Proc. of 1991 VLDB Conference, pages 577–589

Deriving Production Rules for Incremental View Maintenance

Stefano Ceri*
Jennifer Widom

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
ceri@cs.stanford.edu, widom@ibm.com

Abstract. It is widely recognized that production rules in database systems can be used to automatically maintain derived data such as views. However, writing a correct set of rules for efficiently maintaining a given view can be a difficult and ad-hoc process. We provide a facility whereby a user defines a view as an SQL **select** expression, from which the system automatically derives set-oriented production rules that maintain a materialization of that view.

The maintenance rules are triggered by operations on the view's base tables. Generally, the rules perform incremental maintenance: the materialized view is modified according to the sets of changes made to the base tables, which are accessible through logical tables provided by the rule language. However, for some operations substantial recomputation may be required. We give algorithms that, based on key information, perform syntactic analysis on a view definition to determine when efficient maintenance is possible.

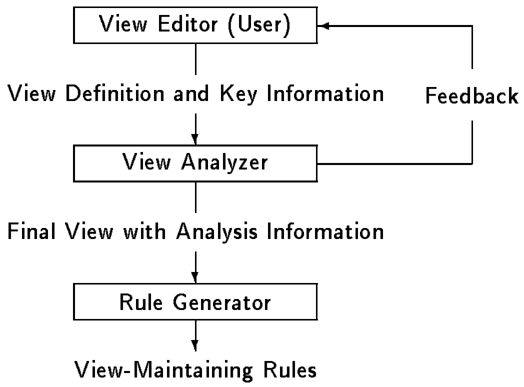

Figure 1: *Rule derivation system*

1 Introduction

In relational database systems, a *view* is a logical table derived from one or more physical (*base*) tables. Views are useful for presenting different levels of abstraction or different portions of a database to different users. Typically, a view is specified as an SQL **select** expression. A retrieval query over a view is written as if the view were a physical table; the query's answer is logically equivalent to evaluating the view's **select** expression, then performing the query using the result. There are two well-known approaches to implementing views. In the first approach, views are *virtual*: queries over views are modified into queries over base tables [Sto75]. In the second approach, views are *materialized*: they are computed from the base tables and stored in the database [BLT86,KP81,SI84]. Different applications favor one or the other approach. In this paper we consider the problem of view materialization.

Production rules in database systems allow specification of data manipulation operations that are executed automatically when certain events occur or conditions are met, e.g. [DE89, MD89, SJGP90, WF90]. Clearly, production rules can be used to maintain materialized views: when base tables change, rules are triggered that modify the view.[1] Writing a correct set of rules for efficiently maintaining a given view can be a difficult process, however. The rules could simply rematerialize the view from the base tables, but this can be very inefficient. Efficiency is achieved by *incremental* maintenance, in which the changed portions of the base tables are propagated to the view, without full recomputation. We have developed a method that automatically derives incremental maintenance rules for a wide class of views. The rules produced are executable using the rule language of the Starburst database system at the IBM Almaden Research Center [WCL91].

Figure 1 shows the structure of our system, which is invoked at compile-time when a view is created. Initially, the user enters the view as an SQL **select** expression, along with information about keys for the view's base tables.[2] Our system then performs syntactic analysis on the view definition; this analysis determines two things: (1) whether the view may contain duplicates (2) for each base table referenced in the view, whether efficient view maintenance rules are possible for operations on that table. The user is provided with the results of this analysis. The results may indicate that, in order to improve the efficiency of view maintenance, further interaction with the system is necessary prior to rule generation. In particular:

*Permanent address: Dip. di Elettronica, Politecnico di Milano, Piazza L. Da Vinci 32, 20133 Milano, Italy

[1] Production rules also can be used to implement virtual views, as shown in [SJGP90].

[2] Key information is essential for view analysis, as we will show. Functional dependencies could be specified as well, but we assume that keys are more easily understood and specified by the user; in normalized tables, functional dependencies are captured by keys anyway.

- Views with duplicates cannot be maintained efficiently, as explained in Section 4.3. Hence, if the system detects that the view may contain duplicates, then the user should add **distinct** to the view definition. (In SQL, **distinct** eliminates duplicates.)
- If the system detects that efficient maintenance rules are not possible for some base table operations, this may indicate to the user that not all key information has been included, or the user may choose to modify the view definition.

If changes are made, view analysis is repeated. In practice, we have discovered that efficient rules are possible for most views and operations once all key information is provided. However, there are cases when certain base table operations cannot be supported efficiently. If these operations are expected to occur frequently, view materialization may be inappropriate. The responsibility for considering these trade-offs lies with the user; our system provides all necessary information.

Once the user is satisfied with the view definition and its properties, the system generates the set of view-maintaining rules. Rules are produced for **insert**, **delete**, and **update** operations on each base table referenced in the view. The rule language we use is *set-oriented*, meaning that rules are triggered after arbitrary sets of changes to the database (Section 3). For those operations for which the system has determined that efficiency is possible, the maintenance rules modify the view incrementally according to the changes made to the base tables. These changes are accessible using the rule system's *transition table* mechanism (Section 3). For those operations for which efficiency is not possible, rematerialization is performed.

Note that the view must be computed in its entirety once, after which it is maintained automatically. The frequency of view maintenance depends on the frequency of rule invocation, which is flexible; see Section 3. Our method is directly applicable for simultaneous maintenance of multiple views; see Section 9.

1.1 Related Work

Most other work in incremental view maintenance differs from ours in two ways: (1) It takes an algebraic approach, considering a restricted class of views and operations. In contrast, we consider a practical class of views specified using a standard query language, and we consider arbitrary database operations. (2) It suggests view maintenance mechanisms that must be built into the database system. In contrast, we propose view maintenance as an application of an existing mechanism. In addition, our system provides interaction whereby the user can modify a view so the system will guarantee efficient maintenance.

In [BLT86], views are specified as relational algebra expressions. Algorithms are given for determining when base table changes are irrelevant to the view and for differentially reevaluating a view after a set of insert and delete operations. [Han87] extends this work to exploit common subexpressions and proposes an alternative approach using RETE networks; [Han87] also includes algorithms for incremental aggregate maintenance. In [RCBB89], an algebra of "delta relations" is described, including a "changes" operator that can be applied to views. There is a suggested connection to the production rules of HiPAC [MD89], but rule derivation is not included. In [SP89], incremental maintenance of single-table views is considered, with emphasis on issues of distribution.

Our work here is loosely related to that reported in [CW90], where we gave a method for deriving production rules that maintain integrity constraints. Our solutions to the two problems differ considerably, but the approaches are similar: In both cases we describe a general compile-time facility in which the user provides a high-level declarative specification, then the system uses syntactic analysis to produce a set of lower-level production rules with certain properties relative to the user's specification.

1.2 Outline

Section 2 defines our SQL-based syntax for view definition and Section 3 provides an overview of our production rule language. Section 4 motivates our approach: it gives an informal overview of view analysis, explains incremental maintenance, and describes certain difficulties encountered with duplicates and updates.[3] Subsequent sections contain the core technical material, formally describing our methods for view analysis and rule generation. We consider *top-level* table references in Section 5, *positively nested* subqueries in Section 6, *negatively nested* subqueries in Section 7, and *set operators* in Section 8. In each of these sections we describe how view analysis can guarantee certain properties, and we show how these properties are used to determine if efficient maintenance is possible. Section 9 addresses system execution, showing that the generated rules behave correctly at run-time. Finally, in Section 10 we conclude and discuss future work.

Due to space constraints, some details have been omitted. For further details and additional examples see [CW91].

