain a view V that references T. We first compute the maintenance graph and classify the terms as directly affected, indirectly affected and unaffected. Without loss of generality, assume that the view has n terms, of which terms $1, 2, \cdots, k$ are directly affected, terms $k+1, k+2, \cdots, k+m$ are indirectly affected, and terms $k+m+1, k+m+2, \cdots, n$ are not affected. We can then rewrite the view expression in the form

$$V = V_D \quad V_I \quad V_U \quad \text{where}$$

$$V_D = \bigoplus_{i=1}^{k} D_i, \; V_I = \bigoplus_{i=k+1}^{k+m} D_i, \; V_U = \bigoplus_{i=k+m+1}^{n} D_i.$$

From this form of the expression it is obvious that to update the view we need to compute two delta expressions

$$\Delta V_D = \bigoplus_{i=1}^{k} \Delta D_i, \; \Delta V_I = \bigoplus_{i=k+1}^{k+m} \Delta D_i.$$

We call ΔV^D the *primary delta* and ΔV^I the *secondary delta*.

In summary, maintenance of a view V after updates to one of its underlying base tables is performed in two steps.

1. If there are directly affected terms, compute the primary delta ΔV^D and apply it to the view.
2. If there are indirectly affected terms, compute the secondary delta ΔV^I and apply it to the view.

If the update is an insert (delete), the primary delta is inserted into (deleted from) the view and the secondary delta is deleted from (inserted into) the view. In the following sections, we describe how to efficiently compute the primary delta and the secondary delta, respectively.

3.3 Aggregation Views

An aggregated outer-join view is simply an outer-join view with a group-by on top. Maintaining an aggregated outer-join view is not much more complex then maintaining a non-aggregated view. ΔV^D for the non-aggregated part of the view is computed in the same way. The result is then aggregated as specified in the view definition and applied to the view in the same way as for aggregated inner-join views. The view needs to contains both a regular row count and a not-null count for every table that is null-extended in some term. New rows are created as needed. Any row whose row count becomes zero is deleted. If the not-null count for table T becomes zero, all aggregates referencing a column in T are set to null.

Next we compute ΔV^I, aggregate the result and apply it to the view. However, we may have to compute ΔV^I from base tables as we shall see in Section 5.3 because it may not be possible to extract a required term from the aggregated view. Tuples from different terms that have been combined into the same group can no longer be separated out. Such computation may incur additional overhead. Further discussion of aggregation views and additional simplification rules are described in [7].

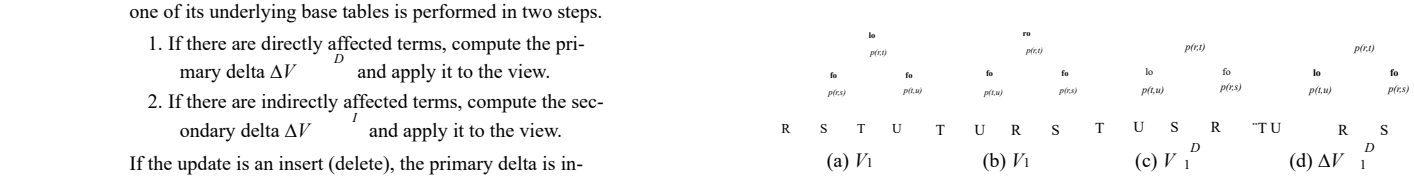
4 Computing the Primary Delta

Suppose table T has been updated and we need to maintain a view V that references T. Every term E_i in V^D has T as one of its source tables, so no tuples in V^D are null extended on T. Conversely, all tuples in V^I and V^U are null extended on T because the terms in V^I and V^U do not reference T. A tuple that is null-extended on a table T cannot subsume a tuple that is not null-extended on T. It follows that a tuple generated by a term in V^D can only be subsumed by a tuple generated by another term in V^D so we can rewrite V^D as

$$V_D = \bigoplus_{i=1}^{k} D_i = \bigoplus_{i=1}^{k} E_i.$$

$V^D = \bigoplus_{i=1}^{k} E_i$ contains all terms that produce tuples containing real tuples from table T. We now show a simple conversion of the original (non-normalized) view expression into an expression equal to V^D that can then trivially be converted into an expression for computing ΔV^D.

Example 3 Suppose table T has been updated and V_1 needs to be maintained. We derive expressions for V_1^D and ΔV_1^D through a series of transformations of expression (1). The original operator tree is shown in Figure 2(a). We tra-



(a) V_1 (original) (b) V_1 (commuted) (c) V_1^D (d) ΔV_1^D

Figure 2. Transforming V_1 to ΔV_1^D

verse the path from T to the root of the tree. On each join

Page 5

operator encountered we commute the inputs, if needed, so that the input expression referencing T is on the left. The only operator affected is the root operator where we swap the inputs and change the join from a left outer to a right outer join. The resulting operator tree is shown in Figure 2(b). This transformation converts the expression to

$$V_1 = (T \overset{fo}{\underset{p(t,u)}{\bowtie}} U) \overset{ro}{\underset{p(r,t)}{\bowtie}} (R \overset{fo}{\underset{p(r,s)}{\bowtie}} S). \quad (2)$$

Now consider the operators on the leftmost path in Figure 2(b). The join $T \overset{fo}{\underset{p(t,u)}{\bowtie}} U$ may produce three types of tuples: *TU*, *T* and *U*. All *U*-only tuples are null extended on *T* and, hence, can never become part of V_1^D because no tuples in V_1^D are null extended on *T*. Tuples of type *TU* and *T* will, if they survive the next join, become part of V_1^D because they contain "real" *T* tuples. We can eliminate the *U*-only tuples by changing the join to a left outer join.

