Efficiently Updating Materialized Views*

José A Blakeley, Per-Åke Larson, Frank Wm Tompa

Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1

Abstract

Query processing can be sped up by keeping frequently accessed users' views materialized. However, the need to access base relations in response to queries can be avoided only if the materialized view is adequately maintained. We propose a method in which all database updates to base relations are first filtered to remove from consideration those that cannot possibly affect the view. The conditions given for the detection of updates of this type, called irrelevant updates, are necessary and sufficient and are independent of the database state. For the remaining database updates, a differential algorithm can be applied to re-evaluate the view expression. The algorithm proposed exploits the knowledge provided by both the view definition expression and the database update operations.

1 Introduction

In a relational database system, a database may be composed of both base and derived relations. A derived relation—or view—is defined by a relational expression (i.e., a query evaluated over the base relations). A derived relation may be virtual, which corresponds to the traditional concept of a view, or materialized, which means that the resulting relation is actually stored. As the database changes because of updates applied to the base relations, the materialized views may also require change. A materialized view can always be brought up to date by re-evaluating the relational expression that defines it. However, complete re-evaluation is often wasteful, and the cost involved may be unacceptable.

The need for a mechanism to update materialized views efficiently has been expressed by several authors. Gardarin et al [GSV84] consider concrete views (i.e., materialized views) as a candidate approach for the support of real-time queries. However, they discard this approach because of the lack of an efficient algorithm to keep the concrete views up to date with the base relations. Horwitz and Tellelbaum [HT85] propose a model for the generation of language-based environments which uses a relational database along with attribute grammars, and they suggest algorithms for incrementally updating views, motivated by the efficiency requirements of interactive editing. Buneman and Clemons [BC79] propose views for the support of alerters, which monitor a database and report to some user or application whether a state of the database, described by the view definition, has been reached.

It must be stressed that the problem analyzed in this paper is different from the traditional view update problem. In the traditional view update problem, a user is allowed to pose updates directly to a view, and the difficulty is in determining how to...
translate updates expressed against a view into updates to the base relations. In the model proposed in this paper, the user can only update base relations, direct updates to views are not considered. Therefore, rather than analysing the traditional problem of deriving appropriate update translations, this paper is concerned with finding efficient ways of keeping materialised views up to date with the base relations.

The purpose of this paper is to present a framework for the efficient update of materialised views when base relations are subject to updates. Section 2 presents some previous related work, Section 3 presents the notation and terminology used throughout the paper, Section 4 describes how to detect updates that have no effect on a view, Section 5 describes a method for differentially updating materialised views; finally, Section 6 contains some conclusions and suggestions for further research.

2 Previous work

Work directly related to the maintenance of materialised views has been reported by Koenig and Paige [KP81] and by Shmueli and Itai [SI84]. Koenig and Paige [KP81] investigate the support of derived data in the context of a functional binary-association data model. This data model puts together ideas borrowed from binary-association models, functional models, and the entity-relationship model, within a programming language suitable for data definition and manipulation. In their model, views can be explicitly stored and then maintained. For each possible change to the operands of the view, there exists a procedure associated with this change that incrementally updates the view. This procedure is called the derivative of the view definition with respect to the change. Their approach relies on the availability of such derivatives for various view definition/change statement combinations.

Shmueli and Itai's approach consists of continuously maintaining an acyclic database, together with information that may be useful for future insertions and deletions. Their definition of views is limited to the projection of a set of attributes over the natural join of all the relations in the database scheme. This is a restricted class of views, since views based on the join of some, but not all, of the relations in the database scheme cannot be handled by this mechanism.

Another restriction on the views is the omission of selection conditions.

In related work, Hammer and Sarin [HS78] present a method for efficiently detecting violations of integrity constraints, called integrity assertions, as a result of database updates. For each integrity assertion, there exists an error-predicate which corresponds to the logical complement of the assertion. If the error-predicate is true for some instance of the database, then the instance violates the assertion. Their approach to the problem of efficiently checking database assertions is based on analysing the potential effects that an update operation may have on the assertions. This analysis is performed by a compile-time assertion processor. The result is a set of candidate tests that will be executed at run-time to determine if the update causes the assertion to be violated. The selection of the least expensive test from the set of candidate tests requires a procedure similar to the one required in query optimisation.

Buneman and Clemons [BC79] propose a procedure for the efficient implementation of alerters. In general, the condition that triggers an alerter is expressed in terms of a query—called the target relation—over several base relations; in our terminology, a target relation corresponds to a virtual view. One aspect that is emphasized in their work is the efficient detection of base relation updates that are of no interest to an alerter, thus determining when re-evaluation of the associated query is unnecessary.

3 Notation and terminology

We assume that the reader is familiar with the basic ideas and notation concerning relational databases, as described in [M83]. A view definition $V$ corresponds to a relational algebra expression on the database scheme. A view materialisation $v$ is a stored relation resulting from the evaluation of this relational algebra expression against an instance of the database. In this paper, we consider only relational algebra expressions formed from the combination of selections, projections, and joins, called SPJ expressions.