2 View Definition Language

Views are defined using a subset of the SQL syntax for **select** expressions. The grammar is given in Figure 2 and should be self-explanatory to readers familiar with SQL [IBM88].[4] Several examples are given in subsequent sections. Our view definition language is quite powerful, but, for brevity and to make our approach more presentable, the language does include certain restrictions:

[3] Note that we are not dealing with the *view update problem*, which addresses how updates on views are propagated to updates on base tables. We are considering how updates on base tables are propagated to updates on views.

[4] We include multi-column **in** (grammar productions 12 and 16), which is not standard in all SQL implementations.

<table>
<tr><td>1.</td><td>View-Def</td><td>::=</td><td>define view V(Col-List):
View-Exp</td></tr>
<tr><td>2.</td><td>View-Exp</td><td>::=</td><td>Select-Exp | Set-Exp</td></tr>
<tr><td>3.</td><td>Select-Exp</td><td>::=</td><td>select [distinct] Col-List
from Table-List
[where Predicate]</td></tr>
<tr><td>4.</td><td>Set-Exp</td><td>::=</td><td>Select-Exp$_1$ union distinct
Select-Exp$_2$ union distinct
... Select-Exp$_n$</td></tr>
<tr><td>5.</td><td></td><td>|</td><td>Select-Exp$_1$ intersect
Select-Exp$_2$ intersect
... Select-Exp$_n$</td></tr>
<tr><td>6.</td><td>Col-List</td><td>::=</td><td>$Col_1, \ldots, Col_n \mid *$</td></tr>
<tr><td>7.</td><td>Col</td><td>::=</td><td>$[T.]C \mid [Var.]C$</td></tr>
<tr><td>8.</td><td>Table-List</td><td>::=</td><td>$T_1\ [Var_1], \ldots, T_n\ [Var_n]$</td></tr>
<tr><td>9.</td><td>Predicate</td><td>::=</td><td>Item Comp Item</td></tr>
<tr><td>10.</td><td></td><td>|</td><td>exists (Simple-Select)</td></tr>
<tr><td>11.</td><td></td><td>|</td><td>not exists (Simple-Select)</td></tr>
<tr><td>12.</td><td></td><td>|</td><td>Item in (Simple-Select)</td></tr>
<tr><td>13.</td><td></td><td>|</td><td>Item not in (Simple-Select)</td></tr>
<tr><td>14.</td><td></td><td>|</td><td>Item Comp any (Simple-Select)</td></tr>
<tr><td>15.</td><td></td><td>|</td><td>Predicate and Predicate</td></tr>
<tr><td>16.</td><td>Item</td><td>::=</td><td>Col | $\langle$Col-List$\rangle$ | constant</td></tr>
<tr><td>17.</td><td>Comp</td><td>::=</td><td>= | < | <= | > | >= | !=</td></tr>
<tr><td>18.</td><td>Simple-Select</td><td>::=</td><td>select Col-List from Table-List
[where Simple-Pred]</td></tr>
<tr><td>19.</td><td>Simple-Pred</td><td>::=</td><td>Item Comp Item</td></tr>
<tr><td>20.</td><td></td><td>|</td><td>Simple-Pred and Simple-Pred</td></tr>
</table>

Figure 2: *Grammar for View Definitions*

- Disjunction in predicates is omitted. (There is little loss of expressive power since **or** usually can be simulated using **union**.)
- Subqueries are limited to one level of nesting.
- Set operators **union** and **intersect** may not be mixed; set operator **minus** is omitted.
- Comparison operators using **all** are omitted.

The reader will see that our method could certainly be extended to eliminate these restrictions, but the details are lengthy. Note also that we have omitted aggregates. Incremental methods for maintaining aggregates have been presented elsewhere [Han87]; these techniques can be adapted for our framework.

3 Production Rule Language

We provide a brief but self-contained overview of the set-oriented, SQL-based production rule language used in the remainder of the paper. Further details and numerous examples appear in [WF90, WCL91]. Here we describe only the subset of the rule language used by the view maintenance rules.

Our rule facility is fully integrated into the Starburst database system. Hence, all the usual database functionality is available; in addition, a set of rules may be defined. Rules are based on the notion of *transitions*, which are database state changes resulting from execution of a sequence of data manipulation operations. We consider only the net effect of transitions, as in [BLT86, WF90]. The syntax for defining production rules is:[5]

```
create rule name
when transition predicate
then action
[ precedes rule-list ]
```

Transition predicates specify one or more operations on tables: **inserted into T**, **deleted from T**, or **updated T**. A rule is *triggered* by a given transition if at least one of the specified operations occurred in the net effect of the transition. The action part of a rule specifies an arbitrary sequence of SQL data manipulation operations to be executed when the rule is triggered. The optional **precedes** clause is used to induce a partial ordering on the set of defined rules. If a rule R_1 specifies R_2 in its **precedes** list, then R_1 is higher than R_2 in the ordering. When no ordering is specified between two rules, their order is arbitrary but deterministic [ACL91].

A rule's action may refer to the current state of the database through top-level or nested SQL **select** operations. In addition, rule actions may refer to *transition tables.* A transition table is a logical table reflecting changes that have occurred during a transition. At the end of a given transition, transition table "**inserted T**" refers to those tuples of table **T** in the current state that were inserted by the transition, transition table "**deleted T**" refers to those tuples of table **T** in the pre-transition state that were deleted by the transition, transition table "**old updated T**" refers to those tuples of table **T** in the pre-transition state that were updated by the transition, and transition table "**new updated T**" refers to the current values of the same tuples. Transition tables may be referenced in place of tables in the **from** clauses of **select** operations.

Rules are activated at *rule assertion points.* There is an assertion point at the end of each transaction, and there may be additional user-specified assertion points within a transaction.[6] We describe the semantics of rule execution at an arbitrary assertion point. The state change resulting from the user-generated database operations executed since the last assertion point (or start of the transaction) create the first relevant transition, and some set of rules are triggered by this transition. A triggered rule R is chosen from this set such that no other triggered rule is higher in the ordering. R's action is executed. After execution of R's action, all other rules are triggered only if their transition predicate holds with respect to the composite transition created by the initial transaction and subsequent execution of R's action. That is, these rules consider R's action as if it were executed as part of the initial transition. Rule R, however,

[5] Rules also may contain *conditions* in **if** clauses, but these are not needed for view maintenance.

[6] Currently, assertion points are at transaction commit only. We will soon extend the system with a flexible mechanism that supports additional points [WCL91].

has already "processed" the initial transition; thus, R is triggered again only if its transition predicate holds with respect to the transition created by its action. From the new set of triggered rules, a rule is chosen such that no other triggered rule is higher in the ordering, and its action is executed. At an arbitrary time in rule processing, a given rule is triggered if its transition predicate holds with respect to the (composite) transition since the last time at which its action was executed; if its action has not yet been executed, it is considered with respect to the transition since the last rule assertion point or start of the transaction. When the set of triggered rules is empty, rule processing terminates.

For view maintenance, it sometimes is necessary for a rule to consider the entire pre-transition value of a table (see, e.g., Section 5.4). Currently there is no direct mechanism in the rule language for obtaining this value, but it can be derived from transition tables. In the action part of view maintenance rules, we use "**old T**" to refer to the value of table **T** at the start of the transition triggering the rule. **old T** is translated to:

```
(T minus inserted T minus new updated T)
union deleted T union old updated T
```

This expression may seem rather complex, but one should observe that in most cases the transition tables are small or empty.

4 Motivation

4.1 View Analysis

Initially, the user defines a view using the language of Section 2, and the user specifies a set of (single- or multi-column) keys for the view's base tables. All known keys for each table should be specified, since this provides important information for view analysis. Using the key information, during view analysis the system considers each list of table references in the view definition. For each list, it first computes the "bound columns" of the table references. Based on the bound columns, it then determines for each table reference whether the reference is "safe". When a table reference is safe, incremental view maintenance rules can be generated for operations on that table, as described in Section 4.2. The system also uses the bound columns for the top-level tables to determine if the view may contain duplicates. Formal definitions for bound columns and safety are based on the context of table references and are given in Sections 5–7.

