Updating Materialized Views: Detecting Conditionally Autonomously Computable Updates

By

Frank Wm. Tompa
Department of Computer Science
University of Waterloo
Waterloo, Ontario, N2L 3G1

and

José A. Blakeley
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 240

Updating Materialized Views: Detecting Conditionally Autonomously Computable Updates

by

Frank Wm. Tompa and José A. Blakeley

February, 1988
Updated Materialized Views: Detecting Conditionally Autonomously Computable Updates

Frank Wm. Tompa
Data Structuring Group,
Department of Computer Science,
University of Waterloo,
Waterloo, Ontario, N2L 3G1

José A. Blakeley
Computer Science Department,
Indiana University,
Lindley Hall 101,
Bloomington, IN 47405

February 11, 1988

Abstract

Access to a database through a user view can be serviced quickly when the view is materialized, that is, the transformed data is explicitly stored. In the presence of database updates, however, the materialised view can become costly to maintain; often it must be completely rederived from the base data using the view definition. Under some conditions the view can be updated directly given only the view definition, the current contents of the materialized view, and the update operation (still expressed against the base data), without accessing the base data itself.

In this paper, we consider relational views defined by projection, selection, and join. We present necessary and sufficient conditions on the view definition, contents, and update operations for insertions and deletions to be reflected in the view without reference to base data. Because the possibility of such view-based updating is dependent on the current contents of the view, we call the update conditionally autonomously computable.

Categories and Subject Descriptions: H.2.1 [Database Management]:Logical Design—Data Models; H.2.4 [Database Management]:Systems—Query Processing

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Database design, Relational databases, Conceptual relations, Stored relations, Pre-joined relations, Derived relations, Materialized views, Snapshots

This work was supported in part by the Department of Computer Science at the University of Waterloo and by grant A09292 from the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

A truly relational database management system supports derived relations, also called views, in addition to base relations. A derived relation is defined by a relational expression (a query) over the base relations or other derived relations. A derived relation may be virtual, meaning that its defining relational expression has to be re-evaluated every time a user refers to it, or materialized, meaning that the relation resulting from evaluating the expression defining the view is actually stored. In this paper we are interested in the study of materialized views.

The support of materialized views in a database management system is important as a means to speed up the performance of frequently posed queries, see for example, [1,9,12,17,20,19,25]. By materializing the result of a frequent query, subsequent accesses to the same data involve only the retrieval of the contents of the materialized view, and thus the effort involved in recomputing the associated query from scratch is avoided. The use of materialized views to speed up the performance of frequent queries is perhaps more important in a distributed environment where the cost of recomputing a query from scratch involves not only processing but also communication cost. Snapshots are an example of the use of materialized views in distributed systems [1,13]. Materialized views have been proposed by several researchers as a way of restructuring a relational database at the internal level [12,19,20]. Materialized views can also be useful to extend relational database management systems to support procedures [21,23] and rules [22].

Because base relations are modified by update operations, the materialized views defined on them may have to be updated as well. The simplest way to bring an affected materialized view up to date is by re-evaluating the query that defines it. However, this alternative is often wasteful and the cost involved may be unacceptable.

The problem of updating a materialized view as a result of changes applied to the underlying base relations that participate in its view definition can be decomposed into the following subproblems [3]: (1) detection of irrelevant updates, (2) detection of autonomously computable updates, and (3) efficient re-evaluation of the view.

Irrelevant updates refer to those updates to the base relations that cannot possibly have an effect on the view [2,6,15]. Necessary and sufficient conditions for the detection of irrelevant updates for the case when a view is defined by an expression involving selection, projection, and join are presented by Blakeley et al. [3,4,5].

If an update is not irrelevant to a view, then some data from the base relations may be needed to update the view. An important case to consider, however, is one in which all data needed is contained in the view itself. Updates of this type are called autonomously computable updates. Two types of autonomously computable updates, namely, unconditional and conditional have been distinguished [3,4]. Unconditionally autonomously computable updates refer to those updates to the base relations that can be reflected into the view regardless of the database instance. This
type of update can be detected by examining the view definition and the update to the base relation at compile time. Necessary and sufficient conditions for the detection of unconditionally autonomously computable updates when a view is defined by a PSJ-expression and the updates are insertions, deletions, or modifications are presented in [3,4]. Conditionally autonomously computable updates refer to those updates to the base relations that can be reflected into a view only for those databases that yield the current view materialization. This paper addresses the problem of conditionally autonomously computable updates, in particular, (1) determining the conditions under which an insert or delete operation to a base relation that participates in the definition of a view is conditionally autonomously computable with respect to the current view materialization, and (2) showing how to carry out the insertions or deletions on the view.

It must be stressed that the problem analyzed in this paper is completely different from the problem of updates through views. In that 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 analyzing the traditional problem of deriving appropriate update translations, this paper is concerned with finding efficient ways of keeping materialized views up to date with the base relations. The reader interested in the problem of updates through views may refer to work by Keller [11] or Medeiros and Tompa [16].

The next section presents some basic concepts and notation required for this paper. Section 3 presents a formal definition for autonomously computable updates. Sections 4 and 5 present our results on conditionally autonomously computable deletions and insertions, respectively. Section 6 discusses cases when conditionally autonomously updates can be performed more efficiently, and finally, Section 7 presents the conclusions of this work.

2 Basic concepts

In the rest of the paper we assume that the reader is familiar with the relational model. In this section, we review some familiar concepts, introduce some less familiar ones, and present the notation used throughout the paper.

We assume there exists a universe $U$ of attributes, and for each $A \in U$, a set of values called the domain of $A$ denoted by $\text{dom}(A)$. A relation scheme $R$ is a finite subset of $U$. A tuple $t$ over scheme $R$ is a mapping from $R$ into $\bigcup_{A \in R} \text{dom}(A)$ such that $t(A) \in \text{dom}(A)$. A relation $r$ over scheme $R$ is a finite set of tuples over $R$. We assume no constraints (e.g., keys or functional dependencies) to be imposed on the relations allowed. For a set $X \subseteq R$ and a tuple $t \in r$ over scheme $R$, we denote the restriction of $t$ to $X$ by $t[X]$. A database schema $D$ is a set of relation schemes. A database $d$
over a database scheme \( D \) is a set of relations containing one relation for each relation scheme in \( D \). Without loss of generality we assume that the relation schemes in \( D \) are non-intersecting, so that attribute names are unique; this avoids having to refer to attributes in \( D \) by specifying both a relation name and an attribute name.

Let \( r_1 \) and \( r_2 \) be relations over schemes \( R_1 \) and \( R_2 \), respectively.

1. The Cartesian product of \( r_1 \) and \( r_2 \), denoted by \( r_1 \times r_2 \), is a relation \( \{ t[R_1 \cup R_2] \mid t[R_1] \in r_1 \text{ and } t[R_2] \in r_2 \} \) over scheme \( R_1 \cup R_2 \).

2. The projection of \( r_1 \) over \( X \subseteq R_1 \), denoted by \( \pi_X(r_1) \), is the relation \( \{ t[X] \mid t \in r_1 \} \) over scheme \( X \).

3. The selection of tuples in \( r_1 \) satisfying predicate \( C \), written \( \sigma_C(r_1) \), is the relation \( \{ t[R_1] \mid t \in r_1 \text{ and } t \text{ satisfies predicate } C \} \) over scheme \( R_1 \). Later in this section we will discuss in more detail the type of predicates we allow.

4. The join of \( r_1 \) and \( r_2 \), denoted by \( r_1 \Join_P r_2 \), where \( P \) is a predicate on \( R_1 \cup R_2 \), is a relation \( \{ t[R_1 \cup R_2] \mid t \in r_1 \times r_2 \text{ and } t \text{ satisfies predicate } P \} \).

A view definition \( E \) is a relational algebra expression over some subset of \( D \). A view materialization (or just view) \( v(E, d) \) is a relation resulting from the evaluation of the relational algebra expression \( E \) against the database \( d \). In this paper we consider only views defined by relational algebra expressions called \( PSJ \)-expressions, formed from a combination of projections, selections, and joins. Every \( PSJ \)-expression can be transformed into an equivalent expression in a standard form consisting of a Cartesian product, followed by a selection, followed by a projection. From this it follows that any \( PSJ \)-expression can be written in the form \( E = \pi_A(\sigma_C(R_1 \times R_2 \times \cdots \times R_m)) \). We can therefore represent any \( PSJ \)-expression by a triple \( E = (A, R, C) \), where \( A = \{A_1, A_2, \ldots, A_k\} \) is called the attribute set, \( R = \{R_1, R_2, \ldots, R_m\} \) is the relation set or base, and \( C \) is a Boolean expression called the selection condition which comprises the predicates of all the select and join operations of the original relational algebra expression defining \( E \). The attributes in \( A \) will often be referred to as the visible attributes of the view.

We assume that for \( E = (A, R, C) \), each relation in \( R \) is referenced only once in the associated \( PSJ \)-expression, that is, the expression contains no self-joins. Also, we use the notation:

- \( \alpha(C) \) the set of all attributes appearing in selection condition \( C \)
- \( \alpha(R) \) the set of all attributes of relation scheme \( R \)

\(^1\) Note that so far we have been referring to a subset from \( U \) as \( R \), however, we introduce the notation \( \alpha(R) \) to make a clear distinction between the subset of attributes and the name of the subset.
2  BASIC CONCEPTS

\( \alpha(R) \) the set of all attributes mentioned in the set of relation schemes \( R \), that is, \( \cup_{R_i \in R} \alpha(R_i) \).

The following example illustrates the above notation.

**Example 2.1** Consider three relation schemes \( R_1(A_1, B_1) \), \( R_2(A_2, B_2, C_2) \), \( R_3(A_3, B_3) \), and a view defined by the expression

\[
E = \pi_{A_1A_2B_3}(\sigma_{A_1>20}(R_1)) \bowtie_{B_1=B_2} \pi_{A_2B_3}(\sigma_{B_2=30}(R_2)) \bowtie_{A_2=A_3} \sigma_{B_3<10}(R_3)).
\]

This relational algebra expression can be converted into the equivalent expression

\[
E' = \pi_{A_1A_2B_3}(\sigma_{(A_1>20)(B_1=B_2)(B_2=30)(A_2=A_3)(B_3<10)}(R_1 \times R_2 \times R_3)),
\]

which in turn can be represented using the triple notation as

\[
E' = \{A_1, A_2, B_3\}, \{R_1, R_2, R_3\}, (A_1 > 20)(B_1 = B_2)(B_2 = 30)(A_2 > A_3)(B_3 < 10),
\]

In this example, \( \alpha(C) = \{A_1, B_1, A_2, B_2, A_3, B_3\} \), \( \alpha(R_1) = \{A_1, B_1\} \), \( \alpha(R_2) = \{A_2, B_2, C_2\} \), \( \alpha(R_3) = \{A_3, B_3\} \), and \( \alpha(R) = \{A_1, B_1, A_2, B_2, C_2, A_3, B_3\} \).