The next join on the path is $\overset{ro}{\underset{p(r,t)}{\bowtie}}$. After the modification, its left input produces only tuples destined for V_1^D. Its right input may produce tuples of types *RS*, *R* and *S*. Because the join is a right outer join, it preserves unmatched tuples from the right input. However, they are null extended on *T* and, hence, cannot become part of V_1^D. We can eliminate the unmatched tuples by changing the join from a right outer to an inner join, as shown in Figure 2(c).

The modified expression produces exactly the same tuples as the original expression for all terms containing actual *T* tuples, that is, all terms in V_1^D, and no tuples that are null extended on *T*. Furthermore, none of the retained tuples were ever subsumed by a tuple in a term eliminated by modifying the joins. It follows that the modified expression exactly computes V_1^D, that is,

$$V_{1D} = (T \overset{lo}{\underset{p(t,u)}{\bowtie}} U) \overset{ro}{\underset{p(r,t)}{\bowtie}} (R \overset{fo}{\underset{p(r,s)}{\bowtie}} S). \quad (3)$$

The leftmost path in Figure 2(c) contains only a left outer join and an inner join. As explained further below, an expression for ΔV_1^D can be obtained simply by substituting ΔT for *T* in expression (3), that is,

$$\Delta V_{D1} = (\Delta T \overset{lo}{\underset{p(t,u)}{\bowtie}} U) \overset{ro}{\underset{p(r,t)}{\bowtie}} (R \overset{fo}{\underset{p(r,s)}{\bowtie}} S). \quad (4)$$

The corresponding operator tree is shown in Figure 2(d).

The simple procedure for constructing expressions for V^D and ΔV^D illustrated in this example generalizes to arbitrary SPOJ views. The general algorithm follows.

Algorithm: Construct ΔV^D expression
Inputs: Original view expression V, updated table T.
Output: Expression for computing ΔV^D.

1. Traverse the operator tree for V along the path from T to the root. On any join operator encountered, apply commutativity rules to ensure that the input referencing T is on the left.
2. Traverse the path from T to the root of V. Convert any full outer join operator encountered to a left outer join and any right outer join operator to an inner join.
3. Substitute T by ΔT.

Step 1 is a normal rewrite of the view expression and does not change the result. Step 2 modifies the expression so that it discards all tuples that cannot become part of V^D. After Step 2, the operators on the path from *T* to the root consists only of selects, inner joins and left outer joins and the delta expression is always the left input. The correctness of Step 3 follows from the following delta propagation rules.

$$\sigma_p(e_1 \pm \Delta e_1) = \sigma_p e_1 \pm \sigma_p \Delta e_1$$

$$(e_1 \pm \Delta e_1) \underset{p}{\bowtie} e_2 = e_1 \underset{p}{\bowtie} e_2 \pm \Delta e_1 \underset{p}{\bowtie} e_2$$

$$(e_1 \pm \Delta e_1) \overset{lo}{\underset{p}{\bowtie}} e_2 = e_1 \overset{lo}{\underset{p}{\bowtie}} e_2 \pm \Delta e_1 \overset{lo}{\underset{p}{\bowtie}} e_2$$

where ± stands for either a set union or a set difference. The rules for selects and inner joins are obvious. The rule for left outer join can be found in [2].

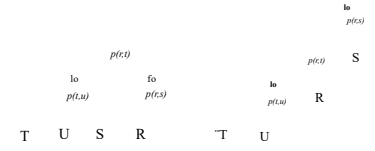
4.1 Conversion to a Left-Deep Tree

In many cases, only a few tuples are inserted or deleted at a time and only a small number of tuples are affected in the view. The expression for ΔV^D produced by the algorithm above are not always efficient for such cases because it may contain subexpressions joining two or more base tables. Joining two base tables may produce a large intermediate result even though the final result is small. We show how to convert the expression to a left-deep join tree that avoids this problem.

Ideally, the optimizer should consider this conversion automatically but current optimizers are deficient in this area. We introduce two new associativity rules for outer joins (rules 4 and 5 below). These additional rules make it possible to always convert the delta expression to a left-deep tree *provided that all join predicates are binary, that is, reference only two tables*.