4.2 Incremental Maintenance

The definition of a view V can be interpreted as an expression mapping base tables to table V. That is, $V = V_{exp}(T_1, .., T_n)$, where $T_1, .., T_n$ are the base tables appearing in V's definition. Efficient maintenance of V is achieved when changes to $T_1, .., T_n$ can be propagated incrementally to V, without substantial recomputation. Consider any table reference T_i in V, and assume for the moment that T_i appears only once in V's definition. If view analysis determines that T_i is safe, then changes to T_i can be propagated incrementally to V. More formally, changes to T_i (sets of insertions, deletions, or updates), denoted ΔT_i, produce changes to V, denoted ΔV, that can be computed using only ΔT_i and the other base tables: $\Delta V = V'_{exp}(T_1, .., \Delta T_i, .., T_n)$, where V'_{exp} is an expression derived from V_{exp}. Table V is then modified by inserting or deleting tuples from ΔV as appropriate. We assume that ΔT_i is small with respect to T_i and ΔV is small with respect to V; hence, safe table references result in efficient maintenance rules. If T_i appears more than once in V's definition, we separately analyze each reference. If all references are safe, then changes to T_i can be propagated incrementally to V. If any reference is unsafe, changes to T_i may cause rematerialization.

4.3 Duplicates

Our method does not support efficient maintenance of views with duplicates. The main difficulty lies in generating rule actions in SQL that can manipulate exact numbers of duplicates. As an example, the SQL **delete** operation is based on truth of a predicate; hence, if a table contains four copies of a tuple (say), there is no SQL operation that can delete exactly two copies. To correctly maintain views with duplicates, such partial deletions can be necessary. [BLT86] also considers the problem of duplicates in views, proposing two solutions. In the first solution, an extra column is added in the view table to count the number of occurrences of each tuple. We choose not to use this approach because rule generation can become quite complex and the result is not transparent to the user. (The user must reference duplicates in the view through the extra column.) The second solution proposed in [BLT86] ensures that a view will not contain duplicates by requiring it to include key columns for each of the base tables. We have essentially taken this approach, however we have devised algorithms that allow us to loosen the key requirement considerably, yet still guarantee that a view will not contain duplicates.

4.4 Update Operations

When update operations are performed on a view's base tables, we would like to consequently perform an update operation on the view. In many cases, however, this is not the semantic effect. As a simple example, consider two tables `T1(A,B)` and `T2(C,D)` where `T1` contains tuples `(x,y)`, `(z,y)`, and `(u,v)`, and `T2` contains tuples `(x,z)` and `(v,x)`. Consider the following view:

```
define view V(A): select T1.A from T1, T2
                  where T1.B = T2.C
```

Initially, `V` contains only one tuple, `(u)`. Now suppose the following two update operations are performed on table `T2`:

```
update T2 set C = u where D = x ;
update T2 set C = y where D = z
```

The effect of the first update is to remove tuple `(u)` from view `V`, while the effect of the second update is to add tuples `(x)` and `(z)` to `V`. There is no way to reflect the update operations on base table `T2` as an update operation on view `V`; rather, the updates must be reflected as delete and insert operations on V. There do exist some cases in which update operations on base tables can be reflected as updates on views. However, for general and automatic rule derivation, in our approach update operations on base tables always result in delete and/or insert operations on the view.

5 Top-Level Table References

Assume now that the user has defined a view and has specified key information for the view's base tables. Assume that the view does not include set operators **union** or **intersect**; views with set operators are covered in Section 8. The system first analyzes the top-level table references, i.e., those references generated from the *Table-List* in grammar production 3 of Figure 2. This analysis reveals both whether the view may contain duplicates and whether efficient maintenance rules are possible for operations on the top-level tables. Consider a view V with the general form:[7]

define view V(*Col-List*):
  select $C_1, .., C_n$ **from** $T_1, .., T_m$ **where** P

where $T_1, .., T_m$ are the top-level table references, $C_1, .., C_n$ are columns of $T_1, .., T_m$, and P is a predicate.

5.1 Bound Columns

View analysis relies on the concept of *bound columns*. The bound columns of the top-level table references in view V are denoted $B(V)$ and are computed as follows:

Definition 5.1 (Bound Columns for Top-Level Table References)

1. Initialize $B(V)$ to contain the columns $C_1, .., C_n$ projected in the view definition.
2. Add to $B(V)$ all columns of $T_1, .., T_m$ such that predicate P includes an equality comparison between the column and a constant.
3. Repeat until $B(V)$ is unchanged:
   (a) Add to $B(V)$ all columns of $T_1, .., T_m$ such that predicate P includes an equality comparison between the column and a column in $B(V)$.
   (b) Add to $B(V)$ all columns of any table T_i, $1 \leq i \leq m$, if $B(V)$ includes a key for T_i. □

Bound columns can be computed using syntactic analysis and guarantee the following useful property (Lemma 5.2 below): If two tuples in the cross-product of top-level tables $T_1, .., T_m$ satisfy predicate P and differ in their bound columns, then the tuples also must differ in view columns $C_1, .., C_m$. Let $Proj(t, C_1, .., C_j)$ denote the projection of a tuple t onto a set of columns $C_1, .., C_j$.

Lemma 5.2 (Bound Columns Lemma for Top-Level Tables) Let t_1 and t_2 be tuples in the cross-product of $T_1, .., T_m$ such that t_1 and t_2 both satisfy P. By definition, columns $C_1,..,C_n$ are in $B(V)$. If $D_1, .., D_k$ are additional columns in $B(V)$ such that t_1 and t_2 are guaranteed to differ in $C_1, .., C_n, D_1, .., D_k$, i.e. $Proj(t_1, C_1, .., C_n, D_1, .., D_k) \neq Proj(t_2, C_1, .., C_n, D_1, .., D_k)$, then t_1 and t_2 also are guaranteed to differ in $C_1, .., C_n$, i.e. $Proj(t_1, C_1, .., C_n) \neq Proj(t_2, C_1, .., C_n)$.

Proof: Suppose, for the sake of a contradiction, that $Proj(t_1, C_1, .., C_n) = Proj(t_2, C_1, .., C_n)$. Then there must be some D_i in $D_1, .., D_k$ such that $Proj(t_1, D_i) \neq Proj(t_2, D_i)$. We show that this is impossible. Consider any column D_i in $D_1, .., D_k$. Since D_i is in $B(V)$, by the recursive definition of $B(V)$ and since t_1 and t_2 both satisfy predicate P, the value of column D_i in both t_1 and t_2 must either

1. satisfy an equality with a constant k, or
2. satisfy an equality with a column C_j in $C_1, .., C_n$, or
3. be functionally dependent on a constant k or column C_j. (This is the case where D_i was added to $B(V)$ because a key for D_i's table was present; recall that all columns of a table are functionally dependent on any key for that table.)

In the case of a constant, $Proj(t_1, D_i)$ and $Proj(t_2, D_i)$ are both equal to or functionally dependent on the same constant, so $Proj(t_1, D_i) = Proj(t_2, D_i)$. In the case of a column C_j, $Proj(t_1, C_j) = Proj(t_2, C_j)$ by our supposition, so $Proj(t_1, D_i) = Proj(t_2, D_i)$. □

5.2 Duplicate Analysis

If V's definition does not include **distinct**, then our system performs duplicate analysis. If this analysis reveals that V may contain duplicates, then the user is notified that maintenance rules cannot be generated for V unless V's definition is modified to include **distinct**. (The system does not add **distinct** automatically since it may change the view's semantics.) Once the bound columns for top-level table references have been computed, duplicate analysis is straightforward:

Theorem 5.3 (Duplicates) If $B(V)$ includes a key for every top-level table, then V will not contain duplicates.