A transaction is an indivisible sequence of update operations to base relations. Indivisible means that
either all the update operations are successfully performed or none are performed. Furthermore, updates within a transaction may update several base relations.

Considering that base relations are updated before the views, it is reasonable to assume that the complete affected tuples from the base relations are available at the time the view is to be updated. The net effect of a transaction on a base relation can be represented by a set of tuples that have been inserted and a set of tuples that have been deleted. Formally, given a base relation \( r \) and a transaction \( T \), there exist sets of tuples \( t_r \) and \( d_r \) such that \( r \), \( t_r \), and \( d_r \) are disjoint and \( T(r) = r \cup t_r - d_r \). Therefore, without any loss of generality we will represent a transaction applied to a base relation \( T(R) \) by \( \text{insert}(R, t_r) \) and \( \text{delete}(R, d_r) \), where \( R \) is the name of the base relation with instance \( r \) such that \( r \), \( t_r \), \( d_r \) are mutually disjoint.

It is assumed that all attributes are defined on discrete and finite domains. Since such a domain can be mapped to a subset of natural numbers, we use integer values in all examples.

### 4 Relevant and irrelevant updates

In certain cases, a set of updates to a base relation has no effect on the state of a view. When this occurs independently of the database state, we call the set of updates irrelevant. It is important to provide an efficient mechanism for detecting irrelevant updates so that re-evaluation of the relational expression defining a view can be avoided or the number of tuples considered can be reduced.

Consider a view defined by the expression

\[
v = \pi_X(\sigma_C(Y)(r_1 \times r_2 \times \cdots \times r_p))
\]

where \( C(Y) \) is a Boolean expression and \( X \) and \( Y \) are sets of variables denoting the names of some attributes for the relations named \( R_1, R_2, \ldots, R_p \). The sets \( X \) and \( Y \) are not necessarily equal (i.e., not all the attributes in the projection participate in the selection condition and vice versa), and in fact may be disjoint.

Suppose that a tuple \( t = (a_1, a_2, \ldots, a_d) \) is inserted into (or deleted from) relation \( r_k \) defined on scheme \( R_k \). Let \( Y_1 = R_k \cap Y \), and \( Y_2 = Y - Y_1 \), so that \( Y = Y_1 \cup Y_2 \). Let the selection condition \( C(Y) \) be modified by replacing the variables \( Y_1 \) by their corresponding values \( t(Y_1) \). If the modified condition \( C(Y) \) can be shown to be unsatisfiable regardless of the database state, then inserting or deleting \( t \) from \( r_k \) has no effect on the view \( v \).

**Example 4.1** Consider two relations \( r \) and \( s \) defined on \( R = \{A, B\} \) and \( S = \{C, D\} \), respectively, and a view \( v \) defined as

\[
v = \pi_{A, D}(\sigma_{(A < 10) \land (C > 5)}(B = C)(r \times s))
\]

That is, \( C(A, B, C) = (A < 10) \land (C > 5) \land (B = C) \).

<table>
<thead>
<tr>
<th>( r )</th>
<th>( A )</th>
<th>( B )</th>
<th>( s )</th>
<th>( C )</th>
<th>( D )</th>
<th>( v )</th>
<th>( A )</th>
<th>( D )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>20</td>
<td>5</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>10</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Suppose that the tuple \( (9, 10) \) is inserted into relation \( r \). We can substitute the values \( (9, 10) \) for the variables \( A \) and \( B \) in \( C(A, B, C) \) to obtain the modified condition \( C(9, 10, C) = (9 < 10) \land (C > 5) \land (10 = C) \). The selection condition \( C(9, 10, C) \) is satisfiable, that is, there exist instances of the relations named \( R \) and \( S \) containing the tuples \( (9, 10) \) and \( (10, 5) \), for some value of \( C \) such that \( C(9, 10, C) = \text{True} \). Therefore, inserting the tuple \( (9, 10) \) into relation \( r \) is relevant to the view \( v \). Notice that there may be some state of \( s \) that contains no matching tuple \( (10, 5) \), in which case the tuple \( (9, 10) \) will have no effect on the view. However, the only way of verifying this is by checking the contents of the database.

On the other hand, suppose that the tuple \( (11, 10) \) is inserted into relation \( r \). After substituting the values \( (11, 10) \) for the variables \( A \) and \( B \) in \( C(A, B, C) \) we obtain

\[
C(11, 10, C) = (11 < 10) \land (C > 5) \land (10 = C)
\]

We can see that \( C \) is now unsatisfiable regardless of the database state. Therefore, inserting the tuple \( (11, 10) \) into relation \( r \) is (provably) irrelevant to the view \( v \).