The update operations considered are insertions and deletions. Following relational languages such as SQL [7] and QUEL [24], each update operation may change only one base relation; multiple updates (possibly involving changes to several relations) can be packaged as transactions, but this has no effect on our considerations. The following notation will be used to describe the update operations:

**INSERT** \((Ru, T)\): Insert into relation \( r_u \) the set of tuples \( T \), where each \( t \in T \) is defined over \( R_u \).

**DELETE** \((Ru, C_D)\): Delete from relation \( r_u \) all tuples satisfying the selection condition \( C_D \) defined on scheme \( R_u \). Notice that by allowing this form of delete operation we automatically cover the form where a set of tuples to be deleted is explicitly presented.

We assume that all attributes have discrete and finite domains. Any such domain can be mapped onto an interval of integers, and therefore for simplicity we will treat all attributes as being defined over some interval of integers. For Boolean expressions, the logical connectives will be denoted by "\( \lor \)" for OR, "\( \land \)" for AND, "\( \neg \)" for NOT, "\( \implies \)" for implication, and "\( \iff \)" for equivalence. To indicate that all variables of a selection condition \( C \), are universally quantified, we write \( \forall (C) \), omitting the names of the variables, and similarly for existential quantification. If we need to identify explicitly which variables are quantified, we write \( \forall X (C) \), where \( X \) is a set of variables.

An *evaluation* of a selection condition is obtained by replacing all the variable names (attribute names) by values from the corresponding domains. The result is either *true* or *false*. A *partial
evaluation (or substitution) of a selection condition is obtained by replacing some of its variables by values from the corresponding domains. Let \( C \) be a selection condition and \( t \) a tuple over some set of attributes. The partial evaluation of \( C \) with respect to \( t \) involves the replacement of variables in \( C \) by values for the corresponding attributes in \( t \) and is denoted by \( C[t] \). The result of a partial substitution is a new condition with fewer variables.

**Example 2.2** Consider the selection condition of the previous example

\[
C = (A_1 > 20)(B_1 = B_2)(B_2 = 30)(A_2 > A_3)(B_3 < 10)
\]

and let \( t = (45, 30, 18) \) be a tuple in \( \tau_2 \) on scheme \( R_2 \). The partial substitution \( C[t] \) is given by the condition \((A_1 > 20)(B_1 = 30)(30 = 30)(45 > A_3)(B_3 < 10)\) which can be simplified to \((A_1 > 20)(B_1 = 30)(45 > A_3)(B_3 < 10)\). The evaluation of \( C \) on the values \( A_1 = 25, B_1 = 45, A_3 = 20, \) and \( B_3 = 8 \) yields the truth value \( \text{false} \).

In principle, the Boolean expressions allowed to be selection conditions are sentences in first-order predicate calculus and in fact our lemmas and theorems impose no restrictions on the Boolean expressions allowed. However, for efficient implementation we limit those Boolean expressions to conjunctive expressions as explained below.

Detecting whether an update operation is autonomously computable involves testing whether certain Boolean expressions are valid (i.e., tautologies), or equivalently, whether their complements are unsatisfiable. Proving the satisfiability of Boolean expressions is, in general, \( NP \)-complete [8]. However, for a restricted subclass of Boolean expressions, polynomial algorithms exist. Rosenkrantz and Hunt [18] developed such an algorithm for conjunctive Boolean expressions. Each expression \( \beta \) must be of the form \( \beta = \beta_1 \land \beta_2 \land \cdots \land \beta_k \), where each \( \beta_i \) is an atomic predicate of the form \((x \theta y + c)\) or \((x \theta c)\), where \( \theta \in \{=, <, \leq, >, \geq \} \), \( x \) and \( y \) are variables, and \( c \) is a constant. Variables and constants are assumed to range over the integers. The improved efficiency arises from not allowing \( \theta \) to be the operator \( \neq \). A modified version of the algorithm by Rosenkrantz and Hunt for the case when each variable ranges over a finite interval of integers has been developed. For full details of the modified algorithm the reader is referred to Blakeley et al. [4].

An expression not in conjunctive form can be handled by first converting it into disjunctive normal form and then testing each disjunct separately. Some theorems and lemmas in this paper require testing the validity of expressions of the form \( C_1 \Rightarrow C_2 \). The implication can be eliminated by converting to the form \((-C_1) \lor C_2 \). Similarly, expressions of the form \( C_1 \Leftrightarrow C_2 \) can be converted to \((C_1 \land C_2) \lor (-C_1 \land -C_2) \). The \( NP \)-completeness of the satisfiability problem is caused by the fact that converting an expression into disjunctive normal form may, in the worst case, lead to exponential growth in the length of the expression.

**Definition 2.1** [12] Let \( C \) be a Boolean expression over variables \( x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m \). A variable \( y_i, 1 \leq i \leq m \), is said to be uniquely determined by \( x_1, x_2, \ldots, x_n \) and \( C \) if
∀ \( x_1, \ldots, x_n, y_1, \ldots, y_m, y_1', \ldots, y_m' \)

\[ C(x_1, \ldots, x_n, y_1', \ldots, y_m') \land C(x_1, \ldots, x_n, y_1, \ldots, y_m) \Rightarrow (y_i = y_i'). \]

If a variable \( y_i \) (or a subset of the variables \( y_1, y_2, \ldots, y_m \)) is uniquely determined by a condition \( C \) and the variables \( x_1, \ldots, x_n \), then given any tuple \( t = (x_1, \ldots, x_n) \) where the extended tuple \( (x_1, \ldots, x_n, y_1, \ldots, y_m) \) is known to satisfy \( C \), the missing value of the variable \( y_i \) can be correctly reconstructed. How to reconstruct the values of uniquely determined variables was shown by Larson and Yang [12]. If the variable \( y_i \) is not uniquely determined, then we cannot guarantee that its value is reconstructible for every tuple. However, it may still be reconstructible for some tuples.

**Example 2.3** Let \( C \equiv (A_1 = A_2)(A_1 > 7)(B_2 = 5) \). It is easy to prove that \( A_2 \) and \( B_2 \) are uniquely determined by \( A_1 \) and the condition \( C \). For example, if we are given a tuple that satisfies \( C \) but only the value of \( A_1 \) is known, then we can immediately determine that the values of \( A_2 \) and \( B_2 \). If \( A_1 = 10 \), then \( A_2 \) and \( B_2 \) must be 10 and 5, respectively.

**Definition 2.2** [12] Let \( E = (A, R, C) \) be a view and let \( A_E \) be the set of all attributes in \( \alpha(R) \) that are uniquely determined by the attributes in \( A \) and the condition \( C \). Then \( A^+ = A \cup A_E \) is called the extended attribute set of \( E \).

It is proved by Larson and Yang [12] that \( A^+ \) is the maximum set of attributes for which values can be reconstructed for every tuple of \( E \).

## 3 Autonomous Computationally Updateable

To illustrate autonomously computable updates let us look at the following example.

**Example 3.1** Consider two relation schemes \( R_1(A_1, B_1) \) and \( R_2(A_2, B_2, C_2) \) with relations:

<table>
<thead>
<tr>
<th>( A_1 )</th>
<th>( B_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( A_2 )</th>
<th>( B_2 )</th>
<th>( C_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>26</td>
</tr>
<tr>
<td>17</td>
<td>20</td>
<td>31</td>
</tr>
</tbody>
</table>

Let a view be defined by the expression \( E = (\{A_1, B_1, C_2\}, \{R_1, R_2\}, (B_1 = B_2)) \) with view materialization:
$v(E, \hat{d}) : A_1 \quad B_1 \quad C_2$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>31</td>
</tr>
</tbody>
</table>

Because the view is defined on more than one base relation we cannot guarantee that any insert operation to either $r_1$ or $r_2$ will be autonomously computable on materializations of $E$ for every possible database $d$ [4]. However, if the database is changed by the operation INSERT $(R_1, \{(4, 10)\})$, then the current materialization of the view contains enough information to determine that the tuples $(4, 10, 25)$ and $(4, 10, 26)$ are the tuples that must be inserted into the view to bring it up to date; that is, the particular insert operation is autonomously computable for all databases $d$ for which $v(E, d) = v(E, \hat{d})$. $\square$

The next definition formalizes the notions of unconditionally and conditionally autonomously computable updates.

**Definition 3.1** Consider a view definition $E$ and an update operation $\mathcal{U}$, both defined over the relation schemes $D$. Let $d$ denote a database on $D$ before applying $\mathcal{U}$ and $d'$ the corresponding database after applying $\mathcal{U}$.

- The effect of the operation $\mathcal{U}$ on materializations of the view defined by $E$ is said to be **unconditionally autonomously computable** if there exists a function $F_{\mathcal{U}, E}(v(E, d))$ such that

$$\forall \quad d \quad v(E, d') = F_{\mathcal{U}, E}(v(E, d)).$$

- The effect of the operation $\mathcal{U}$ on a particular materialization $v(E, \hat{d})$ of the view defined by $E$ on database $\hat{d}$ defined on $D$ is said to be **conditionally autonomously computable with respect to** $v(E, \hat{d})$ if there exists a function $F_{\mathcal{U}, E}(v(E, \hat{d}))$ such that

$$\forall \quad d \quad \text{such that} \quad v(E, d) = v(E, \hat{d}) \quad v(E, d') = F_{\mathcal{U}, E}(v(E, \hat{d})).$$

$\square$

The main aspect of Definition 3.1 is the requirement that $F_{\mathcal{U}, E}$ be a function of the materialization $v(E, d)$. That is, if $d_1$ and $d_2$ are databases such that $v(E, d_1) = v(E, d_2)$, then it must follow that $F_{\mathcal{U}, E}(v(E, d_1)) = F_{\mathcal{U}, E}(v(E, d_2))$. The following simple but important lemma is a slight variation of a lemma introduced by Blakeley, Coburn, and Larson [4], adapted to the definition of conditionally autonomously computable updates. It will be used in several proofs throughout Sections 4 and 5.
Lemma 3.1 Consider a view defined by \( E = (A, R, C) \) with materialization \( v(E, \hat{d}) \), and update operation \( \mathcal{U} \) to relation \( r \) on scheme \( R \in \mathbb{R}, R \subseteq D \). Let \( d_1, d_2 \) be two databases and \( d'_1, d'_2 \) be the updated databases after applying \( \mathcal{U} \), respectively. If \( v(E, d_1) = v(E, d_2) = v(E, \hat{d}) \) and \( v(E, d'_1) \neq v(E, d'_2) \), then \( \mathcal{U} \) is not conditionally autonomously computable.