Example 4 When *T* is updated, our algorithm produces the following expression for ΔV_1^D

$$\Delta V_{D1} = (\Delta T \overset{lo}{\underset{p(t,u)}{\bowtie}} U) \overset{lo}{\underset{p(r,t)}{\bowtie}} (R \overset{fo}{\underset{p(r,s)}{\bowtie}} S). \quad (5)$$



(a) Bushy tree (b) Left-deep tree

Figure 3. Converting ΔV_1^D to a left-deep tree

The operator tree is shown in graphical form in Figure 3(a). This expression is potentially very expensive to compute. Suppose ΔT is very small, containing only a few tuples. Then the join $\Delta T \overset{lo}{\underset{p(t,u)}{\bowtie}} U$ is likely to produce a small result. This small result is the left input to the final join $\underset{p(r,t)}{\bowtie}$ so the final result is also likely to produce a small result. However, the right operand is a join involving base tables only, namely, $R \overset{fo}{\underset{p(r,s)}{\bowtie}} S$. This join may be expensive to compute and will produce a result that is at least as large as the maximum of *R* and *S*.

We can eliminate the potentially large intermediate result by converting the operator tree to a left-deep tree, that is, a tree where the right operand of every join is a single base table, possibly including a select. This can be done by exploiting associativity rules for inner and outer joins. Applying this transformation to expression (5) produces

$$\Delta V_{D_1} = ((\Delta T \ ^{lo}_{p(t,u)}\ U) \ ^{lo}_{p(r,t)}\ R) \ ^{lo}_{p(r,s)}\ S \qquad (6)$$

The corresponding operator tree is shown in Figure 3(b). Using this expression, the intermediate results are likely to stay small if ΔT is small.

In the expression for ΔV^D, the operators on the path from the updated table to the root (the leftmost path) are limited to selection, inner join and left outer join. We convert the tree to a left-deep tree by repeatedly applying the following simple procedure: for any join operator on the leftmost path whose right operand references more than one table, pull the top operator of the right operand into the main path by applying one of the associativity rules listed below. The rules assume that all join predicates are null-rejecting and reference only two tables but they are not required to be equijoins. The notation $p(1, 2)$ means a predicate that references columns in e_1 and in e_2, and similarly for $p(2, 3)$.

We need a new but simple operator called the *null-if operator* and denoted by λ^c_p where p is predicate and c a list of column. For every tuple that satisfies p, the operator sets the value of all columns in c to null; all other tuples are passed through unchanged. The operator can implemented using a project with the case statement of SQL. A null-if operator may create duplicates, which need to be eliminated.

Associativity rules for left outer join

(1) $e_1 \ ^{lo}_{p(1,2)} (\sigma_{p(2)} e_2) = \delta\, \lambda^{*}_{e2.*,\ \neg p(2)} (e_1 \ ^{lo}_{p(1,2)} e_2)$

(2) $e_1 \ ^{lo}_{p(1,2)} (e_2 \ ^{fo}_{p(2,3)} e_3) = (e_1 \ ^{lo}_{p(1,2)} e_2) \ ^{lo}_{p(2,3)} e_3$

(3) $e_1 \ ^{lo}_{p(1,2)} (e_2 \ ^{lo}_{p(2,3)} e_3) = (e_1 \ ^{lo}_{p(1,2)} e_2) \ ^{lo}_{p(2,3)} e_3$

(4) $e_1 \ ^{lo}_{p(1,2)} (e_2 \ ^{ro}_{p(2,3)} e_3) = \delta\, \lambda^{*}_{e2.*,e3.*,\ \neg p(2,3)} ((e_1 \ ^{lo}_{p(1,2)} e_2) \ ^{lo}_{p(2,3)} e_3)$

(5) $e_1 \ ^{lo}_{p(1,2)} (e_2 \ ^{lo}_{p(2,3)} e_3) = \delta\, \lambda^{*}_{e2.*,e3.*,\ \neg p(2,3)} ((e_1 \ ^{lo}_{p(1,2)} e_2) \ ^{lo}_{p(2,3)} e_3)$

Note that all joins added to the main path are inner joins and left outer joins. The null-if operators fix up tuples that are supposed to be null-extended and duplicates, if any, thus created are eliminated. To the best of our knowledge, rules 1, 4 and 5 are new.

5 Computing the Secondary Delta

The secondary delta can be computed efficiently from the primary delta and either the view or base tables – we consider both options. When possible, it is usually cheaper to use the view but the optimizer should choose in a cost-based manner. Recall that the base tables have already been updated and the primary delta has been applied to the view.

Every term in a view has a unique set of source tables and is null-extended on all other tables in the view. We denote the source tables of term E_i by T_i and the set of tables on which it is is null-extended by S_i.

5.1 Extracting Term Deltas from ΔV^D

The primary delta ΔV^D contains the union of the deltas for all directly affected terms. However, we need the deltas for individual terms to compute the secondary delta. Each term is defined over a unique set of tables and null extended on all others so tuples from a particular term are easily identified and can be extracted from ΔV^D by a combination of *null* and $\neg$*null* predicates.