The same argument applies for deletions. That is, if substituting the values of the deleted tuple in the selection condition makes the selection condition
unsatisfiable regardless of the database state, then
the deleted tuple is irrelevant to the view In other
words, the deleted tuple is not visible in the view
Similarly, if substituting the values of the deleted
tuple in the selection condition makes the selection
condition satisfiable, then the deleted tuple may need
to be removed from the view

**Definition 4.1** Consider a view

\[ v = \pi_X (\sigma_{C(Y)}(r_1 \times r_2 \times \cdots \times r_p)) \]

and a tuple \( t = (a_1, a_2, \ldots, a_q) \in \text{r} \), defined on \( R \),
for some \( i, 1 \leq i \leq p \) Let \( Y_1 = Y \cap R \), and \( Y_2 = Y - Y_1 \) Denote by \( C(t, Y_2) \) the modified selection
condition \( C(Y) \) obtained when substituting the variable \( A \in Y_1 \)
in \( C(Y) \) \( C(t, Y_2) \) is said to be a substitution of \( t \) for \( Y_1 \)

**Theorem 4.1** Consider a view

\[ v = \pi_X (\sigma_{C(Y)}(r_1 \times r_2 \times \cdots \times r_p)) \]

and a tuple \( t \) inserted into (or deleted from) \( r_i \), defined on \( R \),
for some \( i, 1 \leq i \leq p \) Let \( Y_1 = Y \cap R \), and \( Y_2 = Y - Y_1 \) The update involving tuple \( t \) is
irrelevant to the view \( v \) (for every database instance \( D \)) if and only if \( C(t, Y_2) \) is unsatisfiable

**Proof:** (if) If the substitution of \( C(t, Y_2) \) is unsatis-
fiable, then no matter what the current state of the
database is, \( C(t, Y_2) \) evaluates to false and therefore
does not affect the view That is, if \( t \) were inserted
it could not cause any new tuples to become visible
in the view, and if \( t \) were deleted it could not cause
any tuples to be deleted from the view. Hence, the
tuple \( t \) is irrelevant to the view \( v \)

(only if) Assume that the tuple \( t \) is irrelevant to the
view and that \( C(t, Y_2) \) is satisfiable \( C(t, Y_2) \) being
satisfiable means that there exists a database instance \( D_0 \) for which a substitution of values \( u \) for \( Y_2 \)
in \( C(t, Y_2) \) makes the selection condition true. To
construct such a database instance we need to find
at least \( p - 1 \) tuples \( t_j \in r_j, 1 \leq j \leq p \) and \( j \neq i \)
since \( t \in r_i \), in such a way that

\[ \pi_X (\sigma_{C(Y)}(\{t_1\} \times \{t_2\} \times \cdots \times \{t_p\})) \neq \emptyset. \]

1) For all attributes \( A \) such that \( A \in R \), and \( A \in Y_1 \),
replace \( t_j(A), 1 \leq j \leq p, j \neq i \) by \( t(A) \)

ii) For all attributes \( B \notin Y \), replace \( t_j(B), 1 \leq j \leq p, j \neq i \) by any value, say one

iii) For all attributes \( C \in Y_2 \), replace \( t_j(C), 1 \leq j \leq p, j \neq i \) by any value in the domain of \( C \) that
makes \( C(t, Y_2) \) true. Such values are guaranteed
to exist because \( C(t, Y_2) \) is satisfiable.

The database instance \( D_0 \) consists of \( p \) relations

\[ r_1 = \{t_1\}, r_2 = \{t_2\}, \ldots, r_s = \emptyset, \ldots, r_p = \{t_p\} \]

Clearly, the view state that corresponds to \( D_0 \) has
no tuples Creating \( D_1 \) from \( D_0 \) by inserting \( t \) into
\( r_i \) produces a view state with one tuple Thus the
insertion of \( t \) is relevant to the view \( v \) Similarly,
deleting \( t \) from \( D_1 \) shows that the deletion of \( t \) is also
relevant to the view \( v \) This proves that the condition
is necessary

Deciding the satisfiability of Boolean expressions
is in general NP-complete However, there is a large
class of Boolean expressions for which satisfiability
can be decided efficiently, as shown by Rosenkrantz
and Hunt [RH80] This class corresponds to expres-
sions formed from the conjunction of atomic formulae
of the form \( x \odot y \), \( x \odot c \), and \( x \odot y + c \), where \( x \)
and \( y \) are variables defined on discrete and infinite
domains, \( c \) is a positive or negative constant, and
\( \odot \in \{=, <, >, \leq, \geq \} \) The improved efficiency arises
from not allowing the operator \( \neq \) in \( \odot \)

Deciding whether a conjunctive expression in the
class described above is satisfiable can be done in
time \( O(n^3) \) where \( n \) is the number of variables con-
tained in the expression. The sketch of the algorithm
is as follows (1) the conjunctive expression is nor-
malised, that is, it is transformed into an equivalent
one where only the operators \( \leq \) or \( \geq \) are used in
the atomic formulae; (2) a directed weighted graph
is constructed to represent the normalised expres-
sion, and (3) if the directed graph contains a cycle
for which the sum of its weights is negative then the
expression is unsatisfiable, otherwise it is satisfiable.
To find whether a directed weighted graph contains
a negative cycle one can use Floyd's algorithm [F62],
which finds all the shortest paths between any two
nodes in a directed weighted graph

We can also decide efficiently the satisfiability of
Boolean expressions of the form

\[ C = C_1 \lor C_2 \lor \cdots \lor C_m \]
where, \( C, 1 = 1, \ldots, m \) is a conjunctive expression in the class described above. The expression \( C \) is satisfiable if and only if at least one of the conjunctive expressions \( C_i \) is satisfiable. Similarly, \( C \) is unsatisfiable if and only if each of the conjunctive expressions \( C_i \) is unsatisfiable. We can apply Rosenkrantz and Hunt's algorithm to each of the conjunctive expressions \( C_i \), this takes time \( O(mn^3) \) in the worst case, where \( n \) is the number of different variables mentioned in \( C \).