Proof: Assume there exists a function \( F_{\mathcal{U}, E} \) as in Definition 3.1 such that \( v(E, d') = F_{\mathcal{U}, E}(v(E, d)) \) for every database \( d \) such that \( v(E, d) = v(E, \hat{d}) \). Consider the databases \( d_1, d_2 \) such that \( v(E, d_1) = v(E, d_2) = v(E, \hat{d}) \). It follows that \( F_{\mathcal{U}, E}(v(E, d_1)) = v(E, d'_1) \) and \( F_{\mathcal{U}, E}(v(E, d_2)) = v(E, d'_2) \). Since \( F_{\mathcal{U}, E} \) is a function and \( v(E, d_1) = v(E, d_2) \) it follows, from the definition of function, that \( F_{\mathcal{U}, E}(v(E, d_1)) = F_{\mathcal{U}, E}(v(E, d_2)) \), or equivalently, that \( v(E, d'_1) = v(E, d'_2) \). This contradicts the condition given and proves the lemma.

In the next two sections we present necessary and sufficient conditions for determining when an insertion or deletion is conditionally autonomously computable. We address the problem of deletions first, because it is simpler.

4 Conditionally autonomously computable deletions

Unconditionally autonomously computable deletions require that all variables in the conditions \( C_D \) and \( C \) which are not seen in the view, be "computationally nonessential in \( C_D \) with respect to \( C \)" [3, 4], that is, for all possible values assigned to those variables in such a way that they satisfy the condition \( C \), these values will make the condition \( C_D \) evaluate to the same truth value.

Occasionally, even though a variable in \( (\alpha(C_D) \cup \alpha(C)) \setminus A^+ \) is computationally essential in \( C_D \) with respect to \( C \), it may still be possible to compute the delete operation autonomously. The following example illustrates this situation.

Example 4.1 Consider the relation schemes \( R_1(A_1, B_1) \) and \( R_2(A_2, B_2, C_2) \), with corresponding relations

\[
\begin{array}{cccc}
    r_1: & A_1 & B_1 \\
         & 16 & 1 \\
         & 20 & 2 \\
         & 24 & 3 \\
\end{array}
\quad
\begin{array}{ccc}
    r_2: & A_2 & B_2 & C_2 \\
         & 17 & 18 & 23 \\
         & 19 & 14 & 25 \\
         & 24 & 18 & 33 \\
\end{array}
\]

Let a view be defined by \( E = (\{B_1, A_2, B_2, C_2\}, \{R_1, R_2\}, (A_1 > A_2)(A_1 < C_2)) \), and thus its corresponding materialization is
\( v(E, d) : \quad B_1 \quad A_2 \quad B_2 \quad C_2 \)

\[
\begin{array}{cccc}
2 & 17 & 18 & 23 \\
2 & 19 & 14 & 25 \\
3 & 19 & 14 & 25 \\
\end{array}
\]

Now consider a delete operation \( \text{DELETE} (R_1, (A_1 < 20) \lor (A_1 > 22)) \). Even though \( A_1 \) is computationally essential in \( C_D \equiv (A_1 < 20) \lor (A_1 > 22) \) with respect to \( C \equiv (A_1 > A_2)(A_1 < C_2) \), we can still determine from the tuples present in the view as well as the absence of tuple \((3, 17, 18, 23)\) that the third tuple in \( v(E, d) \) is precisely the one that needs to be deleted.

For the effect of a delete operation to be conditionally autonomously computable with respect to \( v(E, d) \) we need to be able to prove, for each of the tuples currently stored in \( v(E, d) \), whether the tuple stays in the view as a result of the delete operation on a base relation. If there is at least one tuple in the view for which we cannot decide whether it will stay in the view after the delete operation, then the effect of the delete on the view is not conditionally autonomously computable and hence the view should be updated using some other mechanism such as differential re-evaluation \([5,10]\). We need to be able to prove the above regardless of the values for variables \( x \in (\alpha(C_D) \cup \alpha(C)) - A^+ \), using only the information provided by the current materialization \( v(E, d) \), the view definition \( E = (A, R, C) \), and the operation \( \text{DELETE} (R_u, C_D) \).

We now define notation for the contents of the view materialization that will allow us to establish the relationship between the values for attributes that are visible in the view and the values for attributes not visible in the view.

**Definition 4.1** Let \( V \) denote the \( Z \)-extended view of \( v(E, d) \) where each tuple \( e \in v(E, d) \) is padded with a distinct variable name \( z_k \) for each attribute in \( Z = (\alpha(C_D) \cup \alpha(C)) - A^+ \), \( 1 \leq k \leq |Z| \cdot |v(E, d)| \), as well as with values for uniquely determined variables (i.e., variables in \( A^+ - A \)).

**Example 4.2** Consider again Example 4.1, and assume that the delete operation

\( \text{DELETE} (R_1, (A_1 < 20) \lor (A_1 > 22)) \)

is performed on relation \( r_1 \). The \( Z \)-extended view \( V \) of \( v(E, d) \) is given by:

\( V : \quad A_1 \quad B_1 \quad A_2 \quad B_2 \quad C_2 \)

\[
\begin{array}{cccc}
z_1 & 2 & 17 & 18 & 23 \\
z_2 & 2 & 19 & 14 & 25 \\
z_3 & 3 & 19 & 14 & 25 \\
\end{array}
\]
To explore all possible relationships between the variables in $Z$ and the values that show up in $v(E, d)$ we need the notion of a project-join mapping [14].

**Definition 4.2** The project-join mapping $m_{V}$ of the $Z$-extended view $V$ is defined as:

$$m_{V} = \pi_{G_1}(V) \times \pi_{G_2}(V) \times \cdots \times \pi_{G_m}(V),$$

where $G_i = (\alpha(C_D) \cup \alpha(C) \cup A^+) \cap \alpha(R_i)$, $1 \leq i \leq m$. In other words, $G_i$ represents the attributes from $R_i$ which are either visible or used in the conditions $C_D$ or $C$.

**Example 4.3** The project-join mapping for the $Z$-extended view $V$ in the above example is:

$$m_{V} : A_1 \quad B_1 \quad A_2 \quad B_2 \quad C_2$$

<table>
<thead>
<tr>
<th></th>
<th>$x_1$</th>
<th>2</th>
<th>17</th>
<th>18</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x_2$</td>
<td>2</td>
<td>19</td>
<td>14</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>$x_2$</td>
<td>2</td>
<td>17</td>
<td>18</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>$x_2$</td>
<td>2</td>
<td>19</td>
<td>14</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>$x_3$</td>
<td>3</td>
<td>17</td>
<td>18</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>$x_3$</td>
<td>3</td>
<td>19</td>
<td>14</td>
<td>25</td>
</tr>
</tbody>
</table>

Observe that each tuple from $m_{V}$ represents some knowledge about possible values from the database for the attributes in $Z$. For instance, the tuple $(x_1, 2, 17, 18, 23)$ represents the tuple from $r_1$ of the form $(z_1, 2)$ which when combined with the tuple $(17, 18, 23)$ from $r_2$ in such a way that they satisfy the condition $C$, produce the tuple $(2, 17, 18, 23) \in v(E, d)$.

We partition the set of tuples in $m_{V}$ into two sets $P$ and $Q$ defined as follows:

$$P = \{ p \mid p \in m_{V} \text{ and } p[A] \in v(E, d) \}$$

$$Q = \{ q \mid q \in m_{V} \text{ and } q[A] \notin v(E, d) \}$$

Clearly, $0 \leq |P|, |Q| \leq |v(E, d)|^m$, where $m$ is the number of relations involved in the view definition. Based on the sets $P$ and $Q$ we can build a Boolean expression that captures the knowledge stored in the materialized view about potential values (from the database) for the attributes in $Z$.

**Definition 4.3** The Boolean expression

$$\forall x_1, x_2, \ldots, x_k \bigwedge_{p \in P} C[p] \bigwedge_{q \in Q} \neg C[q]$$

defines all potential values for attributes in $Z$ that are consistent with the current materialization $v(E, d)$. 

\[\square\]
The following lemma establishes the condition under which it is guaranteed that a tuple \( t[A] \in v(E, d) \), \( t \in \mathcal{V} \), stays in the view after the delete operation.

**Lemma D.1** Consider the current materialization \( v(E, d) \) of a view \( E = (A, R, C) \), \( R = \{R_1, R_2, \ldots, R_m\} \), and the update operation DELETE \( (R_u, C_D) \). Let \( \mathcal{V} \) be the extended materialization corresponding to \( v(E, d) \). The tuple \( t[A] \in v(E, d) \), \( t \in \mathcal{V} \), is guaranteed to stay in the updated view \( v(E, d') \) if and only if

\[
\forall z_1, z_2, \ldots, z_k \left( \bigwedge_{p \in P} \neg C[p] \bigwedge_{q \in Q} \neg C[q] \Rightarrow \neg C_D[t] \right),
\]

where \( 1 \leq k \leq |Z| * |v(E, d)| \).

**Proof:** (Sufficiency) If the condition holds, then no matter what values are assigned to the variables \( z_1, z_2, \ldots, z_k \) to make the antecedent true (and hence consistent with the current view materialization \( v(E, d') \)), the tuple \( t[A] \), \( t \in \mathcal{V} \), will not be deleted from \( v(E, d') \).