Proof: Let t_1 and t_2 be two different tuples in the cross-product of the top-level tables in V such that t_1 and t_2 both satisfy predicate P. We must show that t_1 and t_2 cannot produce duplicate tuples in V, i.e. $Proj(t_1, C_1, .., C_n) \neq Proj(t_2, C_1, .., C_n)$. By the theorem's assumption, there must be additional columns $D_1, .., D_k$ in $B(V)$ such that $C_1, .., C_n, D_1, .., D_k$ include a key for every top-level table. Then t_1 and t_2 must differ in $C_1, .., C_n, D_1, .., D_k$. Consequently, by Lemma 5.2, $Proj(t_1, C_1, .., C_n) \neq Proj(t_2, C_1, .., C_n)$. □

5.3 Safety Analysis

Safety of top-level table references is similar to duplicate analysis:

Definition 5.4 (Safety of Top-Level Table References) Top-level table reference T_i is *safe* in V if $B(V)$ includes a key for T_i. □

The following three theorems show that if table reference T_i is safe, then **insert**, **delete**, and **update** operations on T_i can be reflected by incremental changes to V.

[7] For clarity and without loss of generality, we omit the use of table variables here.

Theorem 5.5 (Insertion Theorem for Top-Level Tables) Let T_i be a safe top-level table reference in V and suppose a tuple t is inserted into T_i. If v is a tuple in the cross-product of the top-level tables using tuple t from T_i, and v satisfies predicate P so that $Proj(v, C_1, .., C_n)$ is in view V after the insertion, then $Proj(v, C_1, .., C_n)$ was not in V before the insertion.

Proof: Suppose, for the sake of a contradiction, that there was a tuple v' in V before the insertion such that $Proj(v', C_1, .., C_n) = Proj(v, C_1, .., C_n)$. Let $D_1, .., D_k$ be additional bound columns so that $C_1, .., C_n, D_1, .., D_k$ includes a key for T_i. (We know such columns exist since T_i is safe.) Since v and v' include different tuples from T_i, then $Proj(v, C_1, .., C_n, D_1, .., D_k) \neq Proj(v', C_1, .., C_n, D_1, .., D_k)$. Hence, by Lemma 5.2, $Proj(v, C_1, .., C_n) \neq Proj(v', C_1, .., C_n)$. □

The practical consequence of this theorem is that if a set of tuples ΔT_i are inserted into T_i, then the tuples ΔV that should be inserted into V can be derived from the cross-product of the top-level tables using ΔT_i instead of T_i. This exactly corresponds to the definition of incremental maintenance in Section 4.2, and is implemented in the rules given below.

Similar theorems with similar consequences apply for delete and update operations. The proofs are omitted since they also are similar [CW91].

Theorem 5.6 (Deletion Theorem for Top-Level Tables) Let T_i be a safe top-level table reference in V and suppose a tuple t is deleted from T_i. If v is a tuple in the cross-product of the top-level tables using tuple t from T_i, and v satisfies predicate P so that $Proj(v, C_1, .., C_n)$ was in view V before the deletion, then $Proj(v, C_1, .., C_n)$ is not in V after the deletion. □

Theorem 5.7 (Update Theorem for Top-Level Tables) Let T_i be a safe top-level table reference in V and suppose a tuple t is updated in T_i. Let v_O be a tuple in the cross-product of the top-level tables using the old value of tuple t from T_i, where v_O satisfies P so that $Proj(v_O, C_1, .., C_n)$ was in view V before the update. Let v_N be a tuple in the cross-product of the top-level tables using the new value of tuple t from T_i, where v_N satisfies P so that $Proj(v_N, C_1, .., C_n)$ is in V after the update. Finally, let v be a tuple in the cross-product of the top-level tables not using t, where v satisfies P so v is in V both before and after the update. Then $Proj(v_O, C_1, .., C_n) \neq Proj(v, C_1, .., C_n)$ and $Proj(v_N, C_1, .., C_n) \neq Proj(v, C_1, .., C_n)$. □

5.4 Rule Generation

We describe how maintenance rules are generated for the top-level tables. We first consider safe table references, then unsafe references. Initially, for each table reference we generate four rules—one triggered by **inserted**, one by **deleted**, and two by **updated**. Subsequently we explain how some rules can be combined and how the entire rule set is ordered.

Let T_i be a safe top-level table reference in view V defined as above. If tuples are inserted into T_i, then we want to insert into V those tuples produced by the view definition using **inserted Ti** instead of **Ti** in the top-level table list. By Theorem 5.5, these insertions cannot create duplicates in the view. However, if a similar rule is applied because tuples also were inserted into a different top-level table, then duplicates could appear. Hence, before inserting a new tuple, the rule must ensure that the tuple has not already been inserted by a different rule. This is checked efficiently using transition table **inserted V**. The rule for **inserted** is:

```
create rule ins-Ti-V
when inserted into Ti
then insert into V
       (select C1,..,Cn
        from T1,..,inserted Ti,..,Tm
        where P and <C1,..,Cn> not in inserted V)
```

If tuples are deleted from T_i, then we want to delete from V those tuples produced by the view definition using **deleted Ti** instead of **Ti** in the top-level table list. By Theorem 5.6, we know that these tuples should no longer be in the view. Again, however, we must remember that other tables in the top-level table list may have been modified. Hence, to identify the correct tuples to delete from V, we must consider the pre-transition value of all other tables, obtained using the **old** feature described in Section 3. For predicate P, let *P-old* denote P with all table references **T** replaced by **old T**. The rule for **deleted** is:

```
create rule del-Ti-V
when deleted from Ti
then delete from V
     where <C1,..,Cn> in
       (select C1,..,Cn
        from old T1,..,deleted Ti,..,old Tm
        where P-old)
```

As explained in Section 4.4, update operations on base tables always cause delete and/or insert operations on views. In fact, we generate two separate rules triggered by **updated**—one to perform deletions and the other to perform insertions. They are similar to the rules for **deleted** and **inserted**, and their correctness follows from Theorem 5.7:

```
create rule old-upd-Ti-V
when updated Ti
then delete from V
     where <C1,..,Cn> in
       (select C1,..,Cn
        from old T1,..,old updated Ti,..,old Tm
        where P-old)

create rule new-upd-Ti-V
when updated Ti
then insert into V
       (select C1,..,Cn
        from T1,..,new updated Ti,..,Tm
        where P and <C1,..,Cn> not in inserted V)
```

If a table appears more than once in the top-level table list, then rules are generated for each reference. Rules with identical triggering operations whose actions perform the same operation (either insert or delete) are merged into one rule by sequencing or combining their actions. Once the entire set of rules is generated (including those for nested table references, described below), they are ordered by adding **precedes** clauses so that all rules performing deletions precede all rules performing insertions.[8]

Now consider the case when a top-level table reference T_i is unsafe, so the properties guaranteed by the theorems may not hold. For insertions, incremental maintenance is still possible; the only difference from the safe case is that all new tuples must be checked against V itself to guarantee that duplicates are not produced. If V is indexed, this can be performed efficiently.

```
create rule ins-Ti-V
when inserted into Ti
then insert into V
       (select C1,..,Cn
        from T1,..,inserted Ti,..,Tm
        where P and <C1,..,Cn> not in V)
```

Delete and update operations are more difficult, and this is where recomputation must occur. If a tuple is deleted from T_i, without Theorem 5.6 we cannot determine whether corresponding tuples should be deleted from V—those tuples still may be produced by other base table tuples that have not been deleted; a similar problem occurs with update. The only solution is to reevaluate the view expression itself. Since this is equivalent to rematerializing the view, we choose to create a single distinguished rule that performs rematerialization. This rule will be triggered by all operations for which efficient maintenance is impossible. (As mentioned above, if these operations are expected to occur frequently, then materialization may be inappropriate for this view.) The rematerialization rule with triggering operations for T_i is:

```
create rule rematerialize-V
when deleted from Ti,
     updated Ti
then delete from V;
     insert into V
       (select C1,..,Cn from T1,..,Tm where P);
     deactivate-rules(V)
```

This rule will have precedence over all other rules for V. Since execution of the first two rule actions entirely rematerializes V, the rule's final action, **deactivate-rules(V)**, deactivates all other rules for V until the next rule assertion point.[9] Note that when a triggering operation appears in the rematerialization rule, any other rules triggered by that operation can be eliminated.