Example 5 ΔV^D_1 contains the deltas for four directly affected terms, see Figure 1(b). Consider, for example, the TRS-term. Non-subsumed tuples from this term are uniquely identified by the fact that they are composed of real tuples from T, R, and S but are null extended on U. Hence, ΔD_{TRS} can be extracted from ΔV^D_1 as follows

$$\Delta D_{TRS} = \pi_{(TRS).*} \sigma_{nn(TRS) \wedge n(U)} \Delta V^D_1$$

where $nn(TRS) = \neg null(T) \wedge \neg null(R) \wedge \neg null(S)$ and $n(U) = null(U)$.

ΔD_{TRS} contains only the delta of the net contribution. ΔE_{TRS} contains the complete delta of the term, including both subsumed and non-subsumed tuples. Tuples in ΔE_{TRS} are composed of real tuples from T, R, and S and may or may not be null extended on U. Hence, ΔE_{TRS} can be extracted from ΔV^D_1 as follows

$$\Delta E_{TRS} = \delta\, \pi_{(TRS).*} \sigma_{nn(TRS)} \Delta V^D_1$$

The duplicate elimination (δ) is necessary because a TRS tuple may have joined with multiple U tuples.

Theorem 2 *Consider a view V defined over tables U. Let E_i be a term in V^D defined over tables T_i, and D_i its net contribution. Then ΔD_i and ΔE_i can be computed as*

$$\Delta D_i = \pi_{Ti.*} \sigma_{nn(Ti) \wedge n(U-Ti)} \Delta V^D$$

$$\Delta E_i = \delta\, \pi_{Ti.*} \sigma_{nn(Ti)} \Delta V^D$$

where $nn(T_i) = \bigwedge_{t \in T_i} \neg null(t)$ *and* $n(U-T_i) = \bigwedge_{t \in (U-T_i)} null(t)$.

5.2 Computing ΔV^I Using the View

We first consider how to compute ΔV^I from the primary delta and the view. After applying the primary delta, the state of the view is $V + \Delta V^D$ or $V - \Delta V^D$. ΔD_i denotes the change in the net contribution of the indirectly affected term E_i.

Insertions: After an insertion, ΔD_i can be computed from the view and the primary delta by the expression

$$\Delta D_i = \sigma_{nn(Ti) \wedge n(Si)} (V + \Delta V^D) \ ^{ls}_{P_i =} \bigvee_{E k \in pard(E_i)} nn(T_k) \ \sigma_{eq(Ti)\, P_i} \Delta V^D$$

T_i denotes the source table set of E_i and S_i the set of tables on which E_i is null extended. E_k ranges over all directly affected parents of E_i and T_k denotes the source table set of E_k. $eq(T_i)$ denotes an equijoin condition between the key columns of T_i in the left operand and in the right operand.

This expression makes sense intuitively. The first part selects from the view all orphaned (non-subsumed) tuples of term E_i, that is, the tuples in D_i. The second part extracts from the primary delta all tuples added to a parent term of E_i. The complete expression thus amounts to finding all currently orphaned tuples of the term and deleting those that cease to be orphans because of the insert.

Example 6 Continuing with our running example, we need to compute ΔD_{RS} and ΔD_R. D_{RS} is null extended on T and U and the TRS-term is its only parent, so ΔD_{RS} can be computed as

$$\Delta D_{RS} = \sigma_{nn(RS) \wedge n(TU)}(V_1 + \Delta V_D \ltimes_{eq(RS)} \sigma_{nn(TRS)} \Delta V_D)$$

D_R is null extended on S, T and U and it has one directly affected parent, the TR-term so ΔD_R can be computed as

$$\Delta D_R = \sigma_{nn(R) \wedge n(STU)}(V_1 + \Delta V_D \ltimes_{eq(R)} \sigma_{nn(TR)} \Delta V_D)$$

Deletions: After a deletion, ΔD_i can be computed from the view and the primary delta using the expression

$$\Delta D_i = (\delta \pi_{T_i * \sigma_{P_i}} \Delta V_D) \ltimes_{eq(T_i)} (V - \Delta V_D)$$

where P_i is the same as for the insertion case.

This expression also makes sense intuitively. The first part extracts from the primary delta the tuples deleted from parents of term E_i, projects them onto the tables of term E_i, and eliminates duplicates. This produces the potentially orphaned tuples of E_i. The anti-semijoin then discards every tuple that is still included in a parent tuple. This leaves the actual new orphans to be inserted.

Example 7 After deletions from table T, the delta of the indirectly affected terms of V_1 can be computed as follows.

$$\Delta D_R = (\delta \pi_{R * \sigma_{nn(TR)}} \Delta V_D) \ltimes_{eq(R)} (V_1 - \Delta V_D)$$

$$\Delta D_{RS} = (\delta \pi_{(RS) * \sigma_{nn(TRS)}} \Delta V_D) \ltimes_{eq(RS)} (V_1 - \Delta V_D)$$

Column availability: If a view does not output the columns required by the expressions above, then the expression cannot be used and ΔD_i has to be computed using base tables. The join predicates require access to the key columns of the referenced tables. For insertions, tuples are extracted from the view using a combination of *null* and $\neg$*null* predicates against source tables. However, the view may not output a non-null column for each of the referenced source table. Even so, it may still be possible to extract the required tuples from the view. The exact conditions when extraction is still possible are derived in [6]. The key observation is that the view must expose enough non-null columns to uniquely distinguish the required tuples from tuples of all other terms.