(Necessity) We need to show that if the condition does not hold, then there exists a database \( d_1 \) that yields the view \( v(E, d_1) = v(E, d) \) for which the tuple \( t[A] \) should be deleted. Assume there exists an assignment of values to variables \( z_1, z_2, \ldots, z_k \) such that

\[
\bigwedge_{p \in P} C[p](\hat{z}_1, \hat{z}_2, \ldots, \hat{z}_k) \bigwedge_{q \in Q} \neg C[q](\hat{z}_1, \hat{z}_2, \ldots, \hat{z}_k) \Rightarrow \neg C_D[t](\hat{z}_1, \hat{z}_2, \ldots, \hat{z}_k)
\]

evaluates to \( false \), that is, the antecedent is \( true \) and the consequent \( false \). To construct the database \( d_1 \) we use the project-join mapping \( m_\mathcal{V} \). The database \( d_1 \) consists of the relations \( r_i = \pi_{G_i}(m_\mathcal{V}) \), where \( G_i = (\alpha(C_D) \cup \alpha(C) \cup A^+) \cap \alpha(R_i) \) and \( 1 \leq i \leq m \). Clearly, \( v(E, d_1) = v(E, d) \). If we now apply the update to relation \( r_u \) to obtain the updated database \( d'_1 \), then the tuple \( t[A] \), \( t \in \mathcal{V} \), should be deleted from \( v(E, d') \). \( \square \)

The following lemma establishes the condition under which we can guarantee that a tuple \( t[A] \in v(E, d) \), \( t \in \mathcal{V} \), must be deleted from the view after the delete operation.

**Lemma D.2** Consider the current materialization \( v(E, d) \) of a view \( E = (A, R, C) \), \( R = \{R_1, R_2, \ldots, R_m\} \), and the update operation DELETE \( (R_u, C_D) \). Let \( \mathcal{V} \) be the extended materialization corresponding to \( v(E, d) \). The tuple \( t[A] \in v(E, d) \), \( t \in \mathcal{V} \), is guaranteed to be deleted from the view \( v(E, d) \) if and only if

\[
\forall z_1, z_2, \ldots, z_k \left( \bigwedge_{p \in P} \neg C[p] \bigwedge_{q \in Q} \neg C[q] \Rightarrow C_D[t] \right),
\]

where \( 1 \leq k \leq |Z| * |v(E, d)| \).

**Proof:** (Sufficiency) If the condition holds, then no matter what values are assigned to the variables \( z_1, z_2, \ldots, z_k \) to make the antecedent true (and hence consistent with the current view materialization \( v(E, d) \)), the tuple \( t[A] \), \( t \in \mathcal{V} \), cannot belong into the view.
(Necessity) If the condition does not hold, then there exists a database $d_1$ constructed in a similar way as in the proof of Lemma D.1 that yields the view $v(E, d_1) = v(E, d)$ for which the tuple $t[A]$ should not be deleted.

\[\square\]

**Theorem D.3** If a delete operation is conditionally autonomously computable, then every tuple $t \in V$ satisfies either the condition of Lemma D.1 or the condition of Lemma D.2.

**Proof:** Consider the contrapositive: if there exists a tuple $t \in V$ for which the conditions of Lemmas D.1 and D.2 do not hold, then the effect of the delete operation cannot be conditionally autonomously computable.

In the same way as in the proofs of Lemmas D.1 and D.2, we can construct two databases $d_1$ and $d_2$ such that $v(E, d_1) = v(E, d_2) = v(E, d)$, and such that the delete operation on $r_u$ of $d_1$ will cause $t[A]$ to be deleted, and the same operation on $d_2$ will cause $t[A]$ to stay. Consequently, there cannot exist a function which based on the current view materialization will compute the correct updated view. Thus, by Lemma 3.1 the update cannot be conditionally autonomously computable.

\[\square\]

The following example illustrates Theorem D.3.

**Example 4.4** Consider again the view and delete operation given by Examples 4.1 and 4.2, respectively, with $Z$-extended view and project-join mapping given by:

\[
\begin{array}{cccccc}
V & A_1 & B_1 & A_2 & B_2 & C_2 \\
2 & 2 & 17 & 18 & 23 \\
2 & 2 & 19 & 14 & 25 \\
3 & 3 & 19 & 14 & 25 \\
\hline
\end{array}
\]

\[
\begin{array}{cccccc}
mv & A_1 & B_1 & A_2 & B_2 & C_2 \\
1 & 2 & 17 & 18 & 23 & p_1 \\
1 & 2 & 19 & 14 & 25 & p_2 \\
2 & 2 & 17 & 18 & 23 & p_3 \\
2 & 2 & 19 & 14 & 25 & p_4 \\
3 & 3 & 17 & 18 & 23 & q_1 \\
3 & 3 & 19 & 14 & 25 & p_6 \\
\hline
\end{array}
\]

For the first tuple $t_1 = (z_1, 2, 17, 18, 23) \in V$ we find that the condition of Lemma D.1, given below, holds.

\[
\begin{align*}
\forall z_1 \\
((z_1 > 17)(z_1 < 23) & \quad \text{(from } p_1) \\
\land (z_1 > 19)(z_1 < 25) & \quad \text{(from } p_2) \\
\land (z_1 < 20) & \quad \text{(from } p_3) \\
\land (z_1 > 22) & \quad \text{(from } p_4) \\
\end{align*}
\]

\[
\Rightarrow \neg((z_1 < 20) \lor (z_1 > 22))]
\]

holds, thus tuple $t_1[A]$ should not be deleted from the view. For the second tuple $t_2 = (z_2, 2, 19, 14, 25) \in V$ we find that the condition of Lemma D.1, given below, also holds.

\[
\begin{align*}
\forall z_2 \\
((z_2 > 17)(z_2 < 23) & \quad \text{(from } p_3) \\
\land (z_2 > 19)(z_2 < 25) & \quad \text{(from } p_4) \\
\land (z_2 > 20) & \quad \text{(from } p_5) \\
\end{align*}
\]

\[
\Rightarrow \neg((z_2 < 20) \lor (z_2 > 22))]
\]
Thus tuple \( t_2[A] \) should not be deleted from the view. Finally, for the third tuple \( t_3 = (z_3, 3, 19, 14, 25) \in \mathcal{V} \) we find that the condition of Lemma D.2, given below, holds.

\[
\begin{align*}
\forall z_3 \\
[(z_3 > 19)(z_3 < 25)] & \quad \text{(from } p_5) \\
\land \neg[(z_3 > 17)(z_3 < 23)] & \quad \text{(from } q_1) \\
\Rightarrow (z_3 < 20) \lor (z_3 > 22)]
\end{align*}
\]

Thus tuple \( t_3[A] \) should be deleted from the view. \( \square \)

5 Conditionally autonomously computable insertions

Consider the operation INSERT \((R_u, T)\), where \( T \) is a set of tuples to be inserted into relation \( R_u \). Let a view be defined by the triple \( E = (A, R, C) \), \( R_u \in R \), with materialization \( v(E, d) \). The effect of the insert operation on \( v(E, d) \) is conditionally autonomously computable if and only if

A. for each tuple \( t_u \in T \), we can build the new tuples to be inserted into the view. In this step, the new tuples are assembled using the inserted tuple \( t_u \) and the tuples already present in the view \( v(E, d) \); and

B. we can prove that the new tuples generated in the previous stage represent all the tuples that need to be inserted into \( v(E, d) \) for the set of databases, including the current database \( d \), that yield this view.

Example 5.1 Consider two relation schemes \( R_1(A_1, B_1) \) and \( R_2(A_2, B_2, C_2) \) with relations:

\[
\begin{array}{ccc}
| A_1 | B_1 | & | A_2 | B_2 | C_2 | \\
1 & 10 & | 5 & 10 & 25 | \\
2 & 20 & | 6 & 10 & 26 | \\
3 & 30 & | 17 & 20 & 31 |
\end{array}
\]

Let a view be defined by the expression \( E = (\{A_1, B_1, C_2\}, \{R_1, R_2\}, (B_1 = B_2)) \), which results in the materialization:

\[
\begin{array}{ccc}
| A_1 | B_1 | C_2 | \\
1 & 10 & 25 | \\
1 & 10 & 26 | \\
2 & 20 & 31 |
\end{array}
\]
If relation $r_1$ is updated by the operation INSERT ($R_1, \{(4, 10)\}$), then the tuples $(4, 10, 25)$ and $(4, 10, 26)$ will have to be inserted into $v(E, d)$ to bring it up to date. In contrast to the results for unconditionally autonomously computable insertions [3,4], now several tuples may be inserted into the view. This is possible because for conditionally autonomously computable updates we are allowed to use the contents of the view in deciding whether the update is autonomously computable. □

This section is divided into two parts. The first part deals with the problem of building the new tuples to be inserted into $v(E, d)$ as a result of inserting the tuple $t_u$ into $r_u$ (see part A above). The second part deals with the problem of verifying that the tuples built in the previous stage are all tuples that need to be inserted (part B above). Both parts are performed using only the information provided by the newly inserted tuple $t_u$, the view definition $E$, and the materialization $v(E, d)$.

### 5.1 Building the new tuples

The process of building the new tuples to be inserted into the view is similar to the process used to handle deletions. The notions of $Z$-extended view $V$ of $v(E, d)$, and of the project-join mapping $m_V$ are used here in the same way as for deletions with the exception that they are defined over the set of relevant attributes defined by $Z = \alpha(C) - A^+$ (rather than by $Z = (\alpha(C_D) \cup \alpha(C)) - A^+$).