5.5 Examples

We draw examples from a simple airline reservations database with the following schema:

```
flight (FLIGHT-ID, flight-no, date)
res (RES-ID, psgr-id, flight-id, seat)
psgr (PSGR-ID, name, phone, meal, ffn)
ff (FFN, miles)
```

Most of the schema is self-explanatory, with `res` denoting reservation, `ff` denoting frequent flier, and `ffn` denoting frequent flier number. Primary keys for each table are capitalized; other keys are `<flight-no,date>` for table `flight`, `<psgr-id,flight-id>` or `<flight-id, seat>` for table `res`, and `ffn` for table `psgr`.

Consider the following view, which provides the seat numbers and meal preferences of all passengers on a given flight (`FID`) who have ordered special meals:

```
define view special-meals(seat, meal):
  select res.seat, psgr.meal
  from res, psgr
  where res.flight-id = FID
  and res.psgr-id = psgr.psgr-id
  and psgr.meal != null
```

Using Definition 5.1, we determine that the bound columns of top-level table references `res` and `psgr` are: projected columns `res.seat` and `psgr.meal`, column `res.flight-id` since it is equated to a constant in the predicate, all remaining columns of `res` since `<flight-id,seat>` is a key, and `psgr.psgr-id` since it is equated to bound column `res.psgr-id`. Since the bound columns include keys for both top-level tables, the view will not contain duplicates, and incremental maintenance rules can be generated for both tables. The rules triggered by operations on table `res` are given here; the rules for table `psgr` are similar:

```
create rule ins-res-special-meals
when inserted into res
then insert into special-meals
       (select res.seat, psgr.meal
        from inserted res, psgr
        where res.flight-id = FID
        and res.psgr-id = psgr.psgr-id
        and psgr.meal != null
        and <seat,meal> not in
              inserted special-meals)

create rule del-res-special-meals
when deleted from res
then delete from special-meals
     where <seat,meal> in
       (select res.seat, psgr.meal
        from deleted res, old psgr
        where res.flight-id = FID
        and res.psgr-id = psgr.psgr-id
        and psgr.meal != null)

create rule old-upd-res-special-meals
when updated res
then delete from special-meals
where <seat,meal> in
        (select res.seat, psgr.meal
         from old updated res, old psgr
         where res.flight-id = FID
         and res.psgr-id = psgr.psgr-id
         and psgr.meal != null)
```

[8] This is why we merge only rules with the same action operation and why we create two separate rules for **updated**—for ordering, we cannot generate rule actions that perform both deletions and insertions.

[9] This feature is not included in the current rule system but can easily be simulated using rule conditions; see [Wid91]. We intend to add this feature in the near future.

```
create rule new-upd-res-special-meals
when updated res
then insert into special-meals
       (select res.seat, psgr.meal
        from new updated res, psgr
        where res.flight-id = FID
        and res.psgr-id = psgr.psgr-id
        and psgr.meal != null
        and <seat,meal> not in
              inserted special-meals)
```

As a second example, consider the following view, which provides the frequent flier numbers of all passengers currently holding reservations:

```
define view ff-res(ffn):
  select psgr.ffn
  from psgr, res
  where psgr.psgr-id = res.psgr-id
```

The bound columns are all columns of table **psgr** (since **ffn** is a key) and column **res.psgr-id**. Since the bound columns do not include a key for table **res**, the view may contain duplicates, and **distinct** must be added. Table reference **psgr** is safe, so the rules for operations on **psgr** are similar to those in the previous example. Table reference **res** is unsafe, however, so the following rules are generated:

```
create rule ins-res-ff-res
when inserted into res
then insert into ff-res
       (select distinct psgr.ffn
        from psgr, inserted res
        where psgr.psgr-id = res.psgr-id
        and ffn not in ff-res)

create rule rematerialize-ff-res
when deleted from res,
     updated res
then delete from ff-res;
     insert into ff-res
       (select distinct psgr.ffn from psgr, res
        where psgr.psgr-id = res.psgr-id);
     deactivate-rules(ff-res)
```

6 Positively Nested Subqueries

A *positively nested* subquery is a nested **select** expression preceded by **exists**, **in**, or *Comp* **any**, where *Comp* is any comparison operator except !=. We first describe safety analysis and rule generation for table references in **exists** subqueries. Similar methods apply for the other positively nested subqueries and are explained in Section 6.3. Consider a view V as follows, where $N_1, .., N_l$ are the table references under consideration:

> **define view** V(*Col-List*):
> **select** $C_1, .., C_n$ **from** $T_1, .., T_m$
> **where** P' **and exists**
> (**select** *Cols* **from** $N_1, .., N_l$ **where** P)

6.1 Bound Columns and Safety Analysis

To analyze nested table references we introduce the concept of columns that are *bound by correlation* to the bound columns of the top-level tables. We assume that set $B(V)$ of top-level bound columns already has been computed. Correlated bound columns are denoted $C(V)$, and for **exists** they are computed as follows:

Definition 6.1 (Correlated Bound Columns for Exists)

1. Initialize $C(V)$ to contain all columns of $N_1, .., N_l$ such that predicate P includes an equality comparison between the column and a column in $B(V)$.
2. Add to $C(V)$ all columns of $N_1, .., N_l$ such that predicate P includes an equality comparison between the column and a constant.
3. Repeat until $C(V)$ is unchanged:
   (a) Add to $C(V)$ all columns of $N_1, .., N_l$ such that predicate P includes an equality comparison between the column and a column in $C(V)$.
   (b) Add to $C(V)$ all columns of any table N_i, $1 \leq i \leq l$, if $C(V)$ includes a key for N_i. □

Correlated bound columns for **exists** guarantee the following property:

Lemma 6.2 (Bound Columns Lemma for Exists) Consider four tuples, t_1 and t_2 in the cross-product of $T_1, .., T_m$ and n_1 and n_2 in the cross-product of $N_1, .., N_l$, such that t_1 and t_2 satisfy predicate P', n_1 satisfies nested predicate P using t_1 for the top-level cross-product, and n_2 satisfies P using t_2 for the top-level cross-product. Let $D_1, .., D_k$ be columns of $N_1, .., N_l$ in $C(V)$ such that n_1 and n_2 are guaranteed to differ in $D_1, .., D_k$, i.e. $Proj(n_1, D_1, .., D_k) \neq Proj(n_2, D_1, .., D_k)$. Then t_1 and t_2 are guaranteed to differ in $C_1, .., C_n$, i.e. $Proj(t_1, C_1, .., C_n) \neq Proj(t_2, C_1, .., C_n)$.