### 4.1 Detection of relevant updates

This section presents an algorithm to detect those relation updates that are relevant to a view. Before describing the algorithm we need another definition.

**Definition 4.2** Consider a conjunctive expression \( C(Y) \), and a tuple \( t = (a_1, a_2, \ldots, a_q) \in r \) defined on \( R \). Let \( \alpha(C) \) denote the set of variables that participate in \( C, Y = \alpha(C), Y_1 = Y \cap R, Y_2 = Y - Y_1, \) and \( C(t, Y_2) \) be the substitution of \( t \) for \( Y_1 \) in \( C \). We distinguish between two types of atomic formulae in \( C(t, Y_2) \) called variant and invariant formulae respectively.

1. **Variant formulae** are those directly affected by the substitution of \( t(A) \) for \( A \in Y_1 \) in \( C \). This type of formula may have the form \((z \text{ op } c)\), or \((c \text{ op } d)\), where \( x \) is a variable and \( c, d \) are constants. Furthermore, formulae of the form \((x \text{ op } c)\) are called variant non-evaluable formulae, and formulae of the form \((c \text{ op } d)\) are called variant evaluable formulae. Variant evaluable formulae are either true or false.

2. **Invariant formulae** are those that remain invariant with respect to the substitution of \( t \) for \( Y_1 \) in \( C \). This type of formula may have the form \((x \text{ op } y + c)\), where \( x, y \) are variables, and \( c \) is a constant. That is, the attributes \( X, Y \) represented by the variables \( x, y \) are not in \( Y_1 \).

Notice that the classification of atomic formulae in \( C \) depends on the relation scheme of the set of tuples \( t \) substituting for attributes \( Y_1 \) in \( C \).

**Algorithm 4.1**

The input to the algorithm consists of

1. a conjunctive Boolean expression
   \[ C = f_1 \land f_2 \land \cdots \land f_n, \]
   where each \( f_i, 1 \leq i \leq n \), is an atomic formula of the form \((x \text{ op } y)\), \((x \text{ op } y + c)\), or \((x \text{ op } c)\), where \( x, y \) are variables (representing attributes) and \( c \) is a constant,

2. a relation scheme \( R \) of the updated relation, and

3. a set of tuples \( T_{in} = \{ t_1, t_2, \ldots, t_n \} \) on scheme \( R \)

   \( T_{in} \) contains those tuples inserted to or deleted from the relation \( r \).

The output from the algorithm consists of a set of tuples \( T_{out} \subseteq T_{in} \) which are relevant to the view.

1. The conjunctive expression \( C \) is normalized.

2. The normalized conjunctive expression \( C_{N} \) is expressed as \( C_{INV} \land C_{EVAL} \land C_{NEVAL} \)

   - \( C_{INV} \) is a conjunctive subexpression containing only invariant formulae.
   - \( C_{EVAL} \) is a conjunctive subexpression containing only variant evaluable formulae.
   - \( C_{NEVAL} \) is a conjunctive subexpression containing only variant non-evaluable formulae.

3. Using \( C_{INV} \), build the invariant portion of the directed weighted graph.

4. For each tuple \( t \in T_{in} \), substitute the values of \( t \) for the appropriate variables in \( C_{EVAL} \) and \( C_{NEVAL} \). Build the variant portion of the graph and check whether the substituted conjunctive expression represented by the graph is satisfiable. If the expression is satisfiable, then add \( t \) to \( T_{out} \); otherwise ignore it.

An important component of the algorithm is the construction of a directed weighted graph \( G = (n, e) \), where \( n = \alpha(C) \cup \{0\} \) is the set of nodes, and \( e \) is the set of directed weighted edges representing atomic formulae in \( C \). Each member of \( e \) is a triple \((n_0, n_d, w)\), where \( n_0, n_d \in n \) are the origin and destination nodes respectively, and \( w \) is the weight of the edge. The atomic formula \((x \leq y + c)\) translates to the edge \((x, y, c)\). The atomic formula \((x \geq y + c)\) translates to the edge \((y, x, -c)\).
\((x \leq c)\) translates to the edge \((0, x, c)\). The atomic formula \((x \geq c)\) translates to the edge \((x', 0, -c)\).

The normalization procedure mentioned in the algorithm takes a conjunctive expression and transforms it into an equivalent one where each atomic formula has as comparison operator either \(\leq\) or \(\geq\). Atomic formulae \((x < y + c)\) are transformed into \((x < y + c - 1)\). Atomic formulae \((x > y + c)\) are transformed into \((x \geq y + c + 1)\). Atomic formulae \((x = y + c)\) are transformed into \((x \leq y + c) \land (x \geq y + c)\).