**Definition 5.1** Let $V$ denote the $Z$-extended view of $v(E, d)$ where each tuple $e \in v(E, d)$ is padded with a distinct variable name $z_k$ for each attribute in $Z = \alpha(C) - A^+, 1 \leq k \leq \lvert Z \rvert \ast |v(E, d)|$. □

**Definition 5.2** The project-join mapping $m_V$ of the $Z$-extended view $V$ is defined as:

$$m_V = \pi_{G_1}(V) \times \pi_{G_2}(V) \times \cdots \times \pi_{G_m}(V),$$

where $G_i = (\alpha(C) \cup A^+) \cap \alpha(R_i), 1 \leq i \leq m$. In other words, $G_i$ represents the attributes from scheme $R_i$ which are either visible or used in the condition $C$. □

**Definition 5.3** The set of candidate tuples $T_u$ to be inserted into the view as a result of inserting the tuple $t_u$ into $r_u$ is given by

$$T_u = \pi_{G_1}(V) \times \pi_{G_2}(V) \times \cdots \times \pi_{G_{u-1}}(V) \times \{t_u\} \times \pi_{G_{u+1}}(V) \times \cdots \times \pi_{G_m}(V).$$

The set $T_u$ represents all new tuples that can be built using the information provided by the newly inserted tuple $t_u$ and the current view materialization $v(E, d)$. □
Example 5.2 Consider three relation schemes $R_1(A_1, B_1)$, $R_2(A_2, B_2, C_2)$, and $R_3(A_3, B_3)$, and a view $E = \{\{A_1, B_1, A_2, C_2, B_3\}, \{R_1, R_2, R_3\}, \{B_1 > B_2\}(A_2 = A_3)\}$ with materialization:

$$
\begin{array}{cccccc}
E, d & A_1 & B_1 & A_2 & C_2 & B_3 \\
2 & 22 & 30 & 15 & 16 \\
3 & 26 & 42 & 25 & 19
\end{array}
$$

Since $Z = \alpha(C) - A^+ = \{B_1, B_2, A_2, A_3\} - \{A_1, B_1, A_2, C_2, A_3, B_3\} = \{B_2\}$ the $Z$-extended view $V$ of $v(E, d)$ and the project-join mapping $m_V$ are given by:

$$
\begin{array}{cccccc}
V & A_1 & B_1 & A_2 & B_2 & C_2 & A_3 & B_3 \\
2 & 22 & 30 & z_1 & 15 & 30 & 16 \\
3 & 26 & 42 & z_2 & 25 & 42 & 19
\end{array}
\quad
\begin{array}{cccccc}
m_V & A_1 & B_1 & A_2 & B_2 & C_2 & A_3 & B_3 \\
2 & 22 & 30 & z_1 & 15 & 30 & 16 & p_1 \\
2 & 22 & 30 & z_1 & 15 & 42 & 19 & q_1 \\
2 & 22 & 42 & z_2 & 25 & 30 & 16 & q_2 \\
2 & 22 & 42 & z_2 & 25 & 42 & 19 & q_3 \\
3 & 26 & 30 & z_1 & 15 & 30 & 16 & q_4 \\
3 & 26 & 30 & z_1 & 15 & 42 & 19 & q_5 \\
3 & 26 & 42 & z_2 & 25 & 30 & 16 & q_6 \\
3 & 26 & 42 & z_2 & 25 & 42 & 19 & p_2
\end{array}
$$

If the tuple $t_u = (30, 25, 28)$ is inserted into $r_2$, then the set $T_u$ is given by:

$$
T_u = \pi_{A_1, B_1}(V) \times \{t_u\} \times \pi_{A_3, B_3}(V) = \{(2, 22, 30, 25, 28, 30, 16), (2, 22, 30, 25, 28, 42, 19), (3, 26, 30, 25, 28, 30, 16), (3, 26, 30, 25, 28, 42, 19)\}.
$$

For $t = (2, 22, 30, 25, 28, 30, 16)$ to influence the view, for example, we must be able to show that $t[A] = (2, 22, 30, 28, 16)$ will be in $v(E, d')$.

We now present the conditions that allow us to prove, for each tuple $t \in T_u$, whether it belongs to the updated view $v(E, d')$.

Lemma I.1 Consider a view defined by $E = \{A, R, C\}$, $R = \{R_1, R_2, \ldots, R_m\}$, and the operation INSERT $(R_u, \{t_u\})$, $R_u \in R$. Let $v(E, d)$ denote the view materialization before the insert operation is performed. Also let $V$, $m_V$, and $T_u$ denote the extended view of $v(E, d)$, project-join mapping, and set of candidate tuples, respectively. A tuple $t[A], t \in T_u$, is guaranteed to belong to the updated view materialization $v(E, d')$ if and only if

$$
\forall z_1, z_2, \ldots, z_k \left( \bigwedge_{P \in P} C[p] \bigwedge_{Q \in Q} \neg C[q] \Rightarrow C[t] \right),
$$

where $Z = \alpha(C) - A^+$, $1 \leq k \leq |Z| \ast |v(E, d)|$, and $P = \{p \mid p \in m_V$ and $p[A] \in v(E, d)\}$ and $Q = \{q \mid q \in m_V$ and $q[A] \not\in v(E, d)\}$. 

\qed
Proof: (Sufficiency) Similar to the proof for Lemma D.1. □

Lemma I.2 Consider a view defined by \( E = (A, R, C), R = \{R_1, R_2, \ldots, R_m\} \), and the operation \( \text{INSERT}(R_u, \{t_u\}), R_u \in R \). Let \( v(E, d) \) denote the view materialization before the insert operation is performed. Also let \( V, m_V, \) and \( T_u \) denote the extended view of \( v(E, d) \), project-join mapping, and set of candidate tuples, respectively. A tuple \( t[A], t \in T_u \), is guaranteed not to belong to the updated view materialization \( v(E, d') \) if and only if

\[
\forall z_1, z_2, \ldots, z_k \left( \bigwedge_{p \in P} C[p] \bigwedge_{q \in Q} \neg C[q] \Rightarrow \neg C[t] \right),
\]

where \( Z, k, P, \) and \( Q \) are defined as in Lemma I.1.

Proof: (Sufficiency) Similar to the proof of Lemma D.2. □

Theorem I.3 If an insert operation is conditionally autonomously computable, then every tuple \( t \in T_u \) satisfies either the condition of Lemma I.1 or the condition of Lemma I.2.

Proof: Consider the contrapositive: if there exists a tuple \( t \in T_u \) such that \( t \) satisfies neither the condition of Lemma I.1 nor Lemma I.2, then the effect of inserting \( t_u \) into \( \tau_u \) is not conditionally autonomously computable: we can construct two databases \( d_1 \) and \( d_2 \) such that \( v(E, d_1) = v(E, d_2) = v(E, d) \), and such that the insertion of \( t_u \) into \( \tau_u \) of \( d_1 \) will cause \( t[A] \) not to be inserted, but insertion of \( t_u \) into \( \tau_u \) of \( d_2 \) will require \( t[A] \) to be inserted into the view. Consequently, there cannot exist a function which based on the current view materialization will compute the correct updated view. Thus, by Lemma 3.1 the update cannot be conditionally autonomously computable. □

Theorem I.3 is the basis for Algorithm Buildtuples described below.

Algorithm Buildtuples

Input: A view definition \( E = (A, R, C) \) with its corresponding view materialization \( v(E, d) \), and a newly inserted tuple \( t_u \) defined on scheme \( R_u \).

Output: The set of candidate tuples \( \pi_A(T_u) \) or fail if the insertion of \( t_u \) is not conditionally autonomously computable.

1. Compute the sets \( T_u \) and \( m_V \).

2. For each \( t \in T_u \), if the condition of Lemma I.1 holds then keep \( t \) in \( T_u \), else if the condition of Lemma I.2 holds then remove \( t \) from \( T_u \), else return fail.

3. Return \( \pi_A(T_u) \).

\(^2\)The set of tuples \( T_u \) may be reduced by retaining only one copy of the tuples that agree on all attributes \( A^+ \cup \alpha(R_u) \).
Step 2 of the algorithm could be made more efficient by first discarding all tuples $t \in T_u$ such that $C[t]$ evaluates to false. Based on the proofs of Lemmas 1.1 and 1.2, it is clear that the set of tuples $\pi_A(T_u)$ obtained from Algorithm Buildtuples represents a valid subset of the tuples that should be inserted into the view as a result of inserting $t_u$ into $r_u$. For an illustration of the algorithm, refer to Step A in Example 5.4 on page 23 below.

The next subsection deals with the question of how we can make sure that the tuples in $\pi_A(T_u)$ resulting from the above algorithm are all the tuples that must be inserted to bring the view up to date.

5.2 Testing coverage

The rest of this subsection presents a necessary and sufficient condition for testing whether the set of tuples $\pi_A(T_u)$ built by the above algorithm contains all the tuples that must be inserted into the view as a result of inserting the tuple $t_u$ into $r_u$. We must prove that all values for variables in $\alpha(R - \{R_u\}) \cap \alpha(C)$ which can potentially interact with the values from the tuple $t_u$ to produce a new insertion into $v(E, d)$ are somehow contained in the view itself. This follows from the notion of coverage, which is stated in the following definition.

Definition 5.4 Consider a view definition $v = (A, R, C)$ with materialization $v(E, d)$, where $R = \{R_1, R_2, \ldots, R_m\}$, and the update operation UPDATE $(R_u, \{t_u\})$, $R_u \in R$. Let $Y_l = \alpha(R_i) \cap \alpha(C) \cap A^+$ (i.e., the attributes from relation $R_i$ that participate in $C$ and are visible or uniquely determined by the view), and $Z_l = \alpha(R_i) \cap \alpha(C) - A^+$ (i.e., the attributes from relation $R_i$ that participate in $C$ but are not visible nor uniquely determined by the view), $1 \leq l \leq m$, $l \neq u$. The variables $\bigcup_{l=1}^{m} \{Y_l \cup Z_l\}$ are said to be covered in the materialization $v(E, d)$ with respect to the insertion of the tuple $t_u$ into $r_u$ if the following condition holds:

$$\forall Y_1, Z_1, \ldots, Y_{u-1}, Z_{u-1}, Y_{u+1}, Z_{u+1}, \ldots, Y_m, Z_m$$

$$[C[t_u](Y_1, Z_1, \ldots, Y_{u-1}, Z_{u-1}, Y_{u+1}, Z_{u+1}, \ldots, Y_m, Z_m) \Rightarrow$$

$$\{\forall z_1, z_2, \ldots, z_k (\land_{p \in P} C[p] \land_{q \in Q} -C[q] \Rightarrow \lor_{p \in P} C[p \setminus R_1](Y_1, Z_1))\} \land$$

$$\ldots \land$$

$$\{\forall z_1, z_2, \ldots, z_k (\land_{p \in P} C[p] \land_{q \in Q} -C[q] \Rightarrow \lor_{p \in P} C[p \setminus R_{u-1}](Y_{u-1}, Z_{u-1}))\} \land$$

$$\{\forall z_1, z_2, \ldots, z_k (\land_{p \in P} C[p] \land_{q \in Q} -C[q] \Rightarrow \lor_{p \in P} C[p \setminus R_{u+1}](Y_{u+1}, Z_{u+1}))\} \land$$

$$\ldots \land$$

$$\{\forall z_1, z_2, \ldots, z_k (\land_{p \in P} C[p] \land_{q \in Q} -C[q] \Rightarrow \lor_{p \in P} C[p \setminus R_m](Y_m, Z_m))\}$$

where $P, Q$ are defined as before, $C[p \setminus R_l]$ denotes the substitution of values from $p \in P$ for variables in the condition $C$ except for those in scheme $R_l$, $1 \leq l \leq m$, and $z_1, z_2, \ldots, z_k$, $1 \leq k \leq |Z| * |v(E, d)|$. We use $R_l$ instead of $\alpha(R_l)$ in $C[p \setminus R_l]$ to simplify notation. □
5 CONDITIONALLY AUTONOMOUSLY COMPUTABLE INSERTIONS

In other words, the variables $\bigcup_{i=1,i\neq u}^{m} (Y_i \cup Z_i)$ are covered in the materialization $v(E, d)$ with respect to the insertion of the tuple $t_u$ into $r_u$ if every tuple $t_i \in r_i$, $1 \leq i \leq m$ and $i \neq u$, that can possibly combine with $t_u$ to generate a new insertion $t$ into $v(E, d)$ (i.e., $C[t] = \text{true}$, where $t = t_1 \times \ldots \times t_{u-1} \times t_u \times t_{u+1} \times \ldots \times t_m$) is already present in the view. That is, for some $e \in v(E, d)$, $t_i = e[\alpha(R_i)]$, $1 \leq i \leq m$, $i \neq u$. The following example illustrates the above definition.