Proof: Suppose, for the sake of a contradiction, that $Proj(t_1, C_1, .., C_n) = Proj(t_2, C_1, .., C_n)$. By supposition there is some D_i in $D_1, .., D_k$ such that $Proj(n_1, D_i) \neq Proj(n_2, D_i)$. D_i is in $C(V)$, so by the recursive definitions of $C(V)$ and $B(V)$, since t_1 and t_2 satisfy P', and since n_1 with t_1 and n_2 with t_2 both satisfy predicate P, the value of column D_i in both n_1 and n_2 must either

1. satisfy an equality with a constant k, or
2. satisfy an equality with a column C_j in $C_1, .., C_n$, or
3. be functionally dependent on a constant k or column C_j.

As in Bound Columns Lemma 5.2, in all cases $Proj(n_1, D_i) = Proj(n_2, D_i)$. □

Safety analysis and rule generation for positively nested subqueries is similar to top-level tables:

Definition 6.3 (Safety of Table References for Exists) Table reference N_i in an **exists** subquery is *safe* in V if $C(V)$ includes a key for N_i. □

The following three theorems show that if N_i is safe, then insert, delete, and update operations on N_i can be reflected by incremental changes to V. We include a proof for the insertion theorem only; the other proofs follow by analogy.

Theorem 6.4 (Insertion Theorem for Exists) Let N_i be a safe table reference in an **exists** subquery in V and suppose a tuple n_i is inserted into N_i. Let v be a tuple in the cross-product of the top-level tables such that v satisfies P' and there is a tuple n in the cross-product of the nested tables using n_i such that n satisfies P using v, so $Proj(v, C_1, .., C_n)$ is in view V after the insertion. Then $Proj(v, C_1, .., C_n)$ was not in V before the insertion.

Proof: Suppose, for the sake of a contradiction, that $Proj(v, C_1, .., C_n)$ was in V before the insertion. Then there must have been a tuple n' in the cross-product of the nested tables before the insertion and a tuple v' in the top-level cross-product such that $Proj(v', C_1, .., C_n) = Proj(v, C_1, .., C_n)$, v' satisfies P', and n' satisfies P using v'. Let $D_1, .., D_k$ be correlated bound columns of $N_1, .., N_l$ such that $D_1, .., D_k$ includes a key for N_i. Since n and n' use different tuples from N_i, $Proj(n, D_1, .., D_k) \neq Proj(n', D_1, .., C_k)$. Then, by Lemma 6.2, $Proj(v', C_1, .., C_n) \neq Proj(v, C_1, .., C_n)$. □

Theorem 6.5 (Deletion Theorem for Exists) Let N_i be a safe table reference in an **exists** subquery in V and suppose a tuple n_i is deleted from N_i. Let v be a tuple in the cross-product of the top-level tables such that v satisfies P' and there is a tuple n in the cross-product of the nested tables using n_i such that n satisfies P using v, so $Proj(v, C_1, .., C_n)$ was in view V before the deletion. Then $Proj(v, C_1, .., C_n)$ is not in V after the deletion. □

Theorem 6.6 (Update Theorem for Exists) Let N_i be a safe table reference in an **exists** subquery in V and suppose a tuple n_i is updated in N_i. Let v_O be a tuple in the cross-product of the top-level tables such that v_O satisfies P' and there is a tuple n_O in the cross-product of the nested tables using the old value of n_i such that n_O satisfies P using v_O, so $Proj(v_O, C_1, .., C_n)$ was in view V before the update. Let v_N be a tuple in the cross-product of the top-level tables such that v_N satisfies P' and there is a tuple n_N in the cross-product of the nested tables using the new value of n_i such that n_N satisfies P using v_N, so $Proj(v_N, C_1, .., C_n)$ is in V after the update. If $Proj(v_O, C_1, .., C_n) \neq Proj(v_N, C_1, .., C_n)$, then $Proj(v_O, C_1, .., C_n)$ is not in V after the update and $Proj(v_N, C_1, .., C_n)$ was not in V before the update. □

6.2 Rule Generation

First consider safe table references. The properties guaranteed by Theorems 6.4–6.6 allow incremental maintenance to be performed just as for safe top-level table references: N_i is replaced by **inserted Ni** in the **inserted** rule, by **deleted Ni** in the **deleted** rule, and by **old updated Ni** and **new updated Ni** in the two **updated** rules. In the rules that perform insertions, we must check that tuples have not already been inserted by another rule; in the rules that perform deletions we must use the **old** value of other tables. If a table appears more than once in $N_1, .., N_l$, or if a table in $N_1, .., N_l$ also appears elsewhere in the view definition, then rules are merged as previously described. Unsafe table references also are handled similarly to top-level tables: If nested table reference N_i is unsafe, triggering operations **deleted from Ni** and **updated Ni** are included in the distinguished rematerialization rule for V. The **inserted** rule is similar to the safe rule, except "`not in V`" is added to the predicate rather than "`not in inserted V`".

6.3 Other Positively Nested Subqueries

Safety analysis and rule generation for subqueries preceded by **< any**, **<= any**, **> any**, and **>= any** is identical to **exists**. The method for **= any** and **in** (which are equivalent) also is identical to **exists**, except the set of correlated bound columns may be larger. Consider a view V of the form:

> **define view** V(*Col-List*):
>   **select** $C_1, .., C_n$ **from** $T_1, .., T_m$
>   **where** P' **and** $\langle D_1, .., D_j \rangle$ **in**
>   (**select** $E_1, .., E_j$ **from** $N_1, .., N_l$ **where** P)

Definition 6.1 of correlated bound columns is modified to include the case:

- Add to $C(V)$ every column E_i such that corresponding column D_i is in $B(V)$, $1 \leq i \leq j$.

The reader may note that view V above is equivalent to view V':

> **define view** V'(*Col-List*):
>   **select** $C_1, .., C_n$ **from** $T_1, .., T_m$
>   **where** P' **and exists**
>   (**select** $*$ **from** $N_1, .., N_l$ **where** P
>   **and** $D_1 = E_1$ **and** ... **and** $D_j = E_j$)

As expected, the correlated bound columns of view V' using Definition 6.1 for **exists** are equivalent to the correlated bound columns of V using the extended definition for **in**.[10]

6.4 Example

Using the airline reservations database introduced in Section 5.5, the following view provides the ID's of all passengers with more than 50,000 frequent flier miles:

```
define view many-miles(id):
  select psgr-id from psgr
  where psgr.ffn in
    (select ffn from ff where miles > 50,000)
```

All columns of top-level table `psgr` are bound since `psgr-id` is a key. Using our extended definition for **in**, `ff.ffn` is a correlated bound column. Since `ffn` is a key, nested table reference `ff` is safe. The **inserted** and **deleted** rules for table `ff` follow; the **updated** rules are similar.

[10] The reader may also note that **select** expressions with positive subqueries often can be transformed into equivalent **select** expressions without subqueries, as in [CG85,Kim82]. By considering the actual transformations, we see that the maintenance rules produced for any transformed view are equivalent to the maintenance rules produced for the original view.

```
create rule ins-ff-many-miles
when inserted into ff
then insert into many-miles
       (select psgr-id from psgr
        where psgr.ffn in
          (select ffn from inserted ff
           where miles > 50,000)
        and psgr-id not in inserted many-miles)

create rule del-ff-many-miles
when deleted from ff
then delete from many-miles
     where psgr-id in
           (select psgr-id from old psgr
            where psgr.ffn in
              (select ffn from deleted ff
               where miles > 50,000))
```

7 Negatively Nested Subqueries

A *negatively nested* subquery is a nested **select** expression preceded by **not exists**, **not in**, or **!=any**. We describe safety analysis and rule generation for table references in **not exists** subqueries. Similar methods apply for the other negatively nested subqueries; see [CW91]. Consider a view V of the form:

> **define view** V(*Col-List*):
> **select** $C_1,..,C_n$ **from** $T_1,..,T_m$
> **where** P' **and not exists**
> (**select** *Cols* **from** $N_1,..,N_l$ **where** P)

With negatively nested subqueries, insert operations on nested tables result in delete operations on the view, while delete operations on nested tables result in insert operations on the view.