The satisfiability test consists of checking whether the directed weighted graph contains a negative weight cycle or not. The expression is unsatisfiable if the graph contains a negative cycle.

We can generalize Definition 4.1 to allow substitutions of several tuples for variables in an expression \(C\).

**Definition 4.3** Consider a view

\[ u = \pi_X(r_{C(Y)}(r_1 \times r_2 \times \ldots \times r_p)) \]

and tuples \(t_i \in r_i, 1 \leq i \leq k\). Assume that \(R_i \cap R_j = \emptyset\) for all \(i \neq j\). Let \(Y_1 = Y \cap (R_1 \cup R_2 \cup \ldots \cup R_k)\) and \(Y_2 = Y - Y_1\). Denote by \(C(t_1, t_2, \ldots, t_k, Y_2)\) the modified selection condition obtained when substituting the values \(t_i(X), 1 \leq i \leq k\), for each occurrence of the variable \(A \in Y_1\) in \(C(Y)\). \(C(t_1, t_2, \ldots, t_k, Y_2)\) is said to be the substitution of \(t_1, t_2, \ldots, t_k\) for \(Y_1\) in \(C\).

**Theorem 4.2** Consider a view

\[ u = \pi_X(\sigma_{C(Y)}(r_1 \times r_2 \times \ldots \times r_p)), \]

and tuples \(t_1, t_2, \ldots, t_k\) all either inserted to or deleted from relations \(r_1, r_2, \ldots, r_k\) respectively. Let \(Y_1\) and \(Y_2\) be defined as before. The set of tuples \(\{t_1, t_2, \ldots, t_k\}\) is irrelevant to the view \(u\) (for every database instance \(D\)) if and only if \(C(t_1, t_2, \ldots, t_k, Y_2)\) is unsatisfiable.

**Proof:** Similar to the proof of Theorem 4.1.

While we do not propose the statement of Theorem 4.2 as the basis of an implementation for the detection of irrelevant updates, it shows that the detection of irrelevant updates can be taken further by considering combinations of tuples from different relations.

## 5 Differential re-evaluation of views

The purpose of this section is to present an algorithm to update a view differentially as a result of updates to base relations participating in the view definition. Differential update means bringing the materialized view up to date by identifying which tuples must be inserted into or deleted from the current instance of the view.

For simplicity, it is assumed that the base relations are updated by transactions and that the differential update mechanism is invoked as the last operation within the transaction (i.e., as part of the **commit** of the transaction). It is also assumed that the information available when the differential view update mechanism is invoked consists of (a) the contents of each base relation before the execution of the transaction, (b) the set of tuples actually inserted into or deleted from each base relation, (c) the view definition, and (d) the contents of the view that agrees with the contents of the base relations before the execution of the transaction. Notice in particular that (b) only includes the net changes to the relations for example, if a tuple not in the relation is inserted and then deleted within a transaction, it is not represented at all in this set of changes.

### 5.1 Select views

A **select view** is defined by the expression \(V = \sigma_{C(Y)}(R)\), where \(C\) (the selection condition) is a Boolean expression defined on \(Y \subseteq R\). Let \(r_t\) and \(d_r\) denote the set of tuples inserted into or deleted from relation \(r\), respectively. The new state of the view, called \(u'\), is computed by the expression \(u' = u \cup \sigma_{C(Y)}(r_t) - \sigma_{C(Y)}(d_r)\). That is, the view can be updated by the sequence of operations

\[
\begin{align*}
\text{insert}(V, \sigma_{C(Y)}(r_t)) \\
\text{delete}(V, \sigma_{C(Y)}(d_r))
\end{align*}
\]

Assuming \(|u| \gg |d_r|\), it is cheaper to update the view by the above sequence of operations than re-computing the expression \(V\) from scratch.
5.2 Project views

A project view is defined by the expression \( V = \pi_X(R) \), where \( X \subseteq R \). The project operation introduces the first difficulty to updating views differentially. The difficulty arises when the base relation \( r \) is updated through a delete operation.

Example 5.1 Consider a relation scheme \( R = \{ A, B \} \), a project view defined as \( \pi_B(R) \), and the relation \( r \) shown below:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

If the operation \( \text{delete}(R, \{(3, 20)\}) \) is applied to relation \( r \), then the view can be updated by the operation \( \text{delete}(V, \{20\}) \). However, if the operation \( \text{delete}(R, \{(1, 10)\}) \) is applied to relation \( r \), then the view cannot be updated by the operation \( \text{delete}(V, \{10\}) \). The reason for this difficulty is that the distributive property of projection over difference does not hold (i.e., \( \pi_X(r_1 - r_2) \neq \pi_X(r_1) - \pi_X(r_2) \)).

There are two alternatives for solving the problem:

1. Attach an additional attribute to each tuple in the view, a multiplicity counter, which records the number of operand tuples that contribute to the tuple in the view. Inserting a tuple already in the view causes the counter for that tuple to be incremented by one. Deleting a tuple from the view causes the counter for that tuple to be decremented by one, if the counter becomes zero, then the tuple in the view can be safely deleted.