**Example 5.3** Consider three relation schemes $R_1(A_1, B_1)$, $R_2(A_2, B_2, C_2)$, and $R_3(A_3, B_3)$ with relations $r_1$, $r_2$, and $r_3$, respectively.

<table>
<thead>
<tr>
<th>$r_1$</th>
<th>$A_1$</th>
<th>$B_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$r_2$</th>
<th>$A_2$</th>
<th>$B_2$</th>
<th>$C_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>19</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>22</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>23</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$r_3$</th>
<th>$A_3$</th>
<th>$B_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

i) First consider the view defined by the expression

$$E_1 = \{A_1, B_1, A_2, B_2, C_2\}, \{(R_1, R_2), (B_1 < B_2)\}$$

with materialization $v(E_1, d)$ and project-join mapping $m_{V_1}$ shown below:

<table>
<thead>
<tr>
<th>$v(E_1, d)$</th>
<th>$A_1$</th>
<th>$B_1$</th>
<th>$A_2$</th>
<th>$B_2$</th>
<th>$C_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>15</td>
<td>22</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>50</td>
<td>23</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

$m_{V_1}$:

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$B_1$</th>
<th>$A_2$</th>
<th>$B_2$</th>
<th>$C_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>15</td>
<td>22</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>50</td>
<td>23</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>50</td>
<td>23</td>
<td>30</td>
</tr>
</tbody>
</table>

Only one copy of the tuples $(1, 20, 15, 22, 10)$, $(1, 20, 50, 23, 30)$, and $(2, 22, 50, 23, 30)$ is kept in $m_{V_1}$. Suppose that the update operation INSERT $(R_1, \{(4, 21)\})$ is applied to relation $r_1$. Then we can see that $Y_2 = \{B_2\}$ and $Z_2 = \emptyset$. To be sure that the set of tuples $T_u = \{(4, 21, 15, 22, 10), (4, 21, 50, 23, 30)\}$ are all the tuples that must be inserted into $v(E, d)$ as a result of the insertion of the tuple $(4, 21)$ into $r_1$, we need to prove that the variables in $Y_2$ are covered. That is, we must prove the validity of the following condition:

$$\forall Y_2 \ [C[t_u](Y_2) \Rightarrow (\forall z_1, z_2, \ldots, z_k (\land_{p \in P} C[p] \land_{q \in Q} \neg C[q] \Rightarrow \lor_{p \in P} C[p \setminus R_2](Y_2)))]$$

Since the antecedent $\land_{p \in P} C[p] \land_{q \in Q} \neg C[q] = \text{true}$ in this example (because $\{z_1, z_2, \ldots, z_k\} = \emptyset$), the condition simplifies to:

$$\forall Y_2 \ (C[t_u](Y_2) \Rightarrow \lor_{p \in P} C[p \setminus R_2](Y_2)).$$
This is equivalent to testing the condition:

\[ \forall \ B_2 \ (21 < B_2) \Rightarrow (20 < B_2) \lor (20 < B_2) \lor (22 < B_2). \]

Since the implication is valid we can then conclude that the variable \( B_2 \) is covered in the materialization \( \nu(E_1, d) \) with respect to the given insert operation. That is, there cannot be a tuple in \( r_2 \) such that the \( B_2 \)-value matches the newly inserted tuple but matches no tuple already present in the view.

ii) Now consider the view \( E_2 = \{ \{ A_1, A_2, B_2, C_2 \}, \{ R_1, R_2 \}, (B_1 < B_2) \} \) which is exactly the same view as \( E_1 \) except that attribute \( B_1 \) is not visible. The project-join mapping for \( E_2 \) is given by:

\[
\begin{array}{cccccc}
 m_{\nu_2} : & A_1 & B_1 & A_2 & B_2 & C_2 \\
 1 & z_1 & 15 & 22 & 10 & p_1 \\
 1 & z_1 & 50 & 23 & 30 & p_3 \\
 1 & z_2 & 15 & 22 & 10 & p_3 \\
 1 & z_2 & 50 & 23 & 30 & p_4 \\
 2 & z_3 & 15 & 22 & 10 & q_1 \\
 2 & z_3 & 50 & 23 & 30 & p_5 \\
\end{array}
\]

Here \( Y_1 = \emptyset, Z_1 = \{ B_1 \}, Y_2 = \{ B_2 \}, \) and \( Z_2 = \emptyset. \) Again, to verify whether the variables in \( Y_2 \) are covered we need to test the condition:

\[
\forall \ Y_2 \ [C[t_u](Y_2) \Rightarrow \\
\{ \forall z_1, z_2, z_3 \ (\land_{p \in P} \ C[p] \land_{q \in Q} \neg C[q] \Rightarrow \lor_{p \in P} \ C[p] \land_{R_2} \}(Y_2)) \}].
\]

This is equivalent to testing the condition:

\[
\forall \ B_2 \ (21 < B_2) \Rightarrow \\
\{ \forall z_1, z_2, z_3 \ (z_1 < 22)(z_1 < 23)(z_2 < 22)(z_2 < 23)(z_3 < 23) \land (z_3 \geq 22) \\
\Rightarrow (z_1 < B_2) \lor (z_2 < B_2) \lor (z_3 < B_2) \}.
\]

Once again the implication holds and we can conclude that the variable \( B_2 \) is covered in the materialization \( \nu(E_2, d) \) with respect to the given insert operation. Therefore, the set of tuples \( T_u \) represents all tuples that should be inserted into the view as a result of the given update. For any inserted tuple \( t_u = (a_1, b_1) \) with \( b_1 < 21 \), the variable \( B_2 \) will not be covered in the materialization \( \nu(E_2, d) \).

iii) Finally, consider the view

\[
E_3 = \{ \{ A_1, B_1, A_2, B_2, C_2, A_3, B_3 \}, \{ R_1, R_2, R_3 \}, (B_1 = B_2) \land (A_2 = A_3) \}
\]

with materialization:
\[ v(E_3, d) : \begin{array}{cccccc}
A_1 & B_1 & A_2 & B_2 & C_2 & A_3 & B_3 \\
2 & 22 & 10 & 22 & 15 & 10 & 2 \\
3 & 23 & 30 & 23 & 50 & 30 & 4 \\
\end{array} \]

The corresponding project-join mapping is given by:

\[ m v_2 : \begin{array}{cccccc}
A_1 & B_1 & A_2 & B_2 & C_2 & A_3 & B_3 \\
2 & 22 & 10 & 22 & 15 & 10 & 2 & p_1 \\
2 & 22 & 10 & 22 & 15 & 30 & 4 & q_1 \\
2 & 22 & 30 & 23 & 50 & 10 & 2 & q_2 \\
2 & 22 & 30 & 23 & 50 & 30 & 4 & q_3 \\
3 & 23 & 10 & 22 & 15 & 10 & 2 & q_4 \\
3 & 23 & 10 & 22 & 15 & 30 & 4 & q_5 \\
3 & 23 & 30 & 23 & 50 & 10 & 2 & q_6 \\
3 & 23 & 30 & 23 & 50 & 30 & 4 & p_2 \\
\end{array} \]

Suppose that the operation INSERT \( (R_2, \{(30, 22, 60)\}) \) is applied to relation \( r_2 \). Then we can verify that \( Y_1 = \{B_1\} \), \( Z_1 = \emptyset \), \( Y_3 = \{A_3\} \), and \( Z_3 = \emptyset \). To be sure that the set \( T_u = \{(2, 22, 30, 22, 60, 30, 4)\} \) contains all the tuples to be inserted into \( v(E_3, d) \) as a result of the given insert operation, we need to prove that the variables in \( Y_1 \cup Y_3 \) are covered. That is, we need to prove the validity of the condition:

\[ \forall \ Y_1, Y_3 \left[ C[t_u](Y_1, Y_3) \Rightarrow \left( \forall_{p \in P} C[p \setminus R_1](Y_1) \wedge \forall_{p \in P} C[p \setminus R_3](Y_3) \right) \right]. \]

This is equivalent to testing the condition:

\[ \forall \ B_1, A_3 \left[ (B_1 = 22)(30 = A_3) \Rightarrow \left( (B_1 = 22)(10 = 10) \vee (B_1 = 23)(30 = 30) \right) \wedge \left( (22 = 22)(10 = A_3) \vee (23 = 23)(30 = A_3) \right) \right]. \]

Since the implication is valid we conclude that the variables \( B_1 \) and \( A_3 \) are covered in the view \( v(E_3, d) \) with respect to the given insert operation. Notice that components of the new tuple are not present in any one tuple in the view. In this example, for \( t_u = (a_2, b_2, c_2) \), \( A_3 \) is not covered if \( a_2 \not\in \{10, 30\} \) and \( B_1 \) is not covered if \( b_2 \not\in \{22, 23\} \).

\[ \square \]

**Theorem 1.4** Consider a view definition \( E = (A, R, C) \) with materialization \( v(E, d) \) and the operation INSERT \( (R_u, \{t_u\}) \). The set of tuples \( \pi_A(T_u) \) to be inserted into \( v(E, d) \) as a result of the insert operation includes all tuples that must be inserted into the view if and only if all variables in \( \bigcup_{i=1}^{m} \pi_A(Y_i \cup Z_i) \), \( Y_i = \alpha(R_i) \cap \alpha(C) \cap A^+ \) and \( Z_i = [\alpha(R_i) \cap \alpha(C)] - A^+ \), are covered in the materialization \( v(E, d) \) with respect to the insertion.