7.1 Safety Analysis

For a negatively nested table reference N_i, we define two notions of safety: *I-safety* indicates that insert operations on N_i can be reflected by incremental changes to V, and *DU-safety* indicates that delete and update operations on N_i can be reflected by incremental changes to V. The definition of I-safety is somewhat different from previous safety definitions—correlated bound columns are not used, and all nested table references are considered together. Assume that set $B(V)$ of top-level bound columns already has been computed.

Definition 7.1 (I-Safety of Table References for Not Exists) Table references $N_1,..,N_l$ in a **not exists** subquery are *I-safe* in V if predicate P refers only to columns of N_i, $1 \le i \le l$, columns in $B(V)$, and constants. □

Using this notion of safety, we prove the following theorem for insertions:

Theorem 7.2 (Insertion Theorem for Not Exists) Let N_i be an I-safe table reference in a **not exists** subquery in V and suppose a tuple n_i is inserted into N_i. Let v be a tuple in the cross-product of the top-level tables such that v satisfies top-level predicate P' and there is a tuple n in the cross-product of the nested tables using n_i such that n satisfies nested predicate P using v. Then $Proj(v, C_1,..,C_n)$ is not in V after the insertion.

Proof: Suppose, for the sake of a contradiction, that $Proj(v, C_1,..,C_n)$ is in V after the insertion. Then there must be a tuple v' other than v in the cross-product of the top-level tables such that $Proj(v', C_1,..,C_n) = Proj(v, C_1,..,C_n)$, v' satisfies P', and there is no tuple n' in the cross-product of the nested tables such that n' satisfies P using v'. We show that there is such an n', namely n. By Definition 5.1 of $B(V)$, since v and v' both satisfy P' and $Proj(v', C_1,..,C_n) = Proj(v, C_1,..,C_n)$, v and v' are equivalent in all columns of $B(V)$. Since N_i is I-safe and since n satisfies P using v, by Definition 7.1 of safety, n also satisfies P using v'. □

For deletes and updates, we combine our new notion of I-safety with the previous notion of safety using keys. Correlated bound columns for negatively nested table references are defined as for positive references (Definition 6.1), and Bound Columns Lemma 6.2 still holds.

Definition 7.3 (DU-Safety of Table References for Not Exists) Table reference N_i in a **not exists** subquery is *DU-safe* in V if it is I-safe and $C(V)$ includes a key for N_i. □

Theorem 7.4 (Deletion Theorem for Not Exists) Let N_i be a DU-safe table reference in a **not exists** subquery in V and suppose a tuple n_i is deleted from N_i. Let v be a tuple in the cross-product of the top-level tables such that v satisfies P' and there is a tuple n in the cross-product of the nested tables using n_i such that n satisfies P using v. Then: (1) $Proj(v, C_1,..,C_n)$ was not in V before the deletion. (2) $Proj(v, C_1,..,C_n)$ is in V after the deletion.

Proof: The proof of (1) is analogous to the proof of Insertion Theorem 7.2. For (2), suppose, for the sake of a contradiction, that $Proj(v, C_1,..,C_n)$ is not in V after the deletion. Then there must be a tuple n' in the cross-product of the nested tables such that n' satisfies P using v. Let $D_1,..,D_k$ be correlated bound columns of $N_1,..,N_l$ such that $D_1,..,D_k$ includes a key for N_i. Since n and n' use different tuples from N_i, $Proj(n, D_1,..,D_k) \neq Proj(n', D_1,..,C_k)$. Then, by Lemma 6.2, $Proj(v, C_1,..,C_n) \neq Proj(v, C_1,..,C_n)$, which is impossible. □

Theorem 7.5 (Update Theorem for Not Exists) Let N_i be a DU-safe table reference in a **not exists** subquery in V and suppose a tuple n_i is updated in N_i. Let v_O be a tuple in the cross-product of the top-level tables such that v_O satisfies P' and there is a tuple n_O in the cross-product of the nested tables using the old value of n_i such that n_O satisfies P using v. Let v_N be a tuple in the cross-product of the top-level tables such that v_N satisfies P' and there is a tuple n_N in the cross-product of the nested tables using the new value of n_i such that n_N satisfies P using v. If $Proj(v_O, C_1,..,C_n) \neq Proj(v_N, C_1,..,C_n)$ then: (1) $Proj(v_N, C_1,..,C_n)$ is not in V after the update. (2) $Proj(v_O, C_1,..,C_n)$ was not in V before the update. (3) $Proj(v_O, C_1,..,C_n)$ is in V after the update.

Proof: Analogous to Theorems 7.2 and 7.4. □

7.2 Rule Generation

If nested table reference N_i is I-safe, then, using Theorem 7.2, the following incremental rule is generated:

```
create rule ins-Ni-V
when inserted into Ni
then delete from V
     where <C1,..,Cn> in
        (select C1,..,Cn from T1,..,Tm
         where P' and exists
           (select Cols
            from N1,..,inserted Ni,..,Nl
            where P))
```

Notice that the subquery's "`not exists`" is converted to "`exists`"; this conversion occurs in the **deleted** and **updated** rules as well. If N_i is not I-safe, then the view expression would need to be reevaluated to determine which tuples should be deleted. Hence in the unsafe case, **inserted into Ni** is included in the rematerialization rule for V.

If N_i is DU-safe, then, using Theorems 7.4 and 7.5, the following incremental rule for **deleted** is generated. The rules for **updated** correspond to the **inserted** and **deleted** rules as previously.

```
create rule del-Ni-V
when deleted from Ni
then insert into V
        (select C1,..,Cn from T1,..,Tm
         where P' and exists
           (select Cols
            from old N1,..,deleted Ni,..,old Nl
            where P)
         and <C1,..Cn> not in inserted V)
```

If table reference N_i is not DU-safe, **updated Ti** is included in the rematerialization rule for V. For **deleted**, however, incremental maintenance still can be performed—as previously, for the unsafe case the rule above is modified to use "`not in V`" rather than "`not in inserted V`".

7.3 Example

Using the airline reservations database introduced in Section 5.5, the following view provides the ID's of all reservations whose `flight-id` is not in table `flight`:

```
define view bad-flight(res-id):
  select res-id from res
  where not exists
    (select * from flight
     where flight.flight-id = res.flight-id)
```

By Definitions 7.1 and 7.3, nested table reference `flight` is both I-safe and DU-safe. The **inserted** and **deleted** rules for table `flight` follow; the **updated** rules are similar.

```
create rule ins-flight-bad-flight
when inserted into flight
then delete from bad-flight
     where res-id in
        (select res-id from res
         where exists
           (select * from inserted flight
            where flight.flight-id =
                  res.flight-id))

create rule del-flight-bad-flight
when deleted from flight
then insert into bad-flight
        (select res-id from res
         where exists
           (select * from deleted flight
            where flight.flight-id =
                  res.flight-id)
         and res-id not in inserted bad-flight)
```

8 Set Operators

Finally, consider views with *set operators*. A view definition may include either **union distinct** or **intersect**. For these views, view analysis and rule generation initially is performed independently on each component **select** expression. The rules are then modified to incorporate the set operators.