5.3 Computing ΔV_1 from Base Tables

If the view does not output all required columns, the delta of a term cannot be computed from the view and the primary delta. If so, the term delta has to be computed from base tables, ΔT, and the primary delta. As before, we assume that the update has already been applied to table T and thus only the new state of the table is available. We denote the new state of the table by $T^{\pm}$; $T^{\pm} = T + \Delta T$ after an insertion and $T^{\pm} = T - \Delta T$ after a deletion.

Before proceeding we need to introduce some additional notation. Let $E_i = \sigma_{p_i}(S_{i1} \times \cdots \times S_{im})$ be an indirectly affected term under consideration. E_i has r directly affected parents $pard(E_i) = \{E_{i1}, \cdots, E_{ir}\}, r \geq 1$ and s, $s \geq 0$, indirectly affected parents $pari(E_i)$. Let E_k be one of the parent terms. Because E_k is a parent of E_i, we know that its source set contains $S_i = \{S_{i1}, \cdots, S_{im}\}$, and some other tables $R_k = \{R_{k1}, \cdots, R_{kn}\}$. Furthermore, if the parent is directly affected, it references T but not if it is indirectly affected. We split the expression for E_k into three parts:

$$E_k = \sigma_{p_k}(S_{i1} \times \cdots \times S_{im} \times R_{k1} \times \cdots \times R_{kn} \times T)$$
$$= \sigma_{p_k}(\underbrace{S_{i1} \times \cdots \times S_{im}}) \underbrace{\sigma_{q(S_i,R_k,T)} \sigma_{q(R_k)}(R_{k1} \times \cdots \times R_{kn})}_{\sigma_{q(R_k,T)}} \underbrace{\sigma_{q(T)}(T)}$$
$\leftarrow$ *missing for indirectly affected terms*

The new predicates are constructed from p_k as follows: $q(R_k)$ contains every conjunct of p_k that references only tables in R_k; $q(T)$ contains every conjunct of p_k that references table T only; $q(S_i, R_k, T)$ contains every conjunct that references at least one table among S_k and at least one table among $R_k \cup \{T\}$; and $q(R_k, T)$ contains every conjunct of p_k that references at least one table in R_k and T. ΔD_i can be computed from the last two parts of E_k and the primary delta.

Insertions After an insertion, ΔD_i can be computed as

$$\Delta D_i = (\delta \pi_{T_i * \sigma_{Q_i}} \Delta V_D) \ltimes_{q_{i1} E_{i1} \cdots \ltimes_{q_{ir}} E_{ir}}$$

where Q_i, E_{ip}, and q_{ip}, $p = 1, 2, \cdots, r$ are defined as

$$Q_i = nn(S_i) \wedge n(\cup_{E_k \in pari(E_i)} R_k)$$
$$E_{ip} = \sigma_{q(R_{ip})}(R_{i1} \times \cdots \times R_{im}) \ltimes_{q(R_{ip},T)} (\sigma_{q(T)} T^{\pm} \ltimes_{eq(T)} \Delta T)$$
$$q_{ip} = q(S_i, R_{ip}, T)$$

Example 8 After an insertion into table T, the tuples to be deleted from view V_1 can be computed as

$$\Delta D_R = (\delta \pi_{(R) * \sigma_{nn(TR) \wedge n(S)}} \Delta V_D) \ltimes_{p(r,t)} (T^{\pm} \ltimes_{eq(T)} \Delta T)$$

$$\Delta D_{RS} = (\delta \pi_{(RS) * \sigma_{nn(TRS)}} \Delta V_D) \ltimes_{p(r,t)} (T^{\pm} \ltimes_{eq(T)} \Delta T)$$

Let's see if the expression for ΔD_R makes sense. The expression $\delta \pi_{(R) * \sigma_{nn(TR) \wedge n(S)}} \Delta V_D$ extracts all R tuples in the primary delta that that did not join with any S tuples. These R tuples are no longer orphans but some of them may have been before the insertion. The expression $T^{\pm} \ltimes_{eq(T)} \Delta T$ represents all tuples in T *before* the insertion. An extracted R tuple satisfies the anti-semijoin if it does not join with any of the old T tuples, that is, if it were

Because of the foreign key constraint, no tuples in ΔT will

an orphan. All such prior orphans should be deleted from the view.