**Proof:** (Sufficiency) If the condition for coverage holds, then it is guaranteed that any combination of values for the variables in \( \bigcup_{i=1}^{m} \pi_A(Y_i \cup Z_i) \) that make the condition
\[ C[t_u](Y_1, Z_1, \ldots, Y_{u-1}, Z_{u-1}, Y_{u+1}, Z_{u+1}, \ldots, Y_m, Z_m) \]

hold, will be present as part of some subset of tuples currently stored in \( v(E, d) \). Each of the conjuncts in the consequent of Condition (1) of Definition 5.4, namely,

\[ \forall z_1, z_2, \ldots, z_k \ (\wedge_{p \in P} C[p] \wedge_{q \in Q} \neg C[q] \Rightarrow \vee_{p \in P} C[p \setminus R_i](Y_i, Z_i)), \]

assures that all combinations of values for the variables \( Y_i, Z_i \) which when combined with the inserted tuple \( t_u \) will make the antecedent of Condition (1) evaluate to true (i.e., creating a new insertion into the view) are already represented in the view. Building the set \( T_u \) involves generating all tuples that result from combining \( t_u \) with the tuples in \( v(E, d) \). Hence \( T_u \) will contain all tuples that must be inserted into the view as a result of inserting \( t_u \) into \( r_u \).

(Necessity) If the condition for coverage does not hold, then there is an assignment of values to the variables \( \bigcup_{i=1}^{m} (Y_i \cup Z_i) \) such that

\[ C[t_u](\hat{Y}_1, \hat{Z}_1, \ldots, \hat{Y}_{u-1}, \hat{Z}_{u-1}, \hat{Y}_{u+1}, \hat{Z}_{u+1}, \ldots, \hat{Y}_m, \hat{Z}_m) = \text{true} \]  

(2)

which is not available from tuples currently stored in the view \( v(E, d) \). This assignment of values causes at least one conjunct of the consequent of Condition (1) to be false. Assume that such a conjunct is given by

\[ \forall z_1, z_2, \ldots, z_k \ (\wedge_{p \in P} C[p] \wedge_{q \in Q} \neg C[q] \Rightarrow \vee_{p \in P} C[p \setminus R_h](\hat{Z}_h)), \]  

(3)

where \( Z_h \subseteq \alpha(R_h) \). We can then construct databases \( d_1 = \{ r_1^1, \ldots, r_m^1 \} \) and \( d_2 = \{ r_1^2, \ldots, r_m^2 \} \) such that \( v(E, d_1) = v(E, d_2) = v(E, d) \). Database \( d_1 \) contains the values \( \hat{Z}_h \) in some tuple of relation \( r_h \), but \( d_2 \) does not.

To construct the databases \( d_1 \) and \( d_2 \) we use the project-join mapping \( m_\nu \) for \( \nu \) corresponding to the given view \( v(E, d) \). We assign surrogate values to all variables \( \{ z_1, z_2, \ldots, z_k \} \) appearing in tuples from \( m_\nu \) in such a way that

\[ \wedge_{p \in P} C[p](\hat{z}_1, \ldots, \hat{z}_k) \wedge \neg C[q](\hat{z}_1, \ldots, \hat{z}_k) = \text{true}. \]

Clearly, such values can always be obtained. Relations \( r_1^l, r_2^j, 1 \leq l \leq m, \) are assigned the set of tuples \( \pi_{\alpha(R_i)}(m_\nu) \). In addition, we add a tuple \( t_j \) to every relation \( r_j^2, 1 \leq j \leq m, j \neq u \). Each tuple \( t_j \) is built as follows: \( t_j[Z_j] = \hat{Z}_j, t_j[Y_j] = \hat{Y}_j \), and \( t_j[X_j] = p[X_j], X_j \subseteq \alpha(R_j) - (Z_j \cup Y_j) \), for some \( p \in m_\nu \). Clearly, \( v(E, d_1) = v(E, d_2) = v(E, d) \).

Now suppose tuple \( t_u \) is inserted into \( r_u^1 \) and \( r_u^2 \) to obtain the updated databases \( d'_1 \) and \( d'_2 \). Because of Condition (3) we know that \( v(E, d'_1) = v(E, d_1) \) and because of Condition (2) we know that a new tuple will be inserted into \( v(E, d'_2) \) thus \( v(E, d'_1) \neq v(E, d'_2) \). Therefore, there exists a database \( d_2' \) where \( v(E, d_2') = v(E, d) \) for which \( \pi_A(T_u) \) does not contain all tuples that must
be inserted into the view. Thus, by Lemma 3.1 the update cannot be conditionally autonomously computable.

□

Corollary 5.5 The effect of inserting the tuple \( t_u \) into \( r_u \) on the view is conditionally autonomously computable with respect to the view materialization \( v(E, d) \) if and only if the conditions of Theorems 1.3 and 1.4 hold.

□

We illustrate the full algorithm resulting from Theorems 1.3 and 1.4 through an example.

Example 5.4 Consider two relation schemes \( R_1(A_1, B_1) \), \( R_2(A_2, B_2, C_2) \), and their corresponding relations:

\[
\begin{array}{ccc}
  r_1 : & A_1 & B_1 \\
  1 & 18 & \\
  2 & 24 & \\
  3 & 28 & \\
  4 & 25 & \\
  5 & 21 & \\
\end{array}
\begin{array}{ccc}
  r_2 : & A_2 & B_2 & C_2 \\
  17 & 14 & 11 \\
  21 & 15 & 30 \\
  26 & 50 & 23 \\
  3 & 25 & 42 \\
\end{array}
\]

Consider a view defined by

\[ E = \{(A_1, B_1, A_2, B_2), \{R_1, R_2\}, [(A_1 = 2)(B_1 < A_2) \lor (A_1 = 5)(B_1 > C_2) \lor (B_1 = B_2)]\} \]

and its corresponding materialization:

\[
\begin{array}{cccc}
  v(E, d) : & A_1 & B_1 & A_2 & B_2 \\
  2 & 24 & 26 & 50 & \\
  4 & 25 & 3 & 25 & \\
  5 & 21 & 17 & 14 & \\
\end{array}
\]

Assume the following update is applied to the database: \( \text{INSERT} \ (R_1, \{(2, 25)\}) \).

Step A: First, we build \( V \), \( m_V \), and \( T_u \).

\[
\begin{array}{cccccc}
  V : & A_1 & B_1 & A_2 & B_2 & C_2 \\
  2 & 24 & 26 & 50 & z_1 & \\
  4 & 25 & 3 & 25 & z_2 & \\
  5 & 21 & 17 & 14 & z_3 & \\
\end{array}
\begin{array}{cccccc}
  m_V : & A_1 & B_1 & A_2 & B_2 & C_2 \\
  2 & 24 & 26 & 50 & z_1 & p_1 \\
  2 & 24 & 3 & 25 & z_2 & q_1 \\
  2 & 24 & 17 & 14 & z_3 & q_2 \\
  4 & 25 & 26 & 50 & z_1 & q_3 \\
  4 & 25 & 3 & 25 & z_2 & p_2 \\
  4 & 25 & 17 & 14 & z_3 & q_4 \\
  5 & 21 & 26 & 50 & z_1 & q_5 \\
  5 & 21 & 3 & 25 & z_2 & q_6 \\
  5 & 21 & 17 & 14 & z_3 & p_3 \\
\end{array}
\]
$T_u: \begin{array}{ccccc}
A_1 & B_1 & A_2 & B_2 & C_2 \\
2 & 25 & 26 & 50 & z_1 \\
2 & 25 & 3 & 25 & z_2 \\
2 & 25 & 17 & 14 & z_3 
\end{array}$

The first tuple $t = (2, 25, 26, 50, z_1)$ in $T_u$ can be safely accepted because the condition of Lemma I.1, shown below, holds:

$$\forall z_1 \[[((2 = 2)(24 < 26) \lor (2 = 5)(24 > z_1) \lor (24 = 50)) \land \neg((4 = 2)(25 < 26) \lor (4 = 5)(25 > z_1) \lor (25 = 50)) \land \neg((5 = 2)(21 < 26) \lor (5 = 5)(21 > z_1) \lor (21 = 50)) \Rightarrow (2 = 2)(25 < 26) \lor (2 = 5)(25 > z_1) \lor (25 = 50)] \]$$

In the above condition, we only need to consider tuples from $m_\mathcal{V}$ referring to variables $z_1$, namely, $p_1$, $q_3$, and $q_5$. For the second tuple $t = (2, 25, 3, 25, z_2)$ in $T_u$ we also test the condition of Lemma I.1.

$$\forall z_2 \[\neg((2 = 2)(24 < 3) \lor (2 = 5)(24 > z_2) \lor (24 = 25)) \land ((4 = 2)(25 < 3) \lor (4 = 5)(25 > z_2) \lor (25 = 25)) \land \neg((5 = 2)(21 < 3) \lor (5 = 5)(21 > z_2) \lor (21 = 25)) \Rightarrow (2 = 2)(25 < 3) \lor (2 = 5)(25 > z_2) \lor (25 = 25)] \]$$

Clearly, the above condition holds, and the tuple $t = (2, 25, 3, 25, z_2)$ in $T_u$ can be safely accepted into the view. Finally, for the third tuple $t = (2, 25, 17, 14, z_3)$ in $T_u$ we test the condition of Lemma I.2.

$$\forall z_3 \[\neg((2 = 2)(24 < 17) \lor (2 = 5)(24 > z_3) \lor (24 = 14)) \land \neg((4 = 2)(25 < 17) \lor (4 = 5)(25 > z_3) \lor (25 = 14)) \land ((5 = 2)(21 < 17) \lor (5 = 5)(21 > z_3) \lor (21 = 14)) \Rightarrow \neg((2 = 2)(25 < 17) \lor (5 = 2)(25 > z_3) \lor (25 = 14)] \]$$

Clearly, the above condition holds, and the tuple $t = (2, 25, 17, 14, z_3)$ in $T_u$ can be safely rejected from the view.