2. Include the key of the underlying relation within the set of attributes projected in the view. This alternative allows unique identification of each tuple in the view. Insertions or deletions cause no trouble since the tuples in the view are uniquely identified.

We choose alternative (1) since we do not want to impose restrictions on the views other than the class of relational algebra expressions allowed in their definition. In addition, alternative (2) becomes an special case of alternative (1) in which every tuple in the view has a counter value of one.

We require that base relations and views include an additional attribute, which we will denote \( N \). For base relations, this attribute need not be explicitly stored since its value in every tuple is always one. The select operation is not affected by this assumption. The project operation is re-defined as \( \pi_X(r) = \{ t(X') | X' = X \cup \{ N \} \text{ and } \exists u \in r \left( (u(X) = t(X)) \land (t(N) = \sum_{w \in W} w(N)) \right) \} \). Notice that by re-defining the project operation, the distributive property of projection over difference now holds (i.e., \( \pi_X(r_1 - r_2) = \pi_X(r_1) - \pi_X(r_2) \)).

To complete the definition of operators to include the multiplicity counter the join operation is re-defined as \( r \bowtie s = \{ t(Y_1) | Y_1 = R \cup S \text{ and } \exists u, v \left( (u \in r) \land (v \in s) \land (t(R - \{ N \}) = u(R - \{ N \}) \land (t(S - \{ N \}) = v(S - \{ N \}) \land (t(N) = u(N) \cdot v(N))) \right) \}, \) where \(' \cdot '\) denotes scalar multiplication.

5.3 Join views

A join view is defined by the expression

\[ V = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_p \]

We consider first changes to the base relations exclusively through insert operations, next we consider changes to the base relations exclusively through delete operations, and finally we consider changes to the base relations through both insert and delete operations.

Example 5.2 Consider two relation schemes \( R = \{ A, B \} \) and \( S = \{ B, C \} \), and a view \( V \) defined as \( V = R \bowtie S \). Suppose that after the view \( v \) is materialized, the relation \( r \) is updated by the insertion of the set of tuples \( t_r \). Let \( r' = r \cup t_r \). The new state of the view, called \( v' \), is computed by the expression

\[ v' = (r' \bowtie s) = (r \cup t_r) \bowtie s = (r \bowtie s) \cup (t_r \bowtie s) \]

If \( t_v = t_r \bowtie s \), then \( v' = v \cup t_v \). That is, the view can be updated by inserting only the new set of tuples.
In other words, one only needs to compute the contribution of the new tuples in \( r \) to the join. Clearly, it is cheaper to compute the view \( v' \) by adding \( v_0 \) to \( v \) than to re-compute the join completely from scratch.

This idea can be generalized to views defined as the join of an arbitrary number of base relations by exploiting the distributive property of join with respect to union.

Consider a database \( P = \{ r_1, r_2, \ldots, r_p \} \) and a view \( V \) defined as \( V = r_1 \Join r_2 \Join \cdots \Join r_p \). Let \( v \) denote the materialized view, and the relations \( r_1, r_2, \ldots, r_p \) be updated by inserting the sets of tuples \( t_{r_1}, t_{r_2}, \ldots, t_{r_p} \). The new state of the view \( v' \) can be computed as

\[
v' = (t_{r_1} \cup t_{r_1}) \Join (t_{r_2} \cup t_{r_2}) \Join \cdots \Join (t_{r_p} \cup t_{r_p})
\]

Let us associate a binary variable \( B_s \) with each of the relation schemes \( R_s, 1 \leq s \leq p \). The value zero for \( B_s \) refers to the tuples of \( r_s \) considered during the current materialization of the view \( v \) (i.e., the old tuples), and the value one for \( B_s \) refers to the set of tuples inserted into \( r_s \), since the latest materialization of \( v \) (i.e., the new tuples \( t_r \)). The expansion of the expression for \( v' \), using the distributive property of join over union, can be depicted by the truth table of the variables \( B_s \). For example, if \( p = 3 \) we have

\[
\begin{array}{ccc}
B_1 & B_2 & B_3 \\
0 & 0 & 0 & r_1 \Join r_2 \Join r_3 \\
0 & 0 & 1 & \text{which} & r_1 \Join r_2 \Join t_{r_3} \\
0 & 1 & 0 & \text{represents} & r_1 \Join r_2 \Join t_{r_3} \\
0 & 1 & 1 & \text{represents} & t_{r_1} \Join r_2 \Join t_{r_3} \\
1 & 0 & 1 & t_{r_1} \Join r_2 \Join r_3 \\
1 & 1 & 0 & t_{r_1} \Join t_{r_2} \Join r_3 \\
1 & 1 & 1 & t_{r_1} \Join t_{r_2} \Join t_{r_3}
\end{array}
\]