Deletions: After a deletion, ΔD_i can be computed as

$$\Delta D_i = (\delta \pi_{T_i \cdot *} \sigma_{Q_i} \Delta V_D) \; la_{q_{i1} E_{i1} \cdots la_{q_{ir}} E_{ir}}$$

where Q_i, E_{ip} and q_{ip}, $p = 1, 2, \cdots, r$ are defined as

$$Q_i = nn(S_i) \wedge n(\cup_{E_k \in part(E_i)} R_k)$$

$$E_{ip} = \sigma_{q(R_{ip})}(R_{i1} \times \cdots \times R_{in}) \; q(R_{ip}, T)(\sigma_{q(T)} T_{\pm})$$

$$q_{ip} = q(S_i, R_{ip}, T)$$

Example 9 Applying the formula above, we find that ΔD_R and ΔD_{RS} of our example view V_1 can be computed as

$$\Delta D_R = (\delta \pi_{(R)\cdot *} \sigma_{nn(TR) \wedge n(S)} \Delta V_D \; 1) \; la_{p(r,t)} T_{\pm}$$

$$\Delta D_{RS} = (\delta \pi_{(RS)\cdot *} \sigma_{nn(TRS)} \Delta V_D \; 1) \; la_{p(r,t)} T_{\pm}$$

Again, let's analyze the expression for ΔD_R. The expression $\delta \pi_{(R)\cdot *} \sigma_{nn(TR)} \Delta V^D_1$ extracts from the primary delta all deleted R tuples that do not join with an S tuple. These are the potential new R-only orphans. Any new orphan that does not join with a tuple remaining in T after the deletion is an actual orphan and is inserted into the view.

6 Exploiting Foreign Keys

In our first example, we exploited foreign-key constraints to conclude that the view could be maintained after insertion of a *part* tuple simply by inserting the new tuple into the view. The techniques we have developed so far would not recognize this opportunity. In this section we show to exploit foreign-key constraints to further simplify computation of the primary delta and the secondary delta.

However, the optimization described in this section cannot be applied under the following circumstances.

1. When an update is logically decomposed into a delete and an insert for the purpose of view maintenance. (The tuples in T may be only modified and there are no actual deletions and insertions.)
2. The constraint is declared with cascading deletes.
3. The constraint is deferrable and the insert/delete statement is part of a multi-statement transaction.

6.1 Simplifying ΔV_D Computation

Example 10 Consider our running example view V_1 but with a slight modification. We add a foreign key constraint from column $U.fk$ to column $T.pk$ where $T.pk$ is a primary key of T, and assume that the join predicate $p(t, u)$ equals $T.pk = U.fk$. The view definition then becomes

$$V_1 = (R \; fo_{p(r,s)} S) \; lo_{p(r,t)} (T \; fo_{pk=fk} U).$$

and our algorithms generate the primary delta expression

$$\Delta V_{D_1} = ((\Delta T \; lo_{pk=fk} U) \; p(r,t) R) \; lo_{p(r,s)} S \qquad (7)$$

join with tuples in U. Let $t \in \Delta T$ be a tuple that has been inserted into T. Tuple t has a unique pk value so there cannot exist a tuple $u \in U$ that references t. If such at tuple u existed, it would violate the foreign-key constraint. The same reasoning can be applied to a tuple t that is deleted from T. If a tuple u existed, it would violate the foreign-key constraint after the deletion.

As no tuples in ΔT join with U, the outer join $\Delta T \; lo_{pk=fk} U$ simply passes through the tuples from ΔT and can therefore be eliminated. Doing so reduces the expression to

$$\Delta V_{D_1} = (\Delta T \; p(r,t) R) \; lo_{p(r,s)} S$$

None of the other joins reference the discarded table U so no further modifications are needed.

Let $F_i, i = 1, \cdots, m$ be a foreign key constraint from a table S_i to the updated table T that matches a join in the expression for ΔV^D. To simplify the operator tree based on the foreign key constraints, call the procedure **SimplifyTree** below with inputs ΔV^D and $S = \{S_1, \cdots, S_m\}$.

Procedure: SimplifyTree(Tree *DT*, Set of Tables *S*)
Traverse DT from the leftmost leaf to the root. At each operator node n, do the following

1. If n is an inner join or a select with a predicate that is null-rejecting on a table $s \in S$, set $DT = \emptyset$ and return.
2. If n is a left outer join with a predicate that is null-rejecting on a table $s \in S$, eliminate node n and connect its left input to its parent. Let R denote the set of tables of the right input expression. Add R to S.

6.2 Simplifying ΔV_I Computation

Foreign key constraints can also be exploited to reduce the number of affected terms and potentially reduce the cost of computing ΔV^I. The following theorem summarizes how to use foreign-key constraints to detect additional terms that are unaffected by an update.

Theorem 3 *Consider a directly affected term with base S_i in the normal form of a SPOJ view and assume that a table $T \in S_i$ is updated by an insertion or deletion. The net contribution of the term is unaffected if S_i contains another table R with a foreign key referencing a non-null, unique key of T, and R and T are joined on this foreign key.*

We exploit this theorem to eliminate directly affected nodes and their edges from the maintenance graph. Elimination of directly affected nodes may leave an indirectly affected node without incoming edges, that is, without affected parents. Any such nodes can also be eliminated. We call the resulting graph the *reduced maintenance graph*.