**Step B:** Here we are interested in finding out whether the new tuples assembled in Step A represent all tuples that need to be inserted into the view. We can verify that in this example $Y_2 = \{A_2, B_2\}$ and $Z_2 = \{C_2\}$. Testing whether the variables $Y_2 \cup Z_2$ are covered requires testing the following condition:

$$\forall Y_2, Z_2 (C[t_u](Y_2, Z_2) \Rightarrow \forall z_1, z_2, z_3 \land_{p \in P} C[p] \land_{q \in Q} \neg C[q] \Rightarrow \lor_{p \in P} C[p \setminus R_2](Y_2, Z_2)).$$
This is equivalent to testing the condition:

\[ \forall \ A_2, B_2, C_2 \ [((2 = 2)(25 < A_2) \vee (2 = 5)(25 > C_2) \vee (25 = B_2)) \Rightarrow \\
(\forall z_1, z_2, z_3 \\
((2 = 2)(24 < 26) \vee (2 = 5)(24 > z_1) \vee (24 = 50)) \\
\wedge ((4 = 2)(25 < 3) \vee (4 = 5)(24 > z_2) \vee (25 = 25)) \\
\wedge ((5 = 2)(21 < 17) \vee (5 = 5)(21 > z_3) \vee (21 = 14)) \]

Because none of \( z_1, z_2, z_3 \) remain in the conclusion, this reduces to

\[ \forall \ A_2, B_2, C_2 \ [((25 < A_2) \vee (25 = B_2)) \Rightarrow \\
((24 < A_2) \vee (24 = B_2) \vee (25 = B_2) \vee (21 > C_2) \vee (21 = B_2))] \]

The implication is valid, and therefore, attributes \( A_2, B_2, \) and \( C_2 \) are covered in the view with respect to the given insert operation. Hence, the set of tuples \( T_u \) generated at Step A as a result of the insertion of the tuple \((2, 25)\) into \( \tau_1 \) are all tuples that need to be inserted into \( v(E, d) \) to bring it up to date.

\[ \square \]

## 6 Special cases

The conditions required to decide whether an insertion into or deletion from a base relation is conditionally autonomously computable are expensive to test since, in general, the size of the conditions are a function of the number of tuples in the view. The purpose of this section is to explore some special cases when the tests may be performed at a reasonable cost.

### 6.1 Deletions

To determine whether a delete operation is conditionally autonomously computable involves testing the conditions of Theorem D.3 (which in turn involves testing the conditions of Lemmas D.1 and D.2) for every tuple \( t \in V \). We note two cases that lead to substantial savings:
6 SPECIAL CASES

1. The first simplification occurs when \( \alpha(C_D) - A^+ = \emptyset \) because then the tests of Lemmas D.1 and D.2 reduce to evaluating \( C_D[t] \) for each \( t \in V \). This is equivalent to asserting that the deletion is unconditionally autonomously computable.

2. The second simplification is obtained by dropping the term \( \bigwedge_{q \in Q} \neg C[q] \) from the antecedent of the conditions of Lemmas D.1 and D.2, still resulting in a sufficient condition for deciding the deletion of a tuple \( t \in V \). If the simplified condition of Lemma D.2 \( \bigwedge_{p \in P} C[p] \Rightarrow C_D[t] \) holds, then the tuple \( t \) can be safely deleted. However, if the condition does not hold, then it may still be the case that \( t \) could be safely deleted if we were to use the information provided by the condition \( \bigwedge_{q \in Q} \neg C[q] \). Similarly, the condition of Lemma D.1 can be simplified to obtain a sufficient condition to let \( t \) stay in the view.

6.2 Insertions

Determining whether an insert operation is unconditionally autonomously computable involves testing the conditions of Theorem I.3 for every tuple \( t \in T_u \) as well as testing the condition of Theorem I.4 for every tuple \( t_u \) inserted into \( r_u \).

The size of the conditions of Lemmas I.1 and I.2 mentioned in Theorem I.3 depend on the number of free variables in the consequents \( C[t] \) and \( \neg C[t] \), respectively. Let \( W \) denote \( \alpha(C) - (A^+ \cup R_u) \), the set of free variables in the consequents of Lemmas I.1 and I.2. In the same way as with deletions, we note two cases that lead to substantial savings when testing the conditions established by Lemmas I.1 and I.2:

- If \( W = \emptyset \), then the tests of Lemmas I.1 and I.2 reduce to evaluating \( C[t] \), \( t \in T_u \).
- The condition of Lemma I.1 can be simplified by eliminating the condition \( \bigwedge_{q \in Q} \neg C[q] \) from the antecedent, still resulting in a sufficient condition for the acceptance of the tuple \( t \). If the simplified condition \( \bigwedge_{p \in P} C[p] \Rightarrow C[t] \) holds, then the tuple \( t \) can be safely inserted into the view. However, if the simplified condition does not hold, then it may be possible to safely accept \( t \) if we were to use the information provided by the expression \( \bigwedge_{q \in Q} \neg C[q] \). Similarly, the condition of Lemma I.2 can be simplified to obtain a sufficient condition for the rejection of tuple \( t \).

In general, the number of atomic terms in the antecedent of the conditions of Lemmas I.1, I.2, D.1 and D.2, namely \( \bigwedge_{p \in P} C[p] \bigwedge_{q \in Q} \neg C[q] \), is \( \|v(E, d)\|^m|C| \), where \( \|v(E, d)\| \) denotes the number of tuples in the view, \( m \) is the number of different relation schemes in the view definition, and \( |C| \) denotes the number of atomic terms in the condition \( C \). However, not all the Boolean expressions derived from \( C \) in the antecedent refer to free variables in the consequent \( C[t] \). In Example 5.2 on page 16, for instance, the variables \( z_1, z_2 \) for attribute \( B_2 \) are not free in \( C[t] \) because the newly
inserted tuple assigns a value to $B_2$; consequently, restrictions on values of $B_2$ in order to satisfy the condition $(B_1 > B_2)$ in $p \in P$ have no effect on the truth value of $C[t]$. Hence, we want to know how many Boolean expressions derived from $C$ must be included in the antecedent to be able to test the conditions of Lemmas I.1, I.2, D.1 and D.2 correctly.

If the consequent $C[t]$ includes free variables from $k$ distinct relation schemes, then we want to know how many rows $p, q \in m_P$ refer to these free variables. Such rows represent exactly the conditions $C[p], C[q]$ from the expression $\wedge_{p \in P} C[p] \wedge_{q \in Q} -C[q]$ that must be tested. To solve this, consider the problem of counting vectors of size $k$, where each position may be occupied by one of $b$ different elements. The value $k \leq m$ represents the number of relations schemes containing at least one variable in $Z$. In our case $b = |v(E, d)|$, since each tuple in the view can contribute components to each of the positions. The total number of vectors is $b^k$. The value of $C[t]$ specifies the vector $v_1, v_2, \ldots, v_k$ and we are asked to find the total number of vectors that contain the element $v_1$ in position 1 or the element $v_2$ in position 2 and so on, until position $k$. This is given by $b^k - (b - 1)^k$.

Also, we have to count all different rows of $m_P$ that refer to the given vector $v_1, v_2, \ldots, v_k$; each such row is associated with one of the remaining $m - k$ relation schemes. Thus, the number of Boolean expressions in the antecedent of the conditions given by Lemmas I.1, I.2, D.1, and D.2 is:

$$(|v(E, d)|)^m - k \left[ (|v(E, d)|)^k - (|v(E, d)| - 1)^k \right].$$

The cost of testing the condition of Theorem I.4 is influenced by the sizes of the sets of variables $Z = [\alpha(R) \cap \alpha(C)] - A^+$ and $Y = \alpha(R) \cap \alpha(C) \cap A^+$. If all the attributes in the inserted relation that participate in the view condition are reconstructible, i.e., $Z = \emptyset$, then all the conditions of the form:

$$\forall z_1, z_2, \ldots, z_k \wedge_{p \in P} C[p] \wedge_{q \in Q} -C[q] \Rightarrow \forall_{p \in P} C[p \setminus R_i](Y_i, Z_i)$$

reduce to conditions of the form:

$$\forall_{p \in P} C[p \setminus R_i](Y_i)$$

for $1 \leq i \leq m, i \neq u$. This, of course, greatly reduces the cost of testing for coverage. However, even though the complexity of the condition for coverage simplifies substantially when $Z = \emptyset$, it is still necessary to test that all the relevant values for the variables $Y$ are somehow represented in the view. Thus, the size of the simplified condition still depends on the number of tuples in the view. The smaller the number of relations involved in the view definition, the cheaper to evaluate the condition.

As a final special case, the test for coverage given by Definition 5.4 does not require a formal test for satisfiability in cases when $Z = \emptyset, m = 2, C$ is a conjunctive condition involving terms of the form $(x = y)$ or $(x = c)$ where $x$ and $y$ are variables and $c$ is a constant. Here we only need to scan the view $|Y|$ times looking for the values that make $C[t_u](Y)$ evaluate to true.
7 Conclusions

We started this research by asking ourselves the question: How far can we carry the idea of autonomously computable updates if in the process of deciding whether an update is autonomously computable we have access to the data values stored in the view?

As a result, we have been able to establish necessary and sufficient conditions for the detection of conditionally autonomously computable updates for the case when the updates consist of insertions and deletions and the views are defined by PSJ-expressions. The results of this paper give us insight into what is involved when we try to exploit the information stored in the view for inferring values stored in database. This is an important step towards finding efficient ways of maintaining a materialized view without requiring to access base data.

The conditions of Theorems I.3, I.4, and D.3 are, in general, expensive to test so we have provided some special cases where the conditions to be tested can be greatly simplified.

We can apply the results of this research to the process of designing autonomously maintainable materialized views by allowing the view definition to contain exactly those attributes required to be able to carry out insertions and deletions autonomously without additional information from the database.

It still remains to be seen how the ideas presented in this paper perform in practice. Thus, a performance study of these ideas in realistic database settings is a direction for further research. We leave it as an open problem to determine whether there is a simpler characterization of conditionally autonomously computable updates when we assume further constraints on the database resulting from the presence of keys or other data dependencies.

References


REFERENCES