where the union of all expressions in the right hand side of the table is equivalent to \( v' \). The first row of the truth table corresponds to the join of the base relations considering only old tuples (i.e., the current state of the view \( v \)). Typically, a transaction would not insert tuples into all the relations involved in a view definition. In that case, some of the combinations of joins represented by the rows of the truth table correspond to null relations. Using the table for \( p = 3 \), suppose that a transaction contains insertions to relations \( r_1 \) and \( r_2 \) only. One can then discard all the rows of the truth table for which the variable \( B_3 \) has a value of one, namely rows 2, 4, 6, and 8. Row 1 can also be discarded, since it corresponds to the current materialization of the view. Therefore, to bring the view up to date we need to compute only the joins represented by rows 3, 5, and 7. That is,

\[
v' = v \cup (r_1 \Join t_{r_2} \Join r_3) \\
\cup (t_{r_1} \Join r_2 \Join r_3) \\
\cup (t_{r_1} \Join t_{r_2} \Join r_3)
\]

The computation of this differential update of the view \( v \) is certainly cheaper than re-computing the whole join.

So far we have assumed that the base relations change only through the insertion of new tuples. The same idea can be applied when the base relations change only through the deletion of old tuples.

Example 5.5 Consider again two relation schemes \( R = \{A, B\} \) and \( S = \{B, C\} \), and the view \( V \) defined as \( V = R \Join S \). Suppose that after the view \( v \) is materialized, the relation \( r \) is updated by the deletion of the set of tuples \( d_r \). Let \( r' = r - d_r \). The new state of the view, called \( v' \), is computed as

\[
v' = r' \Join s = (r - d_r) \Join s = (r \Join s) - (d_r \Join s)
\]

If \( d_r = d_v \Join s \), then \( v' = v - d_v \). That is, the view can be updated by deleting the new set of tuples \( d_v \) from the relation \( v \). It is not always cheaper to compute the view \( v' \) by deleting from \( v \) only the tuples \( d_v \), however, this is true when \( |v| > |d_v| \).

The differential update computation for deletions can also be expressed by means of binary tables. Thus, the computation of differential updates depends on the ability to identify which tuples have been inserted and which tuples have been deleted. From now on, all tuples are assumed to be tagged in such a way that it is possible to identify inserted, deleted, and old tuples.
Example 5.4 Consider two relation schemes \( R = \{A, B\} \) and \( S = \{B, C\} \), and a view \( V \) defined as \( V = R \bowtie S \). Let \( r \) and \( s \) denote instances of the relations named \( R \) and \( S \), respectively, and \( v = r \bowtie s \). Assume that a transaction \( T \) updates relations \( r \) and \( s \).

Case 1: \( t \in r \land \neg d_a \) is a tuple that has to be inserted into \( u \).
Case 2: \( t \in r \land d_a \) is a tuple that has no effect in the view \( u \), and can therefore be ignored.
Case 3: \( t \in r \land s \) is a tuple that has to be inserted into \( u \).
Case 4: \( t \in d_r \land d_s \) is a tuple that has to be deleted from \( u \).
Case 5: \( t \in d_r \land s \) is a tuple that has to be deleted from \( u \).
Case 6: \( t \in r \land s \) is a tuple that already exists in the view \( u \).

In general, we can describe the value of the tag field of the tuple resulting from a join of two tuples according to the following table:

<table>
<thead>
<tr>
<th>( r_1 )</th>
<th>( r_2 )</th>
<th>( r_1 \bowtie r_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>insert</td>
<td>insert</td>
</tr>
<tr>
<td>insert</td>
<td>delete</td>
<td>ignore</td>
</tr>
<tr>
<td>insert</td>
<td>old</td>
<td>insert</td>
</tr>
<tr>
<td>delete</td>
<td>insert</td>
<td>ignore</td>
</tr>
<tr>
<td>delete</td>
<td>delete</td>
<td>delete</td>
</tr>
<tr>
<td>delete</td>
<td>old</td>
<td>delete</td>
</tr>
<tr>
<td>old</td>
<td>insert</td>
<td>insert</td>
</tr>
<tr>
<td>old</td>
<td>delete</td>
<td>delete</td>
</tr>
<tr>
<td>old</td>
<td>old</td>
<td>old</td>
</tr>
</tbody>
</table>

where the last column of the table shows the value of the tag attribute for the tuple resulting from the join of two tuples tagged according to the values under columns \( r_1 \) and \( r_2 \). Tuples tagged as "ignore" are assumed to be discarded when performing the join. In other words, they do not "emerge" from the join.

The semantics of the join operation has to be redefined once more to compute the tag value of each tuple resulting from the join based on the tag values of the operand tuples. In the presence of projection this will be in addition to the computation of the count value for each tuple resulting from the join as explained in the section on project views. Similarly, the tag value of the tuples resulting from a select or project operation is described in the following table.

<table>
<thead>
<tr>
<th>( r )</th>
<th>( \sigma_{C(Y)}(r) )</th>
<th>( \pi_X(r) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>insert</td>
<td>insert</td>
</tr>
<tr>
<td>delete</td>
<td>delete</td>
<td>delete</td>
</tr>
<tr>
<td>old</td>
<td>old</td>
<td>old</td>
</tr>
</tbody>
</table>

In practice, it is not necessary to build a table with \( 2^p \) rows. Instead, by knowing which relations have been modified, we can build only those rows of the table representing the necessary subexpressions to be evaluated. Assuming that only \( k \) such relations were modified, \( 1 \leq k \leq p \), building the table can be done in time \( O(2^k) \).