Example 11 This optimization does not simplify the computation of ΔV^I_1 for our modified running example. Con-

sider instead view V_2 defined as follows

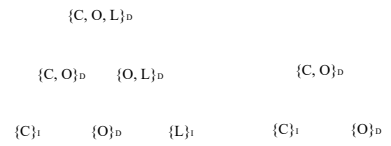
$$V_2 = \sigma_{pc} C \; fo_{ck=ock} (\sigma_{po} O \; fo_{ok=lok} L)$$
$$= \sigma_{pc \wedge po \wedge ck=ock \wedge ok=lok} (C \times O \times L) \oplus$$

full outer join part
on l partkey=p partkey and p retailprice < 2000

create unique clustered index V4 clu on
V4(c custkey, p partkey, l orderkey, l linenumber, o orderkey)

$$\sigma_{pc \wedge po \wedge ck=ock}(C \times O) \oplus \sigma_{po \wedge ok=lok}(O \times L) \oplus \sigma_{pc} C \oplus \sigma_{po} O \oplus L$$

The maintenance graph for updates to O in Figure 4(a) shows two indirectly affected terms so $\Delta V_2^I = \Delta D_C \oplus \Delta D_L$.



(a) Original graph (b) Reduced graph

Figure 4. V_2 **Maintenance graphs (update** O**).**

Now assume that there is foreign key constraint from *L.lok* to *O.ok*. Then an insertion into *O* will not affect nodes *OL*, and *COL*. Eliminating these two nodes, leaves node *L* without a parent so it is also eliminated. This leaves the reduced graph shown in Figure 4(b), which has only one indirectly affected term so $\Delta V_2^I = \Delta D_C$.

7 Experimental Results

We ran a series of experiments on Microsoft SQL Server 2005 Beta2 to evaluate the cost of maintaining outer join views. The experiments were performed on a workstation with two 3.2 GHz Xeon processors, 2GB of memory and three SCSI disks. All queries were against a 10GB version (SF=1) of TPC-H database.

View maintenance was implemented using insert and delete triggers that called stored procedures. The stored procedures followed the steps in our maintenance algorithm for outer-join views. We also compared the maintenance cost for a outer join view with the maintenance cost for the corresponding *core view*, which is the view obtained by replacing all outer joins with regular inner joins.

For each experiment, we measured the maintenance costs both with a warm buffer pool and with a cold buffer pool. The trends are quite similar so we only report the results with a cold buffer pool.

In the first experiment, we created an outer join view of the tables *customer*, *orders*, *lineitem*, *part*, as shown below. The corresponding core view contained inner joins of the four tables with the same join predicates. Both views had the same indexes.

```
create view V3 as
select l orderkey, l linenumber, l quantity,
    l extendedprice, l shipdate, l returnflag,
    o orderkey, o orderdate, o clerk,
    c custkey, c nationkey, c mktsegment,
    p partkey, p type, p retailprice
from ((select * from lineitem, orders
            where l orderkey=o orderkey and
            o orderdate between '1994-06-01' and '1994-12-31') lo
    right outer join customer
        on c custkey = o custkey)
```

```
create index V4 idx on
V4(p partkey, c custkey, l orderkey, l linenumber, o orderkey)
```

Term	*COLP*	*COL*	*C*	*P*
Cardinality	5208168	131702	184224	789131
Rows affected	4863	128	323	346

Table 1. Terms in view V_3 **and rows affected when inserting 60,000 lineitem rows**

The normal form of view V_3 contains four terms with the cardinalities shown in Table 1. Because of the foreign key constraint between *lineitem* and *orders*, insertion or deletion of order rows does not affect the view. When inserting (or deleting) customer rows, due to a foreign key constraint between *orders* and *customer*, we only need to add (or delete) the customer in the view. The resulting maintenance overhead for the view is very small and not reported here. Inserting or deleting customer rows or part rows has no effect on the core view.

However, updating *lineitem* can affect all four terms. The last line in Table 1 shows the number of rows affected when inserting 60*,* 000 rows into *lineitem*. The maintenance steps for V_3 after an insertion into *lineitem* are listed below. The table *Inserted* contains all the rows that were inserted into the *lineitem* table. Maintaining the view in case of deletion is similar but we omit the details due to lack of space.

Q_1: Compute primary delta

```
insert into #delta1
select l orderkey, l linenumber, l quantity,
    l extendedprice, l shipdate, l returnflag,
    o orderkey, o orderdate, o clerk,
    c custkey, c nationkey, c mktsegment,
    p partkey, p type, p retailprice
from ((select * from inserted, orders, customer
        where l orderkey=o orderkey and
        c custkey = o custkey and
        o orderdate between '1994-06-01' and '1994-12-31') lo
    left outer join part
        on l partkey=p partkey and p retailprice < 2000
```

Q_2: Apply primary delta

```
insert into V3
select * from #delta1
```

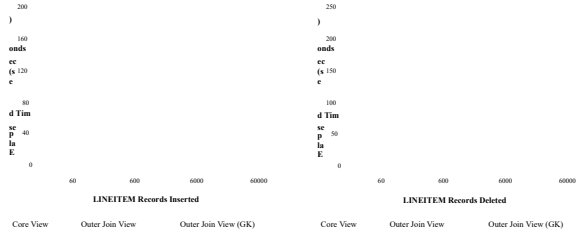
Q_3: Update term C

```
delete from V3
where o orderkey is null and l orderkey is null
and p partkey is null
and c custkey in (select c custkey from #delta1)
```