8.1 Union Views

Consider a view V of the form:

define view V(*Col-List*):
  select $Cols_1$ **from** $Tables_1$ **where** P_1
  union distinct ...
  union distinct **select** $Cols_k$ **from** $Tables_k$ **where** P_k

First, duplicate analysis is performed on each **select** expression as in Section 5.2; if any **select** expression may contain duplicates, the user is required to add **distinct** to that **select** expression. For each **select** expression, an initial set of view-maintaining rules is generated using the methods of the preceding sections. The rules' actions are then modified to incorporate **union**. In actions that perform **insert** operations, if "`not in inserted V`" has been added to predicate P_i due to a safe table reference, it is changed to "`not in V`"; this ensures that duplicates are not added by different **select** expressions. If the rule already includes "`not in V`" due to an unsafe table reference, it remains unchanged. Modifications for **delete** operations are more complicated. If a tuple no longer is produced by one of the **select** expressions, it should be deleted from V only if it is not produced by any of the other **select** expressions. Without loss of generality, consider a **delete** operation in the action of a rule generated from the first **select** expression in V. The following conjunct must be added to the **delete** operation's **where** clause:

```
and <Cols> not in
      (select Cols2 from Tables2 where P2)
and ...
and <Cols> not in
      (select Colsk from Tablesk where Pk)
```

Clearly, such conjuncts may cause considerable recomputation, depending on the complexity of the **select** expressions. For rules in which the recomputation cost appears large, the user may choose to move the triggering operation to the rematerialization rule for V.

As usual, rules with common triggering and action operations are merged, and rules whose triggering operations also appear in the rematerialization rule are eliminated.

8.2 Intersect Views

A view V with **intersect** operators is handled similarly to views with **union** operators. In rule modification, however, all rules performing **delete** operations remain unchanged. (If a tuple is deleted from any **select** expression, then it always should be deleted from V.) Modifications for **insert** operations are similar to the modifications for **delete** operations in **union** views: If a tuple is newly produced by one of the **select** expressions, it should be inserted into V only if it also is produced by all the other **select** expressions. Consider an **insert** operation in the action of a rule generated from the first **select** expression in V. The following conjunct must be added to the **where** clause of the **insert** operation's **select** expression:

```
and <Cols> in
      (select Cols2 from Tables2 where P2)
and ...
and <Cols> in
      (select Colsk from Tablesk where Pk)
```

Again, if the **select** expressions are sufficiently complex, the user may decide that rematerialization is more appropriate.

9 System Execution

So far, we have described only the compile-time aspects of our facility. View definition, view analysis, and rule generation all occur prior to database system execution. We still must ensure that, at run-time, derived rules will behave as desired, i.e., views will be maintained correctly. Suppose our facility has been used to derive sets of maintenance rules for several views. The system orders the set of rules for each view so that all **delete** operations in rule actions precede all **insert** operations. No ordering is necessary between rules for different views—the action part of each rule modifies only the view itself, so rules for different views have no effect on each other.

Consider the set of rules for a given view V, and suppose an arbitrary set of changes has been made to V's base tables. If the rematerialization rule for V is triggered, the view certainly is maintained correctly: V is recomputed from its base tables; all other rules for V are deactivated, so V cannot be modified until the base tables change again. Suppose the rematerialization rule is not triggered. During rule processing, first some rules delete tuples from V, then other rules insert tuples into V. Consider the deletions. For each type of table reference, our theorems guarantee that the generated **delete** operations never delete tuples that should remain in V. Furthermore, these operations always delete all tuples that should no longer be in V. Consider the insertions. First, notice that all generated **insert** operations use nested **select** expressions based on the view definition itself. Since we know the view definition cannot produce duplicates, the set of tuples in **insert** operations never includes duplicates. Furthermore, our theorems (along with the "`not in inserted V`" clauses) guarantee that tuples already in V are never inserted. Finally, in each case the **insert** operations produce all tuples that should be added to V.

We must consider that other production rules in addition to view-maintaining rules may be defined in the system. Although these rules cannot modify views, they can modify base tables. Our view-maintaining rules behave correctly even in the presence of other rules, and no additional rule ordering is necessary. Recall the semantics of rule execution (Section 3): a rule is considered with respect to the transition since the last time its action was executed; if its action has not yet been executed, it is considered with respect to the transition since the last rule assertion point (or start of the transaction). Hence, the first time a view-maintaining rule R is triggered during rule processing, it processes all base table changes since the last assertion point. Suppose that, subsequently during rule processing, the base tables are changed by a non-view-maintaining rule. Then R will be triggered again and will modify the view according to the new set of changes. When rule processing terminates, no rules are triggered, so all view-maintaining rules will have processed all relevant changes to base tables.

10 Conclusions and Future Work

We have described a facility that automatically derives a set of production rules to maintain a materialization of a user-defined view. This approach both frees the view definer from handling view maintenance and guarantees that the view remains correct. Through analysis techniques based on key information, incremental maintenance rules are generated whenever possible. Our facility allows the user to interact with the system: view definitions and key information can be modified to guarantee that the system produces efficient maintenance rules for frequent base table operations. In practice, efficient rules are possible for a wide class of views—efficiency relies on safe table references, and it can be seen from our criteria for safety that table references routinely fall into this class. In those cases where efficiency is not possible for the user's desired view, our system provides recognition of this fact; the user either may use the rules produced for automatic rematerialization or may decide that query modification is more appropriate.

We plan to implement our facility using the Starburst Rule System, then conduct experiments to evaluate the run-time efficiency of our approach on a variety of views. Meanwhile, we want to extend view analysis and rule generation so that the full power of SQL **select** statements can be used in view definitions. (We have started this and expect it to be tedious but not difficult.) Currently, the biggest drawback of our approach is that views with duplicates are not handled; we will consider ways to remove this restriction. We would like to add automatic rule optimization as a post-rule-generation component in our system. The rules produced by our method have a standard form, and in some cases can be optimized as in [CW90]. In addition, rules for different views could be merged and common subexpressions could be exploited as in [Han87]. Finally, the properties guaranteed by our algorithms are useful in other areas (such as query optimization), and we intend to explore this connection.

Acknowledgements

Thanks to Guy Lohman and Laura Haas for helpful comments on an initial draft.

References

[ACL91] R. Agrawal, R.J. Cochrane, and B. Lindsay. On maintaining priorities in a production rule system. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 479–487, Barcelona, Spain, September 1991.

[BLT86] J.A. Blakeley, P.-A. Larson, and F.W. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–71, Washington, D.C., June 1986.

[CG85] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, April 1985.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 566–577, Brisbane, Australia, August 1990.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. IBM Research Report RJ 8027, IBM Almaden Research Center, San Jose, California, March 1991.

[DE89] L.M.L. Delcambre and J.N. Etheredge. The Relational Production Language: A production language for relational databases. In L. Kerschberg, editor, *Expert Database Systems—Proceedings from the Second International Conference*, pages 333–351. Benjamin/Cummings, Redwood City, California, 1989.

[Han87] E. Hanson. *Efficient Support for Rules and Derived Objects in Relational Database Systems.* PhD thesis, University of California, Berkeley, August 1987.

[IBM88] IBM Form Number SC26-4348-1. *IBM Systems Application Architecture, Common Programming Interface: Database Reference*, October 1988.

[Kim82] W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.

[KP81] S. Koenig and R. Paige. A transformational framework for the automatic control of derived data. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 306–318, Cannes, France, September 1981.

[MD89] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215–224, Portland, Oregon, May 1989.

[RCBB89] A. Rosenthal, S. Chakravarthy, B. Blaustein, and J. Blakeley. Situation monitoring for active databases. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 455–464, Amsterdam, The Netherlands, August 1989.

[SI84] O. Shmueli and A. Itai. Maintenance of views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 240–255, Boston, Massachusetts, May 1984.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 281–290, Atlantic City, New Jersey, May 1990.

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

[Sto75] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 65–78, San Jose, California, May 1975.

[WCL91] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 275–285, Barcelona, Spain, September 1991.

[WF90] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, Atlantic City, New Jersey, May 1990.

[Wid91] J. Widom. Deduction in the Starburst production rule system. IBM Research Report RJ 8135, IBM Almaden Research Center, San Jose, California, May 1991.