Once we know what subexpressions must be computed, we can further reduce the cost of materializing the view by using an algorithm to determine a good order for execution of the joins. Notice that a new feature of our problem is the possibility of saving computation by re-using partial subexpressions appearing in multiple rows within the table. Efficient solutions are being investigated.

### 5.4 Select-Project-Join views

A select-project-join view (SPJ view) is defined by the expression

\[
V = \pi_X(\sigma_{C(Y)}(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_p)),
\]

where \( X \) is a set of attributes and \( C(Y) \) is a Boolean expression. We can again exploit the distributive property of join, select, and project over union to provide a differential update algorithm for SPJ views.

Example 5.5 Consider two relation schemes \( R = \{A, B\} \) and \( S = \{B, C\} \), and a view defined as \( V = \pi_A(\sigma_{C(O>10)}(R \bowtie S)) \). Suppose that after the view \( v \) is materialized, the relation \( r \) is updated by the insertion of tuples \( t_r \). Let \( r' = r \cup t_r \). The new state of the view, called \( u' \), is computed by the expression

\[
u' = \pi_A(\sigma_{C(O>10)}(r' \bowtie s)) = \pi_A(\sigma_{C(O>10)}(r \cup t_r \bowtie s)) = \pi_A(\sigma_{C(O>10)}(r \bowtie s)) \cup \pi_A(\sigma_{C(O>10)}(t_r \bowtie s)) = v \cup \pi_A(\sigma_{C(O>10)}(t_r \bowtie s))
\]

If \( t_v = \pi_A(\sigma_{C(O>10)}(t_r \bowtie s)) \), then \( u' = v \cup t_v \). That is, the view can be updated by inserting only the new set of tuples \( t_v \) into the relation \( v \).
We can again use a binary table to find out what portions of the expression have to be computed to bring the materialized view up to date. To evaluate each SPJ expression associated with a row of the table, we can make use of some known algorithm such as QUEL's decomposition algorithm by Wong and Yousefi [WY70]. Once more, there is a possibility of saving computation by re-using partial computations common to several rows in the table.

We now present the outline of an algorithm to update SPJ views differentially.

**Algorithm 5.1**

The input consists of

i) the SPJ view definition

\[ V = \pi_X (\sigma_C (R_1 \bowtie R_2 \bowtie \ldots \bowtie R_p)) \]

ii) the contents of the base relations \( r_j \), \( 1 \leq j \leq p \), and

iii) the sets of updates to the base relations \( u_{r_j} \), \( 1 \leq j \leq p \).

The output of the algorithm consists of a transaction to update the view.

1. Build those rows of the truth table with \( p \) columns corresponding to the relations being updated.
2. For each row of the table, compute the associated SPJ expression substituting \( r_j \) when the binary variable \( B_j = 0 \), and \( u_{r_j} \) when \( B_j = 1 \).
3. Perform the union of results obtained for each computation in step 2. The transaction consists of inserting all tuples tagged as insert, and deleting all tuples tagged as delete.

Observe that (i) we can use for \( V \) an expression with a minimal number of joins. Such expression can be obtained at view definition time by the tableau method of Aho, Sagiv, and Ullman [ASU79] extended to handle inequality conditions [K80], and (ii) step 2 poses an interesting optimization problem, namely, the efficient execution of a set of SPJ expressions (all the same) whose operands represent different relations and where intermediate results can be re-used among several expressions.

### 6 Conclusions

A new mechanism for the maintenance of materialized views has been presented. The mechanism consists of two major components. First, necessary and sufficient conditions for the detection of database updates that are irrelevant to the view were given. Using previous results by Rosenkrantz and Hunt we defined a class of Boolean expressions for which this detection can be done efficiently. Our detection of irrelevant updates extends previous results presented by Buneman and Clemons and by Hammer and Sarm. Since their papers were presented in the contexts of trigger support and integrity enforcement, our results can be used in those contexts as well. Second, for relevant updates, a differential view update algorithm was given. This algorithm supports the class of views defined by SPJ expressions.

Our differential view update algorithm does not automatically provide the most efficient way of updating the view. Therefore, a next step in this direction is to determine under what circumstances differential re-evaluation is more efficient than complete re-evaluation of the expression defining the view.

This paper carries the assumption that the views are materialized every time a transaction updates the database. It is also possible to envision a mechanism in which materialized views are updated periodically or only on demand. Such materialized views are known as snapshots [AL80] and their maintenance mechanism as snapshot refresh. The approach proposed in this paper also applies to this environment, and further work in this direction is in progress.

### References


1System R* provides a differential snapshot refresh mechanism for snapshots defined by a selection and projection on a single base relation [L88]. However, details of this mechanism have not been published.
on Database Systems, Vol 4, No 4, December 1979, pages 435–454


[L85] Lindsay, Bruce G , Personal communication