Q_4: Update term P

```
delete from V3
where c custkey is null and o orderkey is null
and l orderkey is null
and p partkey in (select p partkey from #delta1)
```

Figure 5 compares the maintenance cost of view V_3 with those of the corresponding core view when inserting 60, 600, 6,000 and 60,000 rows, respectively. The costs for the outer-join view are virtually the same as for the core view – the overhead for fixing up the *P* and *C* terms is very low.



new tuple may force removal of *both* an orphaned *part* tuple and an orphaned *orders* tuple from the view. This happens if the new *lineitem* tuple is the first line item of the order and nobody has ordered this particular part before. Gupta's and Mumick's algorithm would modify one of the tuples but not delete the other one, leaving the view in an incorrect state. It wrongly assumes that that the view also contains *{part, lineitem}* and *{orders, lineitem}* terms.

(a) Insertion (b) Deletion

Figure 5. Maintenance costs for V_3

The results for Griffin's and Kumar's (GK) algorithm [2] are also shown in Figure 5. Their maintenance expressions are quite complex. Their performance is similar to ours when the number of insertions is very small, but deteriorates dramatically with more insertions. For deletions their performance is much worse than ours. Gupta's and Mumick's algorithm [5] was not included in the experiment because it may produce an incorrect result.

These experiments show that our algorithms generate very efficient maintenance expressions. As a consequence, maintaining an outer-join view need not be more expensive than maintaining an inner-join view.

8 Related Work

It is well understood how to incrementally maintain views with inner joins. References [4] and [3] provide good overviews of the large body of work in this area. Much work has also been done on optimization of outer-join queries; for details see [9] and its references.

We are aware of only two earlier papers that describe algorithms for incremental maintenance of outer-join views. Griffin's and Kumar's algorithm [2] produces maintenance expressions of the correct form but they are incomplete because the predicates of the semi and anti-semi joins used are not specified. Getting the predicates right is not trivial. The experiments reported in Section 7 showed that their approach is significantly more expensive than ours. Their algorithm consistently produces maintenance expressions that are more complex and more expensive than ours. The main reasons are that (a) their expressions may involve joins of base tables only and may produce large intermediate results; (b) their expressions never exploit the view itself, everything is computed from base tables and (c) null-rejecting predicates and foreign keys are not exploited to deduce what terms are unaffected so (empty) deltas for many terms may be computed unnecessarily.

Gupta's and Mumick's algorithm [5] assumes than each directly affected tuple can subsume at most one indirectly affected tuple, which is incorrect. We can illustrate the problem using view *oj view* from the introductory section. The view contains tuples of three types only: *{part, orders, lineitem}*, *part*, and *orders*. Suppose we insert a new *lineitem* tuple. This causes insertion of a new *{part, orders, lineitem}* tuple into the view. However, the This flaw in the algorithm appears to be fundamental and not easily fixable.

9 Conclusion

We introduced an efficient incremental maintenance procedure for materialized outer-join views. Efficient incremental maintenance expressions are constructed for such views. The expressions are composed of regular algebraic operators – no new operators are needed. Exploiting a normal form and subsumption graphs enables us to precisely identify which terms are affected and how to maintain them, and therefore avoid unnecessary work. If foreign key constraints are available, they are also exploited to simplify maintenance. Experimental results show that maintaining an outer-join view is not necessarily more expensive than maintaining an inner-join view.

One direction for future work is to investigate even more efficient ways to compute ΔV^I. It may be possible to combine (parts of) the computations for the different terms, for example, by exploiting outer joins or by saving and reusing partial results.

References

[1] C. Galindo-Legaria. Outerjoins as disjunctions. In *SIGMOD Conference*, 1994.

[2] T. Griffin and B. Kumar. Algebraic change propagation for semijoin and outerjoin queries. *SIGMOD Record*, 27(3):22–27, 1998.

[3] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems,techniques, and applications. *Data Engineering Bulletin*, 18(2), 1995.

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

[5] H. Gupta and I. S. Mumick. Incremental maintenance of aggregate and outerjoin expressions. *Information Systems*, 31(6), 2006.

[6] P.-˚A. Larson and J. Zhou. View matching for outer-join views. In *VLDB Conference*, 2005.

[7] P.-˚A. Larson and J. Zhou. Maintenance of materialized outer-join views. Technical Report (to appear), Microsoft Research, 2006.

[8] Oracle Corp. *Oracle Database Data Warehousing Guide 10g Release 2*, 2006. http://download-west.oracle.com /docs/cd/B19306 01/server.102/b14223/basicmv.htm.

[9] J. Rao, H. Pirahesh, and C. Zuzarte. Canonical abstraction for outerjoin optimization. In *SIGMOD Conference*, 2004.

[10] Transaction Processing Performance Council. *TPC Benchmark H, (Decision Support), Revision 2.3.0*, 2005. http://www.tpc.org/tpch/spec/tpch2.3.0.pdf